A DPLL-based Calculus for Ground Satisfiability Modulo Theories

Cesare Tinelli

Department of Computer Science, University of Iowa, USA
tinelli@cs.uiowa.edu

Abstract. We describe and discuss DPLL($\mathcal{T}$), a parametric calculus for proving the satisfiability of ground formulas in a logical theory $\mathcal{T}$. The calculus tightly integrates a decision procedure for the satisfiability in $\mathcal{T}$ of sets of literals into a sequent calculus based on the well-known method by Davis, Putman, Logemann and Loveland for proving the satisfiability of propositional formulas. For being based on the DPLL method, DPLL($\mathcal{T}$) can incorporate a number of very effective search heuristics developed by the SAT community for that method. Hence, it can be used as the formal basis for novel and efficient implementations of satisfiability checkers for theories with decidable ground consequences.

1 Introduction

Proving the satisfiability of ground formulas in a given first-order theory is an important research problem with applications in many areas of computer science and artificial intelligence, such as software/hardware verification, compiler optimization, constraint-based planning, scheduling, and so on. Since this problem is decidable for many theories of interest, a lot of research effort has gone into trying to devise efficient decision procedures for such theories. Typically, the effort has concentrated on the simpler problem of devising procedures that decide the satisfiability in a theory $\mathcal{T}$ of just conjunctions of ground literals over some signature Σ. From a theoretical point of view, this is enough to decide the satisfiability in $\mathcal{T}$ of arbitrary ground formulas over Σ. One simply needs to convert the formula into disjunctive normal form and then invoke the decision procedure on each disjunct until a satisfiable disjunct is found. In practice, however, this approach is extremely inefficient because of the exponential explosion caused by the conversion into DNF. Existing checkers for ground satisfiability modulo theories (e.g. SVC [3], STeP [5], Simplify [14]) have relied instead on alternative ways to deal with the Boolean structure of a formula. In most cases, the approach followed was modeled more or less closely on the what is often called the DPLL method, a method collectively due to Davis, Putnam, Logemann and Loveland [7, 6].

In the last years, there has been much renewed interest in the DPLL method in the propositional satisfiability (SAT) community. The method is now the basis for the great majority of current state-of-the art SAT solvers. Several improvements and variations on it have been developed which have lead to spectacular

increases in the performance of SAT solvers. Such successes have pushed several researchers interested in satisfiability modulo theories to find ways to harness the power of modern DPLL-based solvers by coupling them with decision procedures. Possibly the first work along these lines is that described in [1]. Very recent, independent research on the same idea is reported in [2, 4, 8]. Very briefly, the idea common to all these works is the following: given a set of ground clauses to be checked for satisfiability in some theory $\mathcal{T}$, use an off-the-shelf solver to obtain a propositional model of the set; then pass the model (as a set of literals) to a decision procedure for $\mathcal{T}$ and check its consistency with $\mathcal{T}$ before succeeding. Oversimplifying a bit, the major difference among these works is the degree of "laziness" with which the decision procedure is invoked. Some invoke the procedure after a complete propositional model has been found, others invoke it incrementally, as the model is being built, in order to minimize backtracking on wrong choices.

The approaches in [1, 2, 4, 8] are all described procedurally. In this paper, we propose instead a general declarative framework, given as a sequent calculus, for extending the DPLL method with decision procedures. While the calculus is general enough that it can model (with minor changes) each of the approaches above, it also allows a much tighter integration of decision procedures into the DPLL method. This sort of integration is analogous to that achieved in constraint logic programming (CLP) [11] between SLD-resolution and constraint solving. As in CLP, in our calculus the decision procedure for the given theory can be used to *drive* the search toward a solution, as opposed to validate a (partial) solution after it has been found. In principle, this leads to a more efficient search than with the other approaches, while still benefitting from the various optimizations developed for the DPLL method.

We say "in principle" for two reasons. The first is of course that there always a risk that the speed-up obtained by a more focused search is actually offset by the cost of frequently calling the decision procedure. Only experimental work and fine tuning can make sure that that is not the case. The second reason is that, contrary to what seems to be a common belief, the optimization strategies used in DPLL-based SAT solvers do not immediately lift to satisfiability modulo theories.[1] Our current research is aimed at establishing which heuristics do lift and how, and how effective they are in practice. For that we find it more useful to work with a declarative description of the DPLL method which separates control and optimization issues (including the calls to the decision procedure) from the essence of the method. The control aspects of the DPLL method and of modern DPLL-based systems can be conveniently modeled as search strategies for our calculus, instead of being more of less hidden in the details of the various implementations. This paper provides an initianl, incomplete account of our theoretical work in this direction.

