Abstract DPLL and Abstract DPLL Modulo Theories

Robert Nieuwenhuis*, Albert Oliveras*, and Cesare Tinelli **

Abstract. We introduce *Abstract DPLL*, a general and simple abstract rule-based formulation of the Davis-Putnam-Logemann-Loveland (DPLL) procedure. Its properties, such as soundness, completeness or termination, immediately carry over to the modern DPLL implementations with features such as non-chronological backtracking or clause learning. This allows one to formally reason about practical DPLL algorithms in a simple way. In the second part of this paper we extend the framework to *Abstract DPLL modulo theories*. This allows us to express—and formally reason about—state-of-the-art concrete DPLL-based techniques for satisfiability modulo background theories, such as the different *lazy* approaches, or our DPLL(T) framework.

1 Introduction

Most state-of-the-art SAT solvers [MMZ+01,GN02] today are based on different variations of the Davis-Putnam-Logemann-Loveland (DPLL) procedure [DP60,DLL62], a procedure for deciding the satisfiability of propositional formulas in conjunctive normal form.

Starting essentially with the pioneering work on the GRASP [MSS99] and SATO [Zha97] systems, 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, e.g., the *2-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 *non-chronological backtracking*, *conflict-driven lemma learning*, and *restarts*.

Because of their success, both the DPLL procedure and its enhancements have been recently adapted to satisfiability problems in more expressive logics than propositional logic. In particular, they have been used to build efficient solvers for the satisfiability of (certain classes of) ground first-order formulas with respect to theories such as the theory of equality, of the integer/real numbers, or of arrays [ACG00,ABC+02,BDS02,dMR02,FJOS03,GHN+04].

Altogether, it has become non-trivial to reason formally about the properties of such enhanced DPLL procedures and their extensions to satisfiability modulo

* Technical University of Catalonia, Barcelona `www.lsi.upc.es/~roberto|~oliveras`.
Partially supported by Spanish Min. of Educ. and Science by the LogicTools project (TIN2004-03382, both these authors), and FPU grant AP2002-3533 (Oliveras).
** Dept. of Computer Science, The University of Iowa, `www.cs.uiowa.edu/~tinelli`.
Partially supported by Grant No. 237422 from the National Science Foundation.

theories (SMT). However, so far there have been no attempts to do so in the literature, to our knowledge at least, except for a work by Tinelli [Tin02] (one of these authors). That work describes DPLL and DPLL modulo theories at an abstract, formal level by means of a sequent-style logical calculus. This calculus consists of a few deterministic derivation rules, modelling the constraint propagation mechanism of the DPLL procedure, and one branching rule, modelling the non-deterministic guessing step of DPLL. Because of the branching rule the calculus produces *derivation trees*. As a consequence, it can explictly model neither backtracking (chronological or not) nor lemma learning—they are metalogical features for the calculus. Also, the calculus implicitly assumes the procedure to keep track of the current truth values of all clauses, which is not the case in practical implementations.

In this paper we address these limitations of Tinelli's calculus by modelling the DPLL procedure and its SMT extensions as *transitions systems*. While still as declarative in nature as the calculus in [Tin02], our transition systems can explicitly model various features of state-of-the-art DPLL-based solvers, thus bridging the gap between abstract calculi for DPLL and actual implementations.

In Section 2, using transition systems defined by means of conditional transition rules, we introduce general and simple abstract formulations of several variants of propositional DPLL, and discuss their soundness, completeness, and termination. These properties immediately carry over to modern DPLL implementations with features such as non-chronological backtracking and learning. In fact, we also explain and formalize what is done by the different implementations. For example, we explain how different systems implement our backjumping rule, how devices such as implication graphs are just one possibility for computing new lemmas, and how standard backtracking is a special case of the backjumping rule.

We also provide a general and simple termination argument for DPLL procedures that does not depend on an exhaustive enumeration of all truth assignments; instead, it cleanly expresses that a search state becomes more advanced if an additional unit is deduced, the higher up in the search tree the better—which is the very essence of the idea of backjumping.

Our transition systems allow one to formally reason about practical DPLL implementations in a simple way, which to our knowledge had not been done before. In Section 3 we extend the framework to *Abstract DPLL modulo theories*. This allows us to express—and formally reason about—most state-of-the-art DPLL-based techniques for satisfiability modulo background theories, such as various so-called *lazy* approaches [ACG00,ABC$^+$02,BDS02,dMR02,FJOS03] and our own DPLL(T) framework [GHN$^+$04].

2 The Abstract DPLL Procedure

The DPLL procedure works by trying to build incrementally a satisfying truth assignment for a given propositional formula F in conjunctive normal form. At each step, the current assignment M for F is augmented either by a process of *boolean constraint propagation*, which deduces deterministically from M and F the truth value of additional variables of F, or by a non-deterministic guess, or *decision*, on the truth value of one of the remaining undefined variables.

Modern implementations of DPLL use efficient constraint propagation algorithms, and sophisticated backtracking mechanisms for recovering from wrong decisions. We provide here a general abstract framework for describing both constraint propagation and backtracking in DPLL-based systems.

