Splitting on Demand in SAT Modulo Theories

Clark Barrett[1]
barrett@cs.nyu.edu

Robert Nieuwenhuis[2]
roberto@lsi.upc.edu

Albert Oliveras[2]
oliveras@lsi.upc.edu

Cesare Tinelli
tinelli@cs.uiowa.edu

Department of Computer Science
The University of Iowa

Report No. 06-05

August 2006

---

[1]Department of Computer Science, New York University, 251 Mercer St., New York, NY 10012, USA.
[2]LSI Department, Technical University of Catalonia, Jordi Girona 1, E-08034 Barcelona, Spain.

Splitting on Demand in SAT Modulo Theories

Clark Barrett

Department of Computer Science
New York University
251 Mercer St., New York, NY 10012, USA
`barrett@cs.nyu.edu`

Robert Nieuwenhuis

LSI Department
Technical University of Catalonia
Jordi Girona 1, E-08034 Barcelona, Spain
`roberto@lsi.upc.edu`

Albert Oliveras

LSI Department
Technical University of Catalonia
Jordi Girona 1, E-08034 Barcelona, Spain
`roberto@lsi.upc.edu`

Cesare Tinelli

Department of Computer Science
The University of Iowa
14 MacLean Hall, Iowa City, IA 52242, USA
`tinelli@cs.uiowa.edu`

August 2006

Abstract

Lazy algorithms for *Satisfiability Modulo Theories* (SMT) combine a generic DPLL-based SAT *engine* with a *theory solver* for the given theory T that can decide the T-consistency of conjunctions of ground literals. For many theories of interest, theory solvers need to reason by performing internal case splits. Here we argue that it is more convenient to delegate these case splits to the DPLL engine instead. The delegation can be done on demand for solvers that can encode their internal case splits into one or more clauses, possibly including new constants and literals. This results in drastically simpler theory solvers. We present this idea in an improved version of DPLL(T), a general SMT architecture for the lazy approach, and formalize and prove it correct in an extension of *Abstract DPLL Modulo Theories*, a framework for modeling and reasoning about lazy algorithms for SMT. A remarkable additional feature of the architecture,

also discussed in the paper, is that it naturally includes an efficient Nelson-Oppen-like combination of multiple theories and their solvers.

1 Introduction

The performance of propositional SAT solvers based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [DP60, DLL62] has importantly improved during the last years, and DPLL-based solvers are becoming the tool of choice for attacking more and more practical problems. The DPLL procedure has also been adapted for handling problems in more expressive logics, and, in particular, for the *Satisfiability Modulo Theories (SMT)* problem: deciding the satisfiability of ground first-order formulas with respect to background theories such as the integer or real numbers, or arrays. SMT problems frequently arise in formal hardware and software verification applications, where typical formulas consist of very large sets of clauses like:

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

with propositional atoms as well as atoms over (combined) theories like the integers, arrays, or Equality with Uninterpreted Functions (EUF). SMT has become a very active area of research, and efficient SMT solvers exist that can handle (combinations of) many such theories T (see also the SMT problem library [TR05] and the SMT Competition [BdMS05]). . Currently most SMT solvers follow the so-called *lazy* approach to SMT, combining (i) *theory solvers* that can handle *conjunctions* of literals over the given theory T, with (ii) DPLL *engines* for dealing with the Boolean structure of the formulas.