To simplify the exposition, we start in Section 2 with a description of the basic DPLL method in terms of a simple sequent calculus. Given the simplicity

[1] This is a concern not only for our approach but also for the ones mentioned above, even more so given that they use existing SAT solvers more or less as they are.

of the original method, this results in a clean calculus that is easy to reason about and extend. Then, in Section 3 we show how to extend the calculus with decision procedures to obtain a sound, complete and terminating calculus for deciding the satisfiability of ground formulas in certain logical theories. Finally, in Section 4 we discuss the lifting to the extended calculus of some of the general optimization strategies developed for the DPLL method. For space constraints, we must omit the proofs of the results given here. All proofs, together with a more detailed discussion of the issues and the results presented in this paper and some initial experimental results, can be found in [15].

1.1 Formal Preliminaries

We assume that the reader is familiar with basic theorem proving concepts and terminology. Some specific notions and notation we use are defined below.

We will consider propositional logic as a special case of first-order logic, one in which all atomic formulas consist of predicates of zero arity. We call an atomic formula—whether propositional or first-order—an *atom*. A *(ground) literal* is an atom or a negated atom (with no variables). We denote the complement of a literal l by $\overline{l}$. A *ground clause* (henceforth, a clause, as we consider *only* ground clauses here) is a disjunction of zero or more ground literals. We denote by $l \vee C$ a clause D such that l is a literal of D and C is the (possibly empty) clause obtained by removing one occurrence of l from D. If Φ is a clause set, $Ats(\Phi)$ is the set of all atoms occurring in the clauses of Φ.

A sentence is a closed first-order formula. Let Φ be a set of sentences. A first-order structure $\mathcal{A}$ *satisfies* Φ or *is a model of* Φ if every sentence of Φ is true in $\mathcal{A}$; otherwise, $\mathcal{A}$ *falsifies* Φ. The set Φ is *satisfiable* if it has a model, and is *unsatisfiable* otherwise. A *theory* is a satisfiable set of sentences. The set Φ is *(un)satisfiable in* a theory $\mathcal{T}$ if there is a (no) model of $\mathcal{T}$ that satisfies Φ; equivalently, if $\mathcal{T} \cup \Phi$ is (un)satisfiable. If ψ is a sentence, Φ *entails* ψ *in* $\mathcal{T}$, written $\Phi \models_{\mathcal{T}} \psi$, if every model of $\mathcal{T}$ that satisfies Φ satisfies ψ as well.

2 A Sequent Calculus for the DPLL Method

The DPLL method can be used to decide the satisfiability of propositional formulas in conjunctive normal form, or, more precisely but equivalently, the satisfiability of finite sets of propositional clauses. The three essential operations of the DPLL method are unit resolution with backward subsumption, unit subsumption, and recursive reduction to smaller problems. The method can be roughly described as follows.[2]

Given an input clause set Φ, apply *unit propagation* (aka *Boolean constraint propagation*) to it, that is, close Φ under unit resolution with backward subsumption, and eliminate in the process (a) all non-unit clauses subsumed by a unit clause in the set and (b) all unit clauses whose (only) atom occurs only once in

[2] See the original papers [7, 6], among others, for a more complete description.

$$\textbf{(subsume)}\quad \frac{\Lambda \vdash \Phi,\ l \vee C}{\Lambda \vdash \Phi} \text{ if } l \in \Lambda \qquad \textbf{(resolve)}\quad \frac{\Lambda \vdash \Phi,\ l \vee C}{\Lambda \vdash \Phi,\ C} \text{ if } \bar{l} \in \Lambda$$

$$\textbf{(assert)}\quad \frac{\Lambda \ \vdash \Phi,\ l}{\Lambda, l \vdash \Phi,\ l} \text{ if } l \notin \Lambda \text{ and } \bar{l} \notin \Lambda \qquad \textbf{(empty)}\quad \frac{\Lambda \vdash \Phi,\ \Box}{\Lambda \vdash \Box} \text{ if } \Phi \neq \emptyset$$

$$\textbf{(split)}\quad \frac{\Lambda \ \vdash\ \Phi}{\Lambda, p \ \vdash\ \Phi \qquad \Lambda, \neg p \ \vdash\ \Phi} \text{ if } p \in \mathit{Ats}(\Phi),\ p \notin \Lambda \text{ and } \neg p \notin \Lambda$$

Fig. 1. The rules of the DPLL calculus

the set. If the closure Φ^* of Φ contains the empty clause, then fail. If Φ^* is the empty set, then succeed. Otherwise, choose an arbitrary literal l from Φ^* and check recursively, and separately, the satisfiability of $\Phi^* \cup \{l\}$ and of $\Phi^* \cup \{\bar{l}\}$, succeeding if and only if one of the two subsets is satisfiable.