In this section we deal with propositional logic. Atoms are propositional symbols from a finite set P. If $p \in P$, then p is a *positive* literal and $\neg p$ is a *negative* literal. 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 set of literals and a *cnf* (formula) is a set of clauses. 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* otherwise. M is *total* if no literal of P is undefined in M. A clause C is true in M if $C \cap M \neq \emptyset$, is false in M, denoted $M \models \neg C$, if all its literals are false in M, and is undefined otherwise. A cnf F is true in M (or satisfied by M), denoted $M \models F$, if all its clauses are true in M. In that case, M is called a *model* of F. If F has no models then it is *unsatisfiable*. We write $F \models C$ ($F \models F'$) if the clause C (cnf F') is true in all models of F. If $F \models F'$ and $F' \models F$, we say that F and F' are logically equivalent. We denote by $C \vee l$ the clause D such that $l \in D$ and $C = D \setminus \{l\}$.

2.1 The Basic DPLL Procedure

Here, a DPLL procedure will be modeled by a *transition system*: a set of *states* together with a relation, called the *transition relation*, over these states. States will be denoted by (possibly subscripted) S. We write $S \Longrightarrow S'$ to mean that the pair (S, S') is in the transition relation, and then say that S' is *reachable* from S in one *transition step*. We denote by $\Longrightarrow^*$ the reflexive-transitive closure of $\Longrightarrow$. We write $S \Longrightarrow^! S'$ if $S \Longrightarrow^* S'$ and S' is a *final* state, i.e., if $S' \Longrightarrow S''$ for no S''.

A state is either *fail* or a pair $M \parallel F$, where F is a finite set of clauses and M is a sequence of *annotated literals*. We will denote the empty sequence of literals by $\emptyset$, unit sequences by their only literal, and the concatenation of two sequences by simple juxtaposition. We will not go into a complete formalization of annotated literals; it suffices to know that some literals l will be annotated as being *decision literals*; this fact will be denoted here by writing l^{d} (roughly, decision literals are the ones that have been added to M by the Decide rule given below). Most of the time the sequence M will be simply seen as a set of literals, denoting an assignment, i.e., ignoring both the annotations and the fact that M is a sequence and not a set.

In what follows, the transition relation will be defined by means of (conditional) transition rules. If F is a cnf formula and C is a clause, we will sometimes write F, C in the second component of a state as a shorthand for $F \cup \{C\}$.

Definition 1. *The* Basic DPLL system *consists of the following transition rules:*

UnitPropagate :

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

Decide :

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

Fail :

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

Backjump :

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

Below we will show that the transition relation *terminates* when starting from $\emptyset \parallel F$, that is, there exist no infinite sequences of the form $\emptyset \parallel F \implies S_1 \implies \ldots$, and we will define a *Basic DPLL procedure* to be *any* procedure taking an input cnf F and computing a sequence $\emptyset \parallel F \implies^{!} S$.

Of course, actual DPLL implementations may use the above rules in more restrictive ways, using particular application strategies. For example, many systems will eagerly apply UnitPropagate, but this is not necessary; in fact, below we will show that *any strategy is adequate*: the final state produced by the strategy will be either *fail*, when F is unsatisfiable, or else a state of the form $M \parallel F'$ where M is a model of F. This result holds even if UnitPropagate is not applied at all. Similarly, most implementations will try to minimize the number of applications of Decide. Others may apply it only with literals l belonging to some clause that is not yet true in M (in that case the procedure can also terminate if M is a non-total model).

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

$$\begin{array}{rcll} \emptyset & \parallel & 1 \vee \overline{3},\ \overline{1} \vee \overline{4} \vee 5 \vee 2,\ \overline{1} \vee \overline{2} & \implies \quad \text{(Decide)} \\ \mathbf{3} & \parallel & 1 \vee \overline{3},\ \overline{1} \vee \overline{4} \vee 5 \vee 2,\ \overline{1} \vee \overline{2} & \implies \quad \text{(UnitPropagate)} \\ \mathbf{3}\ 1 & \parallel & 1 \vee \overline{3},\ \overline{1} \vee \overline{4} \vee 5 \vee 2,\ \overline{1} \vee \overline{2} & \implies \quad \text{(UnitPropagate)} \\ \mathbf{3}\ 1\ \overline{\mathbf{2}} & \parallel & 1 \vee \overline{3},\ \overline{1} \vee \overline{4} \vee 5 \vee 2,\ \overline{1} \vee \overline{2} & \implies \quad \text{(Decide)} \\ \mathbf{3}\ 1\ \overline{\mathbf{2}}\ \mathbf{4} & \parallel & 1 \vee \overline{3},\ \overline{1} \vee \overline{4} \vee 5 \vee 2,\ \overline{1} \vee \overline{2} & \implies \quad \text{(UnitPropagate)} \\ \mathbf{3}\ 1\ \overline{\mathbf{2}}\ \mathbf{4}\ 5 & \parallel & 1 \vee \overline{3},\ \overline{1} \vee \overline{4} \vee 5 \vee 2,\ \overline{1} \vee \overline{2} & \quad \text{Final state: model found.} \quad \square \end{array}$$

Concerning the rules Fail and Backjump, we will show below that if in some state $M \parallel F$ there is a *conflict*, i.e., a clause of F that is false in M, it is always the case that either Fail applies (if there are no decision literals in M) or Backjump applies (if there is at least one decision literal in M). In fact, in most implementations Backjump is only applied when such a conflict arises, this is why it is usually called *conflict-driven backjumping*. Note that M can be seen as a sequence $M_0\, l_1\, M_1 \ldots l_k\, M_k$, where the l_i are all the decision literals in M. As in actual DPLL implementations, such a state is said to be in *decision level* k, and the literals of each $l_i\, M_i$ are said to belong to decision level i.