DPLL(T) is a general SMT architecture for the lazy approach [GHN$^+$04]. It consists of a DPLL(X) engine, whose parameter X can be instantiated with a T-solver $Solver_T$, thus producing a DPLL(T) system. The DPLL(X) engine always considers the problem as a purely propositional one. For example, if the theory T is EUF, at some point DPLL(X) might consider a partial assignment containing, among many others, the four literals $a{=}b$, $f(a){=}c$, $f(b){=}d$, and $c \neq d$ without noticing its T-inconsistency, because it just considers such literals as propositional (syntactic) objects. But $Solver_T$ continuously analyzes the partial model that DPLL(X) is building (a conjunction of literals). It can warn DPLL(X) about this T-inconsistency, and generate a clause, called a *theory lemma*, like $a{\neq}b \vee f(a){\neq}c \vee f(b){\neq}d \vee c{=}d$, which can be used by DPLL(X) for backjumping. $Solver_T$ sometimes also does *theory propagation*: as soon as, e.g., $a{=}b$, $f(a){=}c$, and $f(b){=}d$ become true, it can notify DPLL(X) about T-consequences like $c{=}d$ that occur in the input formula. The modular DPLL(T) architecture is flexible, and can be implemented efficiently: the BarcelogicTools implementation of DPLL(T) won all the four divisions it entered at the 2005 SMT Competition [BdMS05].

Here we propose an improved version of the DPLL(T) architecture, to rationalize and simplify the construction of lazy SMT systems where $Solver_T$ does reasoning by cases. We present it formally by means of a corresponding extension of *Abstract DPLL Modulo Theories*, a uniform, declarative framework introduced in [NOT05] for modeling and reasoning about lazy SMT procedures.

Example 1. In the array theory, the equation $read(write(A, i, v), j) = read(A, j)$ holds in two situations: when the indices i and j are distinct, or when they are equal but the $write(A, i, v)$ changes nothing, i.e., the value of array A at position i is already v. Deciding the T-consistency of a large conjunction of equations and disequations over arrays essentially requires $Solver_T$ to do an analysis of many Boolean combinations of such cases. In the extension of DPLL(T) we propose here, $Solver_T$ can delegate all such case splittings to the DPLL(X) engine, e.g., it can *demand* DPLL(X) to split on atoms like $i{=}j$, by sending it a *theory lemma* (i.e., a ground clause valid in the theory) that encodes the split—for instance, a clause like $read(write(A, i, v), j) {\neq} read(A, j) \lor i {\neq} j \lor read(A, i) = v$. ∎

The main novelty, and complication, versus the previous version of DPLL(T) is that the lemma may contain atoms that *do not occur in the input formula*. Sometimes even new constant symbols may be introduced. For example, in (fragments of) set theory [CZ00], a set disequality $s \neq s'$ may be handled by the theory solver by reducing it to the disjunction $(a \in s \land a \notin s') \lor (a \notin s \land a \in s')$, where a is a fresh Skolem constant.

Centralizing all case splitting in the engine allows one to avoid the duplication of search functionality in the theory solver and drastically simplify its implementation, since a case splitting infrastructure is no longer necessary. Roughly, the solver's only requirement reduces to being able to detect T-inconsistencies once all case splits it has requested have been done.

The main contribution of this paper is a general and formal specification of this sort of architecture, together with a rigorous proof of its correctness. The relevance of this architecture is that it unquestionably leads to simpler solvers for theories that require case splits—in practice, all theories T where checking the T-inconsistency of ground literals is NP-hard.

In many SMT applications the background theory T is defined as a combination of several component theories $T_1, \ldots, T_n$, each with its own local solver. An important aspect of our approach is that it can be naturally refined to accommodate such combined theories, giving rise to a DPLL($T_1, \ldots, T_n$) architecture.

Example 2. Let T be the union of two disjoint theories T_1 and T_2 where T_1 is EUF and T_2 is (some fragment of) arithmetic, two of the most common theories in SMT. Let F be the conjunction $a{=}b \ \land \ f(a){-}c \leq 3 \ \land \ f(b){-}c \geq 4$ over the combined signature of T_1 and T_2. Introducing new constants c_1 and c_2, F can be *purified*, into an equisatisfiable conjunction of the T_1-pure formula F_1 and the T_2-pure formula F_2 below:

$$a{=}b \ \land \ c_1{=}f(a) \ \land \ c_2{=}f(b) \qquad\qquad c_1{-}c \leq 3 \ \land \ c_2{-}c \geq 4 \ .$$