The essence of this method can be captured by a sequent calculus, whose rules are described in Figure 1. The calculus manipulates sequents of the form $\Lambda \vdash \Phi$, where Λ, the *context* of the sequent, is a finite multiset of ground literals and Φ is a finite multiset of ground clauses.[3] The intended use of the calculus is to derive a sequent of the form $\Lambda \ \vdash\ \emptyset$ from an initial sequent $\emptyset \ \vdash\ \Phi_0$, where Φ_0 is a clause set to be checked for satisfiability. If that is possible, then Φ_0 is satisfiable; otherwise, Φ_0 is unsatisfiable. Informally, the purpose of the context Λ is to store incrementally a set of *asserted literals*, i.e., a set of literals in Φ_0 that must or can be true for Φ_0 to be satisfiable. When $\Lambda \vdash \emptyset$ is derivable from $\emptyset \ \vdash\ \Phi_0$, the context Λ is indeed a witness of Φ_0's satisfiability as it describes a (Herbrand) model of Φ_0: one that satisfies an atom p in Φ_0 iff p occurs positively in Λ.

The context is grown by the **assert** and the **split** rules. The **assert** rule models the fact that every literal occurring as a unit clause in the the current clause set must be satisfied for the whole clause set to be satisfied. The **split** rule corresponds to the decomposition in smaller subproblems of the DPLL method. This rule is the only *don't-know* non-deterministic rule of the calculus. Its intended use is to guess the truth value of an *undetermined* atom p in the clause set of the current sequent $\Lambda \ \vdash\ \Phi$, where by undetermined we mean not already asserted either positively or negatively in the context Λ. The guess allows the continuation of the derivation with either the sequent $\Lambda,\ p \ \vdash\ \Phi$ or with the sequent $\Lambda, \neg p \vdash \Phi$. The other two main operations of the DPLL method, unit resolution with backward subsumption and unit subsumption, are modeled respectively by the **resolve** and the **subsume** rule. The **resolve** rule removes from a clause all literals whose complement has been asserted (which corresponds to generating the simplified clause by unit resolution and then discarding the old clause by backward subsumption). The **subsume** rule removes all clauses that contain an asserted literal (because all of these clauses will be satisfied in any model in

[3] As customary, we write $\Lambda, l \ \vdash\ \Phi, C$, say, to denote the sequent $\Lambda \cup \{l\} \ \vdash\ \Phi \cup \{C\}$.

which the asserted literal is true). The **empty** rule is in the calculus just for convenience and could be removed with no loss of completeness. It models the fact that a derivation can be terminated as soon as the empty clause ($\Box$) is derived. Note that the **assert** rule as well could be removed without loss of completeness since it is really an optimization of the **split** rule. This optimization is crucial for practical purposes, if not theoretical ones. It is well known that in the DPLL method, unit propagation—achieved in our calculus through the combined use of **assert**, **subsume** and **resolve**—is possibly the single most important factor in the speed of DPLL-based systems [9, 13].

As it is the DPLL calculus is not strong enough for producing practical SAT solvers because, if implemented naively, it basically enumerates all possible Herbrand interpretations of the initial set of clauses. In retrospect, all implementations of the DPLL method can be seen as procedures for exploring systematically, but efficiently, the search space generated by this calculus. Modern implementations follow a number of optimization strategies in the way they perform unit propagation and interleave it with the guessing of undetermined literals. These strategies can be modeled at a more abstract level as derivation strategies for the DPLL calculus. The optimizations found in modern DPLL-based systems go well beyond the choice of the next derivation rule to apply. But these optimizations too can be modeled as (additional) search strategies for the calculus. For instance, a heuristics that chooses the next atom on which to split the search can be modeled in the calculus by adding a selection function to the **split** rule. Similarly, the way and the extent to which unit resolution is applied to any one clause can be modeled with the addition of a proper clause/literal selection function to the **resolve** rule. Finally, modern systems also have sophisticated ways of pruning the search space by intelligent backjumping (to previous decision points created by the **split** rule), and by learning *lemmas* from failed derivations (see later). All these techniques can be recast as search heuristics for the DPLL calculus. Perhaps more importantly, especially for the extension of the calculus described later, their correctness can be proved formally using established proof techniques from the automated deduction research.

In the following, we will give a sense of how various optimizations can be described in terms of the calculus. We will do that, however, not for DPLL itself but for a strictly more powerful extension of it that encapsulates decision procedures for certain first-order theories.