Example 3. Another example of application of the Basic DPLL rules is:

$$
\begin{array}{rcll}
\emptyset & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow \quad \text{(Decide)} \\
\mathbf{1} & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow \quad \text{(UnitPropagate)} \\
\mathbf{1}\ 2 & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow \quad \text{(Decide)} \\
\mathbf{1}\ 2\ \mathbf{3} & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow \quad \text{(UnitPropagate)} \\
\mathbf{1}\ 2\ \mathbf{3}\ 4 & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow \quad \text{(Decide)} \\
\mathbf{1}\ 2\ \mathbf{3}\ 4\ \mathbf{5} & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow \quad \text{(UnitPropagate)} \\
\mathbf{1}\ 2\ \mathbf{3}\ 4\ \mathbf{5}\ \overline{6} & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow \quad \text{(Backjump)} \\
\mathbf{1}\ 2\ \overline{5} & \parallel & \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} &
\end{array}
$$

Indeed, before the application of Backjump there was a conflict: the clause $6\vee\overline{5}\vee\overline{2}$ is false in $\mathbf{1}\ 2\ \mathbf{3}\ 4\ \mathbf{5}\ \overline{6}$. We have backjumped from decision level 3 to decision level 1, whereas standard backtracking would reverse only the last decision, and return to $\mathbf{1}\ 2\ \mathbf{3}\ 4\ \overline{5}$ (decision level 2). The Backjump rule applies here because we can take $\overline{1}\vee\overline{5}$ playing the role of the *backjump clause* $C \vee l'$ in the definition of the rule. In fact, one can always take a disjunction of negated decision literals for this (see the proof of Lemma 6). But in practice one can usually find better backjump clauses by *conflict analysis*, that is, by analyzing the so called *conflict graph* (see, e.g., [MSS99] for details). □

The Backjump rule makes progress in the search by returning to a strictly lower decision level, but with the additional information given by the literal l' that is added to it. In most DPLL implementations the backjump clause $C \vee l'$ is added to the clause set as a *learned* clause (*conflict-driven clause learning*). However, in this Basic system the second component of each state (the clause set) remains unchanged; this will change in Subsection 2.3 when the learning rule is added. In fact, for some readers it may be surprising that backjumping can be done without clause learning. Such a distinction gives the system more flexibility, allowing it to model, for example, the original DPLL procedure [DLL62].

2.2 Correctness of Basic DPLL

In what follows, (possibly subscripted) F and M will always denote finite clause sets and annotated literal sequences, respectively.

Lemma 4. *If $\emptyset \parallel F \Longrightarrow^* M \parallel F$ then the following hold.*

1. *All the atoms in M are atoms of F.*
2. *M contains no literal more than once and is indeed an assignment, i.e., it contains no pair of literals of the form p and $\neg p$.*
3. *If M is of the form $M_0\ l_1\ M_1\ \ldots\ l_n\ M_n$, where $l_1, \ldots, l_n$ are all the decision literals of M, then $F \cup \{l_1, \ldots, l_i\} \models M_i$ for all i in $0 \ldots n$.*

Theorem 5 (Termination). *There exist no infinite sequences of the form $\emptyset \parallel F \Longrightarrow S_1 \Longrightarrow \ldots$*

Proof. We define a well-founded strict partial ordering $\succ$ on states, and show that each rule application $M \parallel F \Longrightarrow M' \parallel F'$ is decreasing with respect to this ordering, i.e., $M \parallel F \ \succ\ M' \parallel F'$.

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

Let N be the number of distinct atoms (propositional variables) in F. It is not difficult to show that p, p' and the length of M and M' are always smaller than or equal to N. Define $m(M)$ to be $N - \mathit{length}(M)$, that is, $m(M)$ is the number of literals "missing" in M for M 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

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

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

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 rule applications are decreasing with respect to $\succ$ if *fail* 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). □

In the previous termination proof one can observe that DPLL search *progresses* (that is, it makes progress w.r.t. $\succ$) by adding a literal to the current decision level (by UnitPropagate), by adding an additional decision level (Decide) or, which is especially interesting, by what the Backjump rule does, i.e., adding an additional literal to a previous decision level, even if all the work done in later decision levels is "thrown away".

Note that it is not trivial to check whether a state is final, because of the Backjump rule. But in practice Backjump is applied only if there is a conflict. If in a state $M \parallel F$ there is no conflict, and UnitPropagate and Decide are not applicable either (i.e., there are no undefined literals in M), then one can of course stop because M is a model of F.