In general, an *arrangement* A for such pure conjunctions $F_1 \ldots F_n$ is a conjunction saying, for every two constants *shared* between at least two different F_i's, whether the constants are equal or distinct. A general combination result underpinning the Nelson-Oppen method [NO79] states that for *stably infinite* and *signature disjoint* T_i's, F is T-consistent if, and only if, for some arrangement A each $F_i \land A$ is T_i-consistent (see, e.g., [TH96] for precise

definitions and details). This can be decided by the respective T_i-solvers. In this example, F is T-inconsistent since $F_1 \wedge c_1 \neq c_2$ is T_1-inconsistent and $F_2 \wedge c_1 = c_2$ is T_2-inconsistent.

In practice, it is useful if each T_i-solver is able to *generate* all clauses $c_1 = c_1' \vee \ldots \vee c_k = c_k'$ over the shared constants that are T_i-entailed by the conjunction F_i. For *convex* T_i, these entailed clauses are in fact always unit. It is not difficult to see that, if these two properties hold for all T_i, we only have to consider one arrangement: the one where every two constants not equated in a propagated equality are distinct. But usually the situation is less ideal. If some T_i is non-convex, it is necessary to do case splitting over the T_i-entailed non-unit clauses, and if some T_i-solver has limited or too expensive generation capabilities, the possible arrangements need to be (partially) guessed and tried.

By centralizing case splitting into the DPLL(X) engine and extending it to equalities over shared constants we can use the engine, in effect, to efficiently enumerate the arrangements on demand, that is, as requested by the individual theory solvers. Note that in the resulting DPLL($T_1, \ldots, T_n$) architecture, the engine will again handle literals possibly not in the input clauses, namely, (dis)equalities between shared variables. ∎

Section 2 of this paper introduces and discusses the correctness of an Extended Abstract DPLL Modulo Theories framework that formalizes our approach. Section 3 illustrates how to use the framework to avoid internal case splits in a very general class of theory solvers. Section 4 discusses the application of the framework to DPLL($T_1, \ldots, T_n$). Finally, Section 5 concludes.

Related work. Some of the ideas formalized in this paper on centralizing case splits in the Boolean engine are implemented in the system SVC [BDL96] and and its successors CVC [BDS02]. and CVC-Lite [BB04]. But apart from a brief note in Clark Barrett's PhD thesis (in Section 3.5.1 of [Bar03]), we are not aware of any other description of them in the literature.

Bozzano *et al.* propose in [BBC$^+$06] to use the Boolean engine in multi-theory SMT systems to do case splitting over the space of all possible arrangements. In contrast to this work, there the centralization of case-splitting concerns only equalities between shared constants, as needed by the Nelson-Oppen method. As far as theory solver combination is concerned, our approach and that in [BBC$^+$06] are in a sense dual, possibly as a consequence of their different motivations. Simplifying a bit, in [BBC$^+$06] the theory solvers are (or can be) completely unaware of each other. The DPLL engine is in charge of identifying shared constants and feeding (dis)equalities between them as appropriate to the solvers. This way, off-the-shelf decision procedures can be used as theory solvers. In our case, the roles are reversed. As we will see, the solvers are aware of their shared constants, and are in charge of producing lemmas containing (dis)equalities between them, for the engine to split on. The advantage in this case is that the same mechanism already in place for splitting on demand can be used for combination as well, with no changes to the engine.

2   Extended Abstract DPLL Modulo Theories

In this section, we briefly describe the Abstract DPLL Modulo Theories framework (see [NOT05] for more details) and then extend it so that it can be used to formalize our new version of DPLL(T) and, more generally, SMT approaches where new atoms and new symbols are introduced.

2.1   Abstract DPLL Modulo Theories

As usual in SMT, given a theory T (a set of closed first-order formulas), we will only consider the SMT problem for *ground* (and hence quantifier-free) CNF formulas F. Such formulas may contain *free* constants, i.e., 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, we will assume that all formulas satisfy these restrictions.