3 The DPLL($\mathcal{T}$) Calculus

In this section we describe the DPLL($\mathcal{T}$) calculus, an extension of the calculus in the previous section obtained by replacing propositional satisfiability with ground satisfiability with respect to a first-order *background theory* $\mathcal{T}$. The calculus, which is parametric in the background theory $\mathcal{T}$, can be used to verify the satisfiability in $\mathcal{T}$ of ground CNF formulas.

The extended calculus works again with ground sequents of the form $\Lambda \vdash \Phi$, where the literals in Λ and the clauses in Φ are now over some fixed signature Σ.

$$\textbf{(subsume)}\ \frac{\Lambda \vdash \Phi,\ l \vee C}{\Lambda \vdash \Phi}\ \text{if}\ \ \Lambda \models_{\mathcal{T}} l \qquad \textbf{(resolve)}\ \frac{\Lambda \vdash \Phi,\ l \vee C}{\Lambda \vdash \Phi,\ C}\ \text{if}\ \ \Lambda \models_{\mathcal{T}} \overline{l}$$

$$\textbf{(assert)}\ \frac{\Lambda\ \ \vdash \Phi,\ l}{\Lambda, l \vdash \Phi,\ l}\ \text{if}\ \ \begin{array}{l}\Lambda \not\models_{\mathcal{T}} l \text{ and} \\ \Lambda \not\models_{\mathcal{T}} \overline{l}\end{array} \qquad \textbf{(empty)}\ \frac{\Lambda \vdash \Phi,\ \Box}{\Lambda \vdash \Box}\ \text{if}\ \ \Phi \neq \emptyset$$

$$\textbf{(split)}\ \frac{\Lambda\ \vdash\ \Phi}{\Lambda, p\ \vdash\ \Phi \qquad \Lambda, \neg p\ \vdash\ \Phi}\ \text{if}\ \ p \in \mathit{Ats}(\Phi) \text{ and } \Lambda \not\models_{\mathcal{T}} p \text{ and } \Lambda \not\models_{\mathcal{T}} \neg p$$

Fig. 2. The rules of the DPLL($\mathcal{T}$) calculus

Its rules, given in Figure 2, are *exactly* the same as in the DPLL calculus. The only difference lies in their side conditions. Whereas in DPLL these conditions involve membership of certain literals in the context Λ of the rule's premise, in DPLL($\mathcal{T}$) they involve entailment by Λ in $\mathcal{T}$. We point out that, in general, the tests in the side conditions of DPLL($\mathcal{T}$)'s rules may not be computable for a given theory $\mathcal{T}$ and signature Σ, which means that DPLL($\mathcal{T}$) does not always yield a decision procedure. For decidability purposes it is necessary, and sufficient, to assume that the satisfiability in $\mathcal{T}$ of finite sets of ground Σ-literals is decidable.

The DPLL($\mathcal{T}$) calculus includes the DPLL calculus as one of its instances: the one in which $\mathcal{T}$ is empty and Σ is a set of propositional variables. To see that, it is enough to notice that in that instance the side conditions of the corresponding rules of DPLL and DPLL($\mathcal{T}$) become equivalent.

For the rest of the paper, $\mathcal{T}$ will be a fixed background theory in which the satisfiability of finite sets of ground Σ-literals is decidable. We will implicitly assume the availability of a corresponding decision procedure for $\mathcal{T}$. We will say that a sequent $\Lambda\ \vdash\ \Phi$ is *satisfiable in* $\mathcal{T}$ iff the set $\Lambda \cup \Phi$ is satisfiable in $\mathcal{T}$.

We describe the salient properties of the DPLL($\mathcal{T}$) calculus in the following. For that, we need an appropriate notion of derivation and proof. As in other sequent calculi, derivations in DPLL($\mathcal{T}$) involve the construction of derivation trees.

Definition 1. *A* derivation tree (in DPLL($\mathcal{T}$)) *is a labeled tree each of whose nodes is labeled by a sequent and such that, for each non-leaf node N, the sequents labeling its successor(s) node(s) can be obtained by applying a rule of* DPLL($\mathcal{T}$) *to the sequent labeling N.*

Definition 2. *A branch in a derivation tree is (a)* successful *if its leaf is labeled by a sequent of the form $\Lambda\ \vdash\ \emptyset$, (b)* failed *if its leaf is labeled by a sequent of the form $\Lambda \vdash \Box$, and (c)* incomplete *otherwise.*

We say that a derivation tree is a *derivation tree of* $\Lambda\ \vdash\ \Phi$ iff its root is labeled by $\Lambda\ \vdash\ \Phi$; we say that it is a *proof tree of* $\Lambda\ \vdash\ \Phi$ iff the tree has either a successful branch or only failed branches. In the latter case, we call it a *refutation tree*. Derivation trees in DPLL($\mathcal{T}$) satisfy the following invariant.