Lemma 6. *Assume that $\emptyset \parallel F \Longrightarrow^* M \parallel F$ and that $M \models \neg D$ for some clause D 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 $M_0\ l_1\ M_1\ \ldots\ l_n\ M_n$ for some $n > 0$, where $l_1, \ldots, l_n$ are all the decision literals of M. Since $M \models \neg D$, we have, due to Lemma 4-3, that $F \cup \{l_1, \ldots, l_n\} \models \neg D$. Now consider any i in $1 \ldots n$ such that $F \cup \{l_1, \ldots, l_i\} \models \neg D$, and j in $0 \ldots i-1$ such that $F \cup \{l_1, \ldots, l_j, l_i\} \models \neg D$. We will show that then backjumping to decision level j is possible.

Let C be the clause $\neg l_1 \vee \ldots \vee \neg l_j$, and note that M is of the form $M'\ l_{j+1}\ N$.

Then Backjump applies to $M \parallel F$ as: $M'\ l_{j+1}\ N \parallel F \Longrightarrow M'\ \neg l_i \parallel F$ because for the clause $C \vee \neg l_i$ all three conditions of the Backjump rule hold. In fact:

(i) $F \models C \vee \neg l_i$ because $F \cup \{l_1, \ldots, l_j, l_i\} \models \neg D$ implies, being D a clause in F, that $F \models \neg l_1 \vee \ldots \vee \neg l_j \vee \neg l_i$. We also obviously have that $M' \models \neg C$.

(ii) $\neg l_i$ is undefined in M' (by Lemma 4-2) and

(iii) either l_i or $\neg l_i$ occurs in a clause of F (by Lemma 4-1). □

It is interesting to observe that the smaller the j in the previous proof the better, because one can backjump "higher up". Note also that, if we take i to be n and j to be $n-1$, the Backjump rule models standard backtracking.

Lemma 7. *If $\emptyset \parallel F \Longrightarrow^{!} M \parallel F$, then $M \models F$.*

Definition 8. *A* Basic DPLL procedure *is any procedure taking an input cnf F and computing a sequence $\emptyset \parallel F \Longrightarrow^{!} S$.*

Now, we can prove that our Basic DPLL system, and hence any Basic DPLL procedure, provides a decision procedure for the satisfiability of cnf formulas.

Theorem 9. *The Basic DPLL system provides a decision procedure for the satisfiability of cnf formulas F, that is:*

1. *$\emptyset \parallel F \Longrightarrow^{!}$ fail if, and only if, F is unsatisfiable.*
2. *$\emptyset \parallel F \Longrightarrow^{!} M \parallel F$ if, and only if, F is satisfiable.*
3. *If $\emptyset \parallel F \Longrightarrow^{!} M \parallel F$ then M is a model of F.*

Proof. For the left-to-right implication of property 1: if $\emptyset \parallel F \Longrightarrow^{!} fail$ then there is some state $M \parallel F$ such that $\emptyset \parallel F \Longrightarrow^{*} M \parallel F \Longrightarrow fail$, there is no decision literal in M and $M \models \neg C$ for some clause C in F. By the case $i = 0$ of Lemma 4-3 we have that $F \models M$, and so $F \models \neg C$. However, since C is a clause in F it follows that F is unsatisfiable. For the right-to-left implication of property 1, if $\emptyset \parallel F \not\Longrightarrow^{!} fail$, then by Theorem 5 there must be a state $M \parallel F$ such that $\emptyset \parallel F \Longrightarrow^{!} M \parallel F$. Then F is satisfiable by Lemma 7.

For property 2, if $\emptyset \parallel F \Longrightarrow^{!} M \parallel F$ then F is satisfiable by Lemma 7. Conversely, if $\emptyset \parallel F \not\Longrightarrow^{!} M \parallel F$, then by Theorem 5 again, $\emptyset \parallel F \Longrightarrow^{!} fail$ and hence F is unsatisfiable by property 1. Property 3 is again Lemma 7. □

The previous theorem does not just prove the desirable properties for a concrete DPLL procedure; rather, it proves the correctness of *any* procedure applying these steps, with *any* strategy. For example, the designer of a practical

DPLL implementation is free to choose her own heuristic for selecting the next decision literal in Decide, or choose the priorities between the different rules.

Note that we may have $\emptyset \parallel F \Longrightarrow^{!} M \parallel F$ and also $\emptyset \parallel F \Longrightarrow^{!} M' \parallel F$, for different M and M'.[1] Then, the formula F is satisfiable and both M and M' are models of F.