The formalism we describe is based on a set of *states* together with a binary relation $\Longrightarrow$ (called the *transition relation*) over these states, defined by means of *transition rules*. Starting with a state containing an input formula F, one can use the rules to generate a finite sequence of states, where the final state indicates whether or not F is T-consistent.

A *state* is either the distinguished state *FailState* (denoting T-unsatisfiability) or a pair of the form $M \parallel F$, where M is a sequence of literals, with $\emptyset$ denoting the empty sequence, and F is a formula in conjunctive normal form (CNF), i.e., a finite set of disjunctions of literals. We additionally require that M never contains both a literal and its negation and that each literal in M is annotated as either a *decision* literal (indicated by l^{d}) or not. Frequently, we will refer to M as a *partial assignment* or consider M just as a set or conjunction of literals, ignoring both the annotations and the order of its elements.

In what follows, a possibly subscripted or primed lowercase l *always* denotes a literal. Similarly C and D always denote clauses (disjunctions of literals), F and G denote conjunctions of clauses, and M and N denote partial assignments.

We write $M \models F$ to indicate that M propositionally satisfies F. If C is a clause $l_1 \vee \ldots \vee l_n$, we sometimes write $\neg C$ to denote the formula $\neg l_1 \wedge \ldots \wedge \neg l_n$. We say that C is *conflicting* in a state $M \parallel F, C$ if $M \models \neg C$.

A formula F is called *T-(in)consistent* if $F \wedge T$ is (un)satisfiable in the first-order sense. We say that M *is a T-model of F* if $M \models F$ and M, seen as a conjunction of literals, is T-consistent. It is not difficult to see that F is T-consistent if, and only if, it has a T-model. 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 $\emptyset \models_T C$.

We start with the transition system first presented in [NOT05].

Definition 3. Abstract DPLL Modulo Theories consists of the following rules:

UnitPropagate :

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

Decide :

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

Fail :

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

Restart :

$$M \parallel F \quad \Longrightarrow \quad \emptyset \parallel F$$

T-Learn :

$$M \parallel F \quad \Longrightarrow \quad M \parallel F, C \quad \textbf{if} \left\{ \begin{array}{l} \text{each atom of } C \text{ occurs in } F \text{ or in } M \\ F \models_T C \end{array} \right.$$

T-Forget :

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

T-Backjump :

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

T-Propagate :

$$M \parallel F \quad \Longrightarrow \quad M\, l \parallel F \quad \textbf{if} \left\{ \begin{array}{l} M \models_T l \\ l \text{ or } \neg l \text{ occurs in } F \\ l \text{ is undefined in } M \end{array} \right.$$

The *Basic DPLL Modulo Theories system* consists of the rules Decide, Fail, UnitPropagate, T-Propagate and T-Backjump. We denote the transition relation defined by these rules by $\Longrightarrow_{\mathrm{B}}$. We denote the transition relation defined by all the rules by $\Longrightarrow_{\mathrm{FT}}$. □

For a transition relation $\Longrightarrow$, we denote by $\Longrightarrow^*$ the reflexive-transitive closure of $\Longrightarrow$. We call any sequence of the form $S_0 \Longrightarrow S_1$, $S_1 \Longrightarrow S_2$, ... a *derivation*, and denote it by $S_0 \Longrightarrow S_1 \Longrightarrow S_2 \Longrightarrow \dots$. We call any subsequence of a derivation a *subderivation*. If $S \Longrightarrow S'$ we say that there is a *transition* from S to S'. A state S is *final* with respect to $\Longrightarrow$ if there are no transitions from S.

The relevant derivations in the Abstract DPLL Modulo Theories system are those that start with a state of the form $\emptyset \parallel F$, where F is a formula to be checked for T-consistency, and end in a state that is final with respect to $\Longrightarrow_{\mathrm{B}}$.