Lemma 3. *Let N, N' be two nodes in a derivation tree with respective labels $\Lambda \vdash \Phi$, $\Lambda' \vdash \Phi'$. Whenever N' is a descendant of N the following holds:*

1. *$\Lambda' \cup \Phi' \models_{\mathcal{T}} \Lambda \cup \Phi$;*
2. *Λ' is satisfiable in $\mathcal{T}$ iff Λ is satisfiable in $\mathcal{T}$.*

The DPLL($\mathcal{T}$) calculus is easily proven terminating, in the sense that every sequent $\Lambda \vdash \Phi$ has a (finite) proof tree in DPLL($\mathcal{T}$). Using termination and Lemma 3, we prove in [15] that it is also sound and complete.

Proposition 4 (Soundness and Completeness). *A clause set Φ is unsatisfiable in $\mathcal{T}$ iff the sequent $\emptyset \vdash \Phi$ has a refutation tree in* DPLL($\mathcal{T}$).

We also prove that DPLL($\mathcal{T}$) is confluent in the following sense.

Proposition 5 (Confluence). *If Φ is unsatisfiable in $\mathcal{T}$, then every proof tree of $\emptyset \vdash \Phi$ in* DPLL($\mathcal{T}$) *is a refutation tree.*

Let us say that a derivation strategy for DPLL($\mathcal{T}$) is *fair* iff it produces a proof tree from every initial sequent $\emptyset \vdash \Phi_0$. Proposition 5 entails that any fair derivation strategy in DPLL($\mathcal{T}$) is complete.

As in the DPLL calculus, the final context of a successful branch in a derivation tree in DPLL($\mathcal{T}$) of a sequent $\emptyset \vdash \Phi_0$ describes a "partial model" of Φ_0. More precisely, and more generally, we have the following.

Proposition 6. *Let $\Lambda_0 \vdash \Phi_0$ be a sequent such that Λ_0 is satisfiable in $\mathcal{T}$ and $\Lambda_0 \vdash \Phi_0$ has a derivation tree in* DPLL($\mathcal{T}$) *with a successful branch $\Lambda_0 \vdash \Phi_0, \ldots, \Lambda_n \vdash \emptyset$. Let*

$$\Lambda := \{p \mid p \in Ats(\Phi_0) \text{ and } \Lambda_n \models_{\mathcal{T}} p\} \cup \{\neg p \mid p \in Ats(\Phi_0) \text{ and } \Lambda_n \models_{\mathcal{T}} \neg p\}$$

Then, (i) Λ is satisfiable in $\mathcal{T}$ and (ii) $\Lambda \models_{\mathcal{T}} \Phi_0$, i.e., every model of $\mathcal{T}$ that satisfies Λ is also a model of Φ_0.

We call Λ above a *partial model* of Φ_0 because in general it may contain only a subset of the atoms in Φ_0. When this is the case, nothing can be said without further computation about the truth value of the atoms in $Ats(\Phi_0) \setminus Ats(\Lambda)$. In particular, and contrarily to the DPLL calculus, one cannot always take the atoms not in Λ to be false. To do that more assumptions on the theory $\mathcal{T}$ are needed. As we show in [15], a sufficient assumption is that the background theory $\mathcal{T}$ is *convex* (see Definition 7 in the next section).

4 Strategies for DPLL($\mathcal{T}$)

In spite of the simplicity of the DPLL method and its extension with decision procedures, one should be careful in assuming that the very same optimization that work for DPLL in the propositional case also work in the modulo theories

case. Some optimizations simply become incorrect in the general case;[4] some others, although still correct, may lose their competitiveness with respect to other techniques. A survey of which of the optimization strategies used in current DPLL-based system remain correct or effective when applied to the DPLL($\mathcal{T}$) calculus is beyond the scope of this paper. In this section, we provide just a sample of them, concentrating on the changes they need to remain correct.

Literal Selection. The choice of the literal to which to apply the **split** rule is a critical one in DPLL-based systems. The SAT literature provides ample experimental evidence that on many problem classes a change in the literal selection strategy can improve (or degrade) performance by orders of magnitude.

Now, not all literal selection strategies from the SAT literature lift immediately to DPLL($\mathcal{T}$). In general, the reason is that some of these strategies are based on specific properties of propositional logic. An example would be, in terms of our calculi, the property that a literal's truth value gets determined only after it or its complement has been added to the current context. This property holds in DPLL but not in DPLL($\mathcal{T}$) typically.[5] This means that strategies based on the number of (positive/negative) occurrences of a literal in the current clause set are not necessarily as effective as in the propositional case. When there is a background theory, this number is a much less accurate measure of the impact a literal can have in unit propagation: in extreme cases, a literal may occur only once but entail in the theory all the other literals in the set.