2.3 DPLL with Clause Learning

Definition 10. *The* DPLL system with learning *consists of the four transition rules of the Basic DPLL system, plus the following two additional rules:*

$$\begin{array}{l} \textit{Learn}: \\ \quad M \parallel F \implies M \parallel F, C \ \textbf{if} \ \begin{cases} \textit{all atoms of } C \textit{ occur in } F \\ F \models C \end{cases} \\ \textit{Forget}: \\ \quad M \parallel F, C \implies M \parallel F \quad \textbf{if} \ \{ F \models C \end{array}$$

In these two rules, the clause C is said to be learned *and* forgotten*, respectively. In the following, we denote by $\Longrightarrow_L$ the transition relation defined by the DPLL system with learning.*

Example 11. (Example 3 continued). When applying Backjump, many actual DPLL implementations learn the backjump clause:

$$\begin{array}{rcl} \ldots \quad \ldots & & \\ \mathbf{1}\,2\,\mathbf{3}\,4 \parallel \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_L & \text{(Decide)} \\ \mathbf{1}\,2\,\mathbf{3}\,4\,\mathbf{5} \parallel \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_L & \text{(UnitPropagate)} \\ \mathbf{1}\,2\,\mathbf{3}\,4\,\mathbf{5}\,\overline{6} \parallel \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_L & \text{(Backjump)} \\ \mathbf{1}\,2\,\overline{5} \parallel \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_L & \text{(Learn)} \\ \mathbf{1}\,2\,\overline{5} \parallel \overline{1}\vee 2,\ \overline{3}\vee 4,\ \overline{5}\vee\overline{6},\ 6\vee\overline{5}\vee\overline{2},\ \overline{1}\vee\overline{5} & & \end{array}$$

When backjumping to decision level j, the backjump clause $C\vee l'$ (in the example $\overline{1}\vee\overline{5}$) is always such that, if it had existed the last time the procedure was at level j, the literal l' could have been added by UnitPropagate. Learning such clauses hence avoids repeated work by preventing decisions such as **5**, if, after more backjumping, one reaches again a state similar to this decision level j (where "similar" roughly means that it could produce the same conflict). Indeed, reaching such similar states frequently happens in industrial problems having some regular structure. The use of Forget is to free memory by removing a clause C, once a search region presenting such similar states has been abandoned. In practice this is usually done if the activity of C (i.e., the number of times C causes some conflict or some unit propagation) has become low [MMZ+01]. □

The results given in the previous subsection for Basic DPLL smoothly extend to DPLL with learning, and again the starting point is the following.

[1] *Confluence*, in the sense of, e.g., rewrite systems is not needed here.

Lemma 12. *If $\emptyset \parallel F \Longrightarrow_L^* M \parallel F'$ then the following hold.*

1. *All the atoms in M and all the atoms in F' are atoms of F.*
2. *M contains no literal more than once and is indeed an assignment, i.e., it contains no pair of literals of the form p and $\neg p$.*
3. *F' is logically equivalent to F.*
4. *If M is of the form $M_0\ l_1\ M_1\ \ldots\ l_n\ M_n$, where $l_1, \ldots, l_n$ are all the decision literals of M, then $F \cup \{l_1, \ldots, l_i\} \models M_i$ for all $i = 0 \ldots n$.*

Proof. It is easy to see that property 3 holds. Using this fact, the other properties can be proven similarly to the proof of Lemma 4.

Theorem 13 (Termination of $\Longrightarrow_L$). *There exist no infinite sequences of the form $\emptyset \parallel F \Longrightarrow_L S_1 \Longrightarrow_L \ldots$ if no clause C is learned infinitely many times along a sequence.*

Proof. The ordering used in Theorem 5 can also be applied here, since, by Lemma 12, atoms appearing in any state are atoms of F. Therefore an infinite sequence of the form $\emptyset \parallel F \Longrightarrow_L S_1 \Longrightarrow_L \ldots$ cannot contain any infinite subsequence of contiguous $\Longrightarrow$ steps, and must hence contain infinitely many Learn or Forget steps, which is not possible since there are only finitely many different clauses with atoms in F, and no clause C is learned infinitely many times along the sequence. □

Note that the condition that no clause C is learned infinitely many times is in fact a necessary and sufficient condition for termination. This condition is easily enforced by applying at least one rule of the Basic DPLL system between two successive applications of Learn. Since states do not increase with respect to the ordering used in Theorem 5 when Learn is applied, any strict alternation between Learn and Basic DPLL rules must be finite as well. As with the basic DPLL system, we have the following definition and theorem (with identical proof).

Definition 14. *A* DPLL procedure with learning *is any procedure taking an input cnf F and computing a sequence $\emptyset \parallel F \Longrightarrow_L^* S$ where S is a final state with respect to the Basic DPLL system.*

Theorem 15. *The DPLL system with learning provides a decision procedure for the satisfiability of cnf formulas F, that is:*

1. *$\emptyset \parallel F \Longrightarrow_L^! \mathit{fail}$ if, and only if, F is unsatisfiable.*
2. *$\emptyset \parallel F \Longrightarrow_L^* M \parallel F'$, where $M \parallel F'$ is a final state with respect to the Basic DPLL system, if, and only if, F is satisfiable.*
3. *If $\emptyset \parallel F \Longrightarrow_L^* M \parallel F'$, where $M \parallel F'$ is a final state with respect to the Basic DPLL system, then M is a model of F.*

3 Abstract DPLL modulo theories

This section deals with procedures for Satisfiability Modulo Theories (SMT), that is, procedures for deciding the satisfiability of ground[2] cnf formulas in the context of a background theory T. Typical theories considered in this context are EUF (equality with uninterpreted function symbols), linear arithmetic (over the integers and over the reals), some theories of arrays and of other data structures such as lists, finite sets, and so on. For each of these theories there exist efficient procedures (in practice) that decide the satisfiability, in the theory, of *conjunctions* of ground literals. To decide efficiently the satisfiability of ground *cnf* formulas, many people have recently worked on combining these decision procedures with DPLL based SAT engines. In this section we show that many of the existing combinations can be described and discussed within the Abstract DPLL framework.