The following two theorems summarize the main results of [NOT05] for the transition relation $\Longrightarrow_{\mathrm{FT}}$.

Theorem 4 (Termination of $\Longrightarrow_{\mathbf{FT}}$). *A derivation Der of the form:* $\emptyset \parallel F = S_0 \Longrightarrow_{\mathrm{FT}}$
$S_1 \Longrightarrow_{\mathrm{FT}} \ldots$ *is finite if the following two conditions hold:*

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

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

Theorem 5. *Let $\emptyset \parallel F \Longrightarrow^*_{\mathrm{FT}} S$ be a derivation where (i) S is final wrt $\Longrightarrow_{\mathrm{B}}$, 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.* □

2.2 The Extended Abstract DPLL Modulo Theories System

Any realization of the Abstract DPLL Modulo Theories framework, in addition to imple-
menting the rules anda terminating strategy, must be able to determine the T-consistency
of M when a final state $M \parallel F$ is reached. For this purpose, one typically assumes the
existence of $Solver_T$ which can do precisely that.

However, for some important theories, determining the T-consistency of a conjunction
of literals requires additional *internal* case splitting. In order to simplify $Solver_T$ and
centralize the case splitting in the DPLL engine, it is desirable to relax the requirement on
$Solver_T$ by allowing it to demand that the DPLL engine do additional case splits before
determining the T-consistency of the partial assignment. For flexibility—and because it
is needed by actual theories of interest—the theory solver should be able to demand case
splits on literals that do not appear in M or F and possibly even contain fresh constant
symbols.

It is not hard to see, however, that allowing this kind of flexibility poses a potential
termination problem. We can overcome this difficulty if, for any input formula F, the set
of all literals needed to check the T-consistency of F is finite. More precisely, as a purely
theoretical construction, we assume that for every input formula F there is a finite set
$\mathcal{L}(F)$ of literals containing all literals on which a given theory solver may demand case
splits when starting with a conjunction of literals from F. For example, for a solver for the
theory of arrays $\mathcal{L}(F)$ could contain atoms of the form $i = j$, where i and j are array indices
occurring in F. This technical requirement poses no limitations on any of the practically
useful theory solver procedures we are aware of (see Section 3). Also, for the proofs here
there is no need to construct the set $\mathcal{L}(F)$. It is enough to know that it exists. Formally,
we require the following.

Definition 6. $\mathcal{L}$ is a *suitable literal-generating function* if for every finite set of literals L:

1. $\mathcal{L}$ maps L to a new finite set of literals L' such that $L \subseteq L'$.

2. For each atomic formula α, $\alpha \in \mathcal{L}(L)$ iff $\neg \alpha \in \mathcal{L}(L)$.

3. If L' is a set of literals and $L \subseteq L'$, then, $\mathcal{L}(L) \subseteq \mathcal{L}(L')$ (monotonicity).

4. $\mathcal{L}(\mathcal{L}(L)) = \mathcal{L}(L)$ (idempotence). □

For convenience, given a formula F, we denote by $\mathcal{L}(F)$ the result of applying $\mathcal{L}$ to the set of all literals appearing in F.

The introduction of new constant symbols poses potential problems not only for termination, but also for soundness. One property of the transition relation $\Longrightarrow_{\text{FT}}$ is that whenever $\emptyset \parallel F \Longrightarrow^*_{\text{FT}} M \parallel F'$, the formulas F and F' are T-equivalent. This will no longer be true if we allow the introduction of new constant symbols. However, it is sufficient to simply ensure T-equisatisfiability of F and F'. To this end, we introduce the following definition.

Definition 7. Given a formula F and a formula G, we define $\gamma_F(G)$ as follows:

1. Let G' be the formula obtained by replacing each free constant symbol in G that does not appear in F with a fresh variable.