In the following we discuss a general strategy, specializable in a number of ways, that is just as effective in the general case as in the propositional one. The strategy is interesting also because its correctness for DPLL($\mathcal{T}$) is not immediate. As a matter of fact, the strategy is incorrect unless the theory $\mathcal{T}$ is *convex*.

Definition 7. *A theory $\mathcal{T}$ is* convex *iff for every set Λ of literals and every finite non-empty set P of positive literals, $\Lambda \models_{\mathcal{T}} \bigvee_{p \in P} p$ iff $\Lambda \models_{\mathcal{T}} p$ for some $p \in P$.*

By well known results about Horn logic (see, e.g. [10]), one can show that the class of convex theories (properly) includes all Horn theories.

The general strategy we propose is motivated by the following result.

Proposition 8. *Let Λ be a set of ground literals satisfiable in a convex theory $\mathcal{T}$, and let Φ be a set of non-positive ground clauses. If $\Lambda \not\models_{\mathcal{T}} p$ for all $p \in Ats(\Phi)$, then $\Lambda \cup \Phi$ is satisfiable in $\mathcal{T}$.*

Proposition 8 entails that for any fair derivation strategy in DPLL($\mathcal{T}$) with $\mathcal{T}$ convex, one can restrict the application of the **split** rule to literals that occur in positive clauses in the current set. In fact, suppose a branch in the derivation tree of some initial sequent $\emptyset \vdash \Phi_0$ contains a sequent $\Lambda \vdash \Phi$ with no positive clauses and such that no atom p is entailed by Λ in $\mathcal{T}$. By Lemma 3, Λ is clearly

[4] This is also recognized in [4], which discusses as an example the incompleteness of the "pure literal" rule in the modulo theories case.

[5] Consider, for instance, the derivation trees in DPLL($\mathcal{T}$) of the sequent $\emptyset \vdash \{p(a),\ \neg q(a) \vee r(b)\}$ where $\mathcal{T} := \{\ \forall x\ p(x) \Rightarrow q(x)\ \}$.

satisfiable in $\mathcal{T}$, which implies by Proposition 8 that $\Lambda \cup \Phi$ is also satisfiable in $\mathcal{T}$. But then, by Lemma 3 again, $\emptyset \vdash \Phi_0$ is satisfiable in $\mathcal{T}$. In practice this means that one can stop a derivation of $\emptyset \vdash \Phi_0$ above as soon as a sequent like $\Lambda \vdash \Phi$ is generated. This optimization applies with no loss of completeness to any literal selection strategy for the DPLL calculus—and is in fact applied, for instance, in the DPLL-based system described in [12]. In terms of the result above this is justified by the fact that the empty theory is trivially convex. But the optimization is not correct for arbitrary theories.[6]

Another consequence of the previous result concerns clauses with *undetermined* literals, that is, literals l in a sequent $\Lambda \vdash \Phi$, such that neither $\Lambda \models_{\mathcal{T}} l$ nor $\Lambda \models_{\mathcal{T}} \overline{l}$. As long as a clause C has one undetermined negative literal $\neg p$ in the current sequent of a derivation, it is not necessary to apply the **resolve** rule to the other literals of C. Delaying the application of **resolve** to C until $\neg p$ becomes determined (if ever) causes no loss of completeness. In the worst case one ends up with a sequent $\Lambda \vdash \Phi$ all of whose clauses have an undetermined negative literal. But then, as one can easily show by an application of Proposition 8, the sequent is guaranteed to be satisfiable in $\mathcal{T}$. In practice, one may want to try to further simplify a clause like C above anyway because that may lead to more unit propagation. See [15] for a more detailed discussion on this.

Lemma Generation. Modern SAT solvers exhibit a primitive but highly effective form of learning. When their sequence of split choices leads them to the generation of the empty clause, they perform some kind of conflict analysis on the failure and then generate a clause that records, in a sense, the decisive wrong choices. This clause is then used to avoid repeating those wrong choices later in the search. This sort of process can applied to the DPLL($\mathcal{T}$) calculus as well. Its logical underpinning is provided by the following result.

Proposition 9. *Let* $\mathbf{B}$ *be a branch in a derivation tree of the sequent* $\emptyset \vdash \Phi_0$ *and let* $\Lambda_m \vdash \Phi_m$ *be the sequent labeling* $\mathbf{B}$*'s leaf. If* $\mathbf{B}$ *is failed, then there is a subset* S *of* Λ_m *such that* $\Phi_0 \models_{\mathcal{T}} \bigvee_{l \in S} \overline{l}$.