In the rest of the paper we consider first-order logic without equality—of which the purely propositional case we have seen until now is a particular instance. We adopt the standard notions of first-order structure, satisfaction, entailment, etc., extended with the following. A *theory* is a satisfiable set of closed first-order formulas. A formula F is *(un)satisfiable in* a theory T, or T-*(in)consistent*, if there is a (no) model of T that satisfies F, that is, if $T \cup F$ is (un)satisfiable. If F and G are formulas, F *entails* G *in* T, written $F \models_T G$, if $T \models \neg F \vee G$. If $F \models_T G$ and $G \models_T F$, we say that F and G are T-*equivalent*. We extend the notion of (partial truth) assignment M from Section 2 to a set of ground first-order literals in the obvious way. We say that M is a T-*model* of a ground formula F if M, seen as the conjuction of its literals, is T-consistent and $M \models_T F$.

In the following we will use T to denote a background theory T such that the satisfiability in T of conjunctions of ground literals is decidable. To decide the satisfiability of ground cnf formulas we consider again the DPLL systems introduced in the previous section—with arbitrary ground atoms now used in place of propositional symbols—and add new rules for dealing with T. However, in the side conditions of the rules presented in the previous section, entailment between formulas is now replaced by entailment in T between formulas. That is, the condition $F \models C$ in Learn and Forget is now $F \models_T C$, the Backjump rule is

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

and Decide, Fail and UnitPropagate remain unchanged. We point out that the rules of the previous section can now be seen as a particular instance of the new ones if we consider T to be the empty theory.

[2] By *ground* we mean containing no variables—although possibly containing constants not in T.

3.1 A simple example: the classical very lazy approach

One way for dealing with SMT is what has been called the *lazy* approach [dMR02,ABC+02,BDS02,FJOS03]. This approach initially considers each atom occurring in a formula F to be checked for satisfiability simply as a propositional symbol, and sends the formula to a SAT solver. If the SAT solver returns a propositional model of F that is T-inconsistent, a ground clause, a *lemma*, precluding that model is added to F and the SAT solver is started again. This process is repeated until the SAT solver finds a T-consistent model or returns unsatisfiable. The main advantage of such a lazy approach is its flexibility, since it can easily combine any SAT solver with any decision procedure for conjunctions of theory literals, as long as the decision procedure is able to generate such lemmas.

The addition of these lemmas can be modelled by the following rule, which we will call Very Lazy Theory Learning:

$$M\ l\ M_1 \parallel F \implies \emptyset \parallel F,\ \neg l_1 \vee \ldots \vee \neg l_n \vee \neg l \ \textbf{if} \begin{cases} M\ l\ M_1 \models F \\ \{l_1, \ldots, l_n\} \subseteq M \\ l_1 \wedge \ldots \wedge l_n \models_T \neg l \end{cases}$$

Combining this rule with the four Basic DPLL rules, or with the six rules of DPLL with learning, the resulting Very Lazy DPLL system terminates if no clause is learned infinitely many times, since only finitely many such new clauses (built over input literals) exist. For this condition to be fulfilled, applying at least one rule of the Basic DPLL system between any two Learn applications does not suffice. It suffices if, in addition, no clause generated with Very Lazy Theory Learning is ever forgotten. The system is also easily proved correct as it is done in the following subsection, by observing that M, seen as the conjunction of its literals, is T-consistent for every state $M \parallel F$ that is final with respect to Basic DPLL and Very Lazy Theory Learning. However, in what follows we will focus on other more interesting—and in practice better—lazy techniques, based on tighter integrations between DPLL and theory solvers.

3.2 Less lazy approaches

It is clear that, as soon as a DPLL procedure reaches a state $M \parallel F$ with a (possibly non-total) T-inconsistent M, the corresponding lemma can already be added. Furthermore, it is also not necessary to restart from scratch once the lemma has been added. These ideas can be modelled by the following rule.

Definition 16. *The Lazy Theory Learning rule is the following:*

$$M\, l\, M_1 \parallel F \implies M\, l\, M_1 \parallel F,\ \neg l_1 \vee \ldots \vee \neg l_n \vee \neg l \ \textbf{if} \begin{cases} \{l_1, \ldots, l_n\} \subseteq M \\ l_1 \wedge \ldots \wedge l_n \models_T \neg l \\ \neg l_1 \vee \ldots \vee \neg l_n \vee \neg l \notin F \end{cases}$$

The Lazy Theory DPLL *system consists of this rule and the six rules of DPLL with learning. In the following, we denote by* $\Longrightarrow_{LT}$ *the transition relation defined by the Lazy Theory DPLL system.*

Note that the lemma $\neg l_1 \vee \ldots \vee \neg l_n \vee \neg l$ added by an application of the Lazy Theory Learning rule is, by construction, always false in $M\,l$, making either Fail or Backjump applicable to the resulting state. In practice, one of these two rules is always applied immediately after Lazy Theory Learning. This makes the third test in the rule—introduced here to ensure termination—unnecessary.