The proposition implies that from the literals asserted along a failed branch of a derivation tree it is possible to generate a clause that is a logical consequence in $\mathcal{T}$ of the initial clause set—hence the name *lemma*. One can use a lemma to refine the search for a successful derivation in implementations of the calculus by adding it to the current clause set as soon as it is discovered. Since the lemma is a consequence (in $\mathcal{T}$) of the initial clause set, its addition preserves the satisfiability of the set. However, as in the case of DPLL solvers, together with a proper derivation strategy, the lemma's presence makes sure that certain portions of the search space leading to unsuccessful derivations are never explored.

We prove in [15] a stronger result of which the above proposition is an immediate corollary. This result shows that for any given failed branch $\mathbf{B}$, the set S in Proposition 9, often called the *conflict set* in the SAT literature, can be

[6] Consider for instance the non-convex theory $\mathcal{T} := \{p \vee q\}$ and the $\mathcal{T}$-unsatisfiable input set $\{\neg p \vee \neg q, p \vee \neg q, \neg p \vee q\}$.

always chosen so that it includes none of the literals asserted by the **assert** rule—we call such a set a *split conflict set induced by* **B**. Intuitively, the reason for this possibility is that the contribution of such literals to a failed branch can be always traced back to literals asserted previously by the **split** rule. This property is well-known in the SAT world and is exploited in all systems with lemma learning. These systems use techniques aimed not only at finding conflict sets like S above, but also at minimizing their size. The idea is that the smaller the conflict set, the less assertions it takes for the corresponding lemma to drive the search away from dead ends in the search space. The challenge in the DPLL($\mathcal{T}$) case is again that current conflict set discovery and minimization techniques are based on specific properties of propositional logic, and so they do not immediately lift to DPLL($\mathcal{T}$). Now, there is a general, if naive, algorithm for producing minimal conflict sets from a failed branch in DPLL($\mathcal{T}$): once a branch $\emptyset_0 \vdash \Phi_0, \ldots, \Lambda_n \vdash \Box$ is generated by a system implementing the calculus,

1. collect in a set Ψ the clause in Φ_0 that has reduced to $\Box$ in the last sequent, plus all the clauses of Φ_0 that became unit at some point in the branch;
2. where S_n is the set of literals in Λ_n asserted by the **split** rule, consider every subset S of S_n and run the system recursively on $S \vdash \Psi$;
3. return any minimal S among the above for which the system fails.

Considering all possible subsets of S_n above to discover and select conflict sets is clearly impractical because its worst-case time complexity is exponential in the size of S_n. We describe in [15] a greedy algorithm whose complexity is instead quadratic. The down-side of this algorithm is that it returns only one minimal conflict set per failed branch. Furthermore, although minimal, the returned set may not be optimal, in the sense of having the smallest cardinality among all minimal conflict sets induced by the branch. However, our initial experimental evidence seems to indicate that this is not a problem in practice because failed branches rarely generate more than one minimal conflict set.

Pruning. Additional optimizations in SAT solvers involve explicit pruning strategies, as opposed or in addition to the implicit pruning caused by lemmas. In systems that traverse the search space in a depth-first manner—basically all the DPLL-based systems we know of—this is often achieved as some form of non-chronological backtracking (aka, intelligent backjumping) that skips entire areas of the search space. In the context of DPLL($\mathcal{T}$), a simple and complete heuristic for non-chronological backtracking is suggested by the following result.

Proposition 10. *Let* **T** *be a derivation tree containing a node N with label $\Lambda \vdash \Phi$ and successors N_1 and N_2 with respective labels $\Lambda, l \vdash \Phi$ and $\Lambda, \overline{l} \vdash \Phi$. Suppose all branches of the subtree of* **T** *rooted at N_1 are failed. Let Φ_f be the set of clauses in Φ that reduce to the empty clause in one of those failed branches. If $\Lambda, \overline{l} \vdash \Phi_f$ is unsatisfiable in $\mathcal{T}$ then $\Lambda, \overline{l} \vdash \Phi$ is also unsatisfiable in $\mathcal{T}$.*

Operationally, this result should be interpreted as follows. If one has verified that the branches of one of the two subtrees of a "split node" $\Lambda \vdash \Phi$ are all

failed, one can sometimes avoid exploring the other subtree altogether by doing the following. First build the set Φ_f defined in the proposition and then verify separately the unsatisfiability of Φ_f under the context $\Lambda \cup \{\overline{l}\}$ (by exploring the derivation tree of $\Lambda, \overline{l} \vdash \Phi_f$). If $\Lambda, \overline{l} \vdash \Phi_f$ is unsatisfiable in $\mathcal{T}$, do not bother exploring the subtree rooted at $\Lambda, \overline{l} \vdash \Phi$ because all of its branches are guaranteed to be failed. If the derivation tree of $\Lambda, \overline{l} \vdash \Phi_f$ has a successful branch, nothing can be said in general on the satisfiability of $\Lambda, \overline{l} \vdash \Phi$, therefore do explore the subtree rooted at $\Lambda, \overline{l} \vdash \Phi$.