This DPLL system is still called lazy because it does not consider any theory information until a T-inconsistent partial interpretation $M\,l$ has been reached. As we will see, this is the essential difference between these lazy approaches and the DPLL(T) approach that is described in Subsection 3.3 below.

All the results below are proved as in the previous section. However, the following key lemma is needed to show that for any state of the form $M \parallel F$ that is final with respect to Basic DPLL and Lazy Theory Learning, M is T-consistent and $M \models_T F$.

Lemma 17. *Let $\emptyset \parallel F_0 \Longrightarrow_{LT}^{*} M \parallel F$. If M is T-inconsistent then the rule Lazy Theory Learning applies to $M \parallel F$.*

Theorem 18 (Termination of $\Longrightarrow_{LT}$). *There exists no infinite sequence of the form $\emptyset \parallel F \Longrightarrow_{LT} S_1 \Longrightarrow_{LT} \ldots$ if no clause C is learned by Learn or Lazy Theory Learning infinitely many times along a sequence.*

Definition 19. *A* Lazy Theory DPLL procedure *for T is any procedure taking an input cnf F and computing a sequence $\emptyset \parallel F \Longrightarrow_{LT}^{*} S$ where S is a final state with respect to the Basic DPLL system and Lazy Theory Learning.*

Theorem 20. *The Lazy Theory DPLL system provides a decision procedure for the satisfiability in T of cnf formulas F, that is:*

1. *$\emptyset \parallel F \Longrightarrow_{LT}^{!}$ fail if, and only if, F is unsatisfiable in T.*
2. *$\emptyset \parallel F \Longrightarrow_{LT}^{*} M \parallel F'$, where $M \parallel F'$ is a final state wrt the Basic DPLL system and Lazy Theory Learning, if, and only if, F is satisfiable in T.*
3. *If $\emptyset \parallel F \Longrightarrow_{LT}^{*} M \parallel F'$, where $M \parallel F'$ is a final state wrt the Basic DPLL system and Lazy Theory Learning, then M is a T-model of F.*

Systems such as CVC Lite [BB04] are concrete implementations of Lazy Theory DPLL. Usually, in such implementations the Lazy Theory Learning rule is applied eagerly, that is, with an empty M_1, as soon as the current partial interpretation becomes T-inconsistent. Therefore, the soundness and completeness of the approach followed by CVC Lite is a particular instance of the previous theorem.

3.3 The DPLL(T) approach with eager theory propagation

The Lazy Theory DPLL systems we have seen are lazy in the sense that they use theory information only after a theory-inconsistent partial assignment has been generated. In this subsection we describe the DPLL(T) approach [GHN+04] with

eager theory propagation, which allows the use of theory information as soon as possible. This new information reduces the search space by discovering the truth value of literals otherwise considered to be unassigned. Moreover, it does this without sacrificing modularity or flexibility: combining arbitrary theory decision procedures for conjunctions of literals with a DPLL system is as simple as for the lazy approaches such as that of CVC Lite. The key idea behind DPLL(T) is the following rule:

Definition 21. *The* Theory Propagate *rule is the following:*

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

The DPLL(T) system with eager theory propagation *consists of this rule and the six rules of DPLL with learning. We denote by* $\Longrightarrow_{Edpll(T)}$ *the transition relation defined by the DPLL(T) system with eager theory propagation where* Theory Propagate *has priority over all the other rules.*

All results as in the previous sections apply here, including termination under the usual assumption (since Theory Propagate also decreases with respect to the ordering $\succ$ used in Theorem 5). The only additional ingredient needed is the following lemma.

Lemma 22. *If* $\emptyset \parallel F_0 \Longrightarrow^*_{Edpll(T)} M \parallel F$ *then* M *is* T*-consistent.*

Proof. This property is true initially, and all rules preserve it, by the fact that $M \models_T l$ if, and only if, $M \cup \{\neg l\}$ is T-inconsistent: the rules only add literals to M that are undefined in M, and Theory Propagate adds all literals l of F that are theory consequences of M, before any literal $\neg l$ making it T-inconsistent can be added to M by any of the other rules. □

Definition 23. *A* DPLL(T) procedure with Eager Theory Propagation *for* T *is any procedure taking an input cnf* F *and computing a sequence* $\emptyset \parallel F \Longrightarrow^*_{Edpll(T)} S$ *where* S *is a final state wrt* Theory Propagate *and the Basic DPLL system.*

Theorem 24. *The DPLL system with eager theory propagation provides a decision procedure for the satisfiability in* T *of cnf formulas* F*, that is:*