In [15], we describe and prove complete a smarter, but also more expensive, pruning strategy that can be used in conjunction with lemma generation. To decide whether to prune or not, the strategy performs an analysis of the split conflict sets induced by each failed branch passing through a node like N in the proposition above. The analysis of the lemmas and the correctness of the strategy depend on the following general result.

Proposition 11. *Let* $\mathbf{T}$ *be a derivation tree containing a node* N *with label* $\Lambda \vdash \Phi$ *and successors* N_1 *and* N_2 *with respective labels* $\Lambda, l \vdash \Phi$ *and* $\Lambda, \overline{l} \vdash \Phi$. *Suppose that all branches* $\mathbf{B}_1, \ldots, \mathbf{B}_n$ *of* $\mathbf{T}$ *that end in the subtree of* $\mathbf{T}$ *rooted at* N_1 *are failed. For* $j \in \{1, \ldots, n\}$, *let* S_j *be a split conflict set induced by branch* $\mathbf{B}_i$. *If* l *does not occur in any of the* S_j*'s, then* $\Lambda \vdash \Phi$ *is unsatisfiable in* $\mathcal{T}$.

5 Conclusion

We have presented DPLL($\mathcal{T}$), a calculus based on the DPLL method that can be used to prove the satisfiability of ground formulas in theories with decidable ground consequences. The main attractiveness of this calculus is that it can incorporate, with proper changes, various types of optimization developed by the SAT community for the DPLL method. The calculus represents at least in principle an improvement over other approaches with the same goal because it allows a tighter integration of theory-specific satisfiability procedures into a DPLL-based engine. In fact, and differently from other approaches, in DPLL($\mathcal{T}$) the decision procedure is used for driving the engine's search at each derivation step, instead of just checking the viability of a possible solution after it has been computed by the engine.

An additional contribution of the work presented here is that it frames the problem of integrating decision procedures into the DPLL method in a declarative setting. The essence of the integration is captured by a simple calculus, DPLL($\mathcal{T}$), that abstracts away control aspects and optimization issues. These aspects can then be better described as search strategies for the calculus. We believe that such an approach makes it easier to describe, compare, and prove correct present and future optimizations and variants of the method.

We are currently working on an implementation of a DPLL($\mathcal{T}$)-based system with the goal of verifying experimentally the extent of the calculus' strengths in practical applications.

References

1. Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. SAT-based procedures for temporal reasoning. In S. Biundo and M. Fox, editors, *Proceedings of the 5th European Conference on Planning (Durham, UK)*, volume 1809 of *Lecture Notes in Computer Science*, pages 97–108. Springer, 2000.
2. Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. A SAT-based approach for solving formulas over boolean and linear mathematical propositions. In Reiner Hähnle, editor, *Proceedings of the 18th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence. Springer, 2002. (to appear).
3. Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. Validity checking for combinations of theories with equality. In M. K. Srivas and A. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (Palo Alto, CA)*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 1996.
4. Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In J. C. Godskesen, editor, *Proceedings of the International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, 2002. (to appear).
5. Nikolaj S. Bjørner, Mark. E. Stickel, and Tomás E. Uribe. A practical integration of first-order reasoning and decision procedures. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction, CADE-14 (Townsville, Australia)*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 101–115, 1997.
6. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
7. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.
8. Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. Presented at the Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02), Cincinnati, USA, May 2002.
9. Jon W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, Departement of computer and Information science, University of Pennsylvania, Philadelphia, 1995.
10. Wilfrid Hodges. Logical features of Horn clauses. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1, pages 449–503. Oxford University Press, 1993.
11. Joxan Jaffar and Michael Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
12. Shie-Jue Lee and David A. Plaisted. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 9(1):25–42, August 1992.
13. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.
14. Greg Nelson and Dave Detlefs. *The Simplify user's manual*. Compaq Systems Research Center. (`http://research.compaq.com/SRC/esc/Simplify.html`).
15. Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. Technical report, Department of Computer Science, University of Iowa, 2002.