1. $\emptyset \parallel F \Longrightarrow^!_{Edpll(T)}$ *fail if, and only if,* F *is unsatisfiable in* T*.*
2. $\emptyset \parallel F \Longrightarrow^*_{Edpll(T)} M \parallel F'$*, where* $M \parallel F'$ *is a final state wrt the Basic DPLL system and* Theory Propagate*, if, and only if,* F *is satisfiable in* T*.*
3. *If* $\emptyset \parallel F \Longrightarrow^*_{Edpll(T)} M \parallel F'$*, where* $M \parallel F'$ *is a final state wrt the Basic DPLL system and* Theory Propagate*, then* M *is a* T*-model of* F*.*

In practice, the DPLL(T) approach can be implemented, very much in the spirit of the CLP(X) scheme in constraint logic programming, by building a component DPLL(X) common to all theories, and instantiating it with solvers for different theories T to obtain different DPLL(T) procedures. At each state $M \parallel F$, the theory solver only sees the part M and communicates to the DPLL(X) engine any input literals entailed by M in the given theory. More details on an architecture for concrete DPLL(T) systems can be found in [GHN+04].

3.4 The DPLL(T) approach with non-exhaustive propagation

For some theories eager Theory Propagate is expensive in an actual implementation. For example, in our experience with EUF, this is the case for detecting input literals entailed by disequations. However, using the information coming from the "cheap enough" applications of Theory Propagate is extremely useful for pruning the search space. Therefore one would like to have a combination of Theory Propagate, for the cheaper cases, and Lazy Theory Learning, for covering the incompletenesses of Theory Propagate making the equivalent of Lemma 22 hold. This is actually what is done in the DPLL(T) implementation of [GHN+04].

Definition 25. *The* DPLL(T) system with non-exhaustive theory propagation *consists of the Lazy Theory Learning and Theory Propagate rules and the six rules of DPLL with learning. We denote by* $\Longrightarrow_{NEdpll(T)}$ *the transition relation defined by the DPLL(T) system with eager theory propagation.*

Definition 26. *A* DPLL(T) procedure with Non-Exhaustive Theory Propagation *for* T *is any procedure taking an input cnf* F *and computing a sequence* $\emptyset \parallel F \Longrightarrow^{*}_{NEdpll(T)} S$ *where* S *is a final state with respect to the Basic DPLL system and Lazy Theory Learning.*

A necessary and sufficient condition for ensuring the termination of the previous system is again that no clause can be learned by Lazy Theory Learning or Learn infinitely many times. In practice, this can be achieved by the same strategy presented in Subsection 3.2. Hence, we have:

Theorem 27. *The DPLL system with non-exhaustive theory propagation provides a decision procedure for the satisfiability in* T *of cnf formulas* F*, that is:*

1. $\emptyset \parallel F \Longrightarrow^{!}_{NEdpll(T)}$ *fail if, and only if,* F *is unsatisfiable in* T*.*
2. $\emptyset \parallel F \Longrightarrow^{*}_{NEdpll(T)} M \parallel F'$*, where* $M \parallel F'$ *is a final state wrt Basic DPLL and Lazy Theory Learning, if, and only if,* F *is satisfiable in* T*.*
3. *If* $\emptyset \parallel F \Longrightarrow^{*}_{NEdpll(T)} M \parallel F'$*, where* $M \parallel F'$ *is a final state wrt Basic DPLL and Lazy Theory Learning, then* M *is a* T*-model of* F*.*

4 Conclusions

We have presented a declarative formal framework for modeling DPLL-based solvers for propositional satisfiability or for satisfiability modulo theories. We have shown that the essence of these solvers can be described simply and abstractly in terms of rule-based transition systems over states consisting of a truth assignment and a clause set.

The declarative and formal nature of our transition systems makes it easier to prove properties such as soundness, completeness or termination of DPLL-style algorithms. Furthermore, it facilitates their comparison as their differences can

be more easily seen as differences in the set of their transition rules or in their rule application strategy.

The approach we presented is as flexible and declarative as the one followed in [Tin02], which first formulated basic DPLL and DPLL modulo theories abstractly, as sequent-style calculi. But it considerably improves on that work because it allows one to model more features of modern DPLL-based engines directly within the framework. This contrasts with the calculi in [Tin02] where features as backjumping and learning can be discussed only at the control level, in terms of proof procedures for the calculi.

References

[ABC+02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *CADE-18*, LNCS 2392, pages 195–210, 2002.

[ACG00] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. SAT-based procedures for temporal reasoning. In *Procs. 5th European Conference on Planning*, LNCS 1809, pages 97–108, 2000.

[BB04] Clark W. Barrett and Sergey Berezin. CVC lite: A new implementation of the cooperating validity checker. Category B. In *Procs. 16th Int. Conf. Computer Aided Verification (CAV)*, LNCS 3114, pages 515–518, 2004.

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

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

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

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

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

[GHN+04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Procs. 16th Int. Conf. Computer Aided Verification (CAV)*, LNCS 3114, pages 175–188, 2004.

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

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

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

[Tin02] Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Procs. 8th European Conf. on Logics in Artificial Intelligence*, LNAI 2424, pages 308–319, 2002.

[Zha97] Hantao Zhang. SATO: An efficient propositional prover. In *CADE-14*, LNCS 1249, pages 272–275, 1997.