2. Let $\bar{v}$ be the set of all fresh variables introduced in the previous step.

3. Then, $\gamma_F(G) = \exists \bar{v}. G'$. □

Now we can give a new transition rule called Extended T-Learn which replaces T-Learn and allows for the desired additional flexibility.

Definition 8. The *Extended DPLL Modulo Theories system*, denoted as $\Longrightarrow_{\text{XT}}$, consists of the rules of Basic DPLL Modulo Theories, together with the rules Restart, T-Forget and the rule:

Extended T-Learn

$$M \parallel F \quad \Longrightarrow \quad M \parallel F, C \quad \textbf{if} \; \left\{ \begin{array}{l} \text{each atom of } C \text{ occurs in } F \text{ or in } \mathcal{L}(M) \\ F \models_T \gamma_F(C) \end{array} \right.$$

□

The key observation is that an implementation using Extended T-Learn has more flexibility when a state $M \parallel F$ is reached which is final with respect to $\Longrightarrow_{\text{B}}$. Whereas before it would have been necessary for the theory solver to determine the T-consistency of M when such a state was reached, the Extended T-Learn rule allows the possibility of *delaying* a response by demanding that additional case splits (on possibly new literals appearing in the clause C) be done first. As we will show below, the properties of $\mathcal{L}$ ensure that the solver's response cannot be delayed indefinitely.

2.3 Correctness of Extended Abstract DPLL Modulo Theories

A decision procedure for SMT can be obtained by generating a derivation using $\Longrightarrow_{\mathrm{XT}}$ with a particular strategy. As with $\Longrightarrow_{\mathrm{FT}}$, the aim of a derivation is to compute 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. First, we prove some invariants.

Lemma 9. *If $\emptyset \parallel F \Longrightarrow_{\mathrm{XT}}^{*} M \parallel G$ then the following hold.*

1. *All the literals in M and all the literals in G are in $\mathcal{L}(F)$.*

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. *$G \models_T F$ and for some H, $F \models_T \gamma_H(G)$.*

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 $G, l_1, \ldots, l_i \models_T M_i$ for all i in $0 \ldots n$.* $\qquad\square$

PROOF. Since all four properties trivially hold in the initial state $\emptyset \parallel F$, we only need to prove that all the rules preserve them. Notice that Fail preserves them vacuously since *FailState* is not of the form $M \parallel G$. For the others, consider a step $M' \parallel F' \Longrightarrow_{\mathrm{XT}} M'' \parallel F''$ and assume all properties hold in $M' \parallel F'$.

It is easy to see that Property 1 holds in $M'' \parallel F''$ for all rules except Extended T-Learn because none of these rules introduce new literals that aren't already present in M' or F'. The only way it could fail for Extended T-Learn is if one of the atoms in the added clause (call it C) is not in $\mathcal{L}(F)$. We know that each atom in C is either in F' or in $\mathcal{L}(M')$. If the former, we are done since we know that $F' \subseteq \mathcal{L}(F)$. If the latter, then since $M' \subseteq \mathcal{L}(F)$, it follows by monotonicity and idempotence of $\mathcal{L}$ that $\mathcal{L}(M') \subseteq \mathcal{L}(F)$.

The side conditions of the rules clearly preserve Property 2. As for Property 3, for all rules except Forget and Extended T-Learn, $F' = F''$, so it is preserved trivially. Now, consider Forget and suppose that C is the clause that was forgotten. Then $F' = F'', C$. The second part is trivial since $F'' \subseteq F'$. For the first part, we know that $F'' \models_T C$, so it follows that $F'' \models_T F'', C$. But this means that $F'' \models_T F'$, and since $F' \models F$, this implies that $F'' \models F$. Suppose on the other hand that Extended T-Learn was applied and that C is the learned clause (so that $F'' = F', C$). In this case, the first part is trivial because $F' \subseteq F''$. For the second part, suppose that $\mathcal{A}$ is a structure satisfying F. We know that $F \models_T \gamma_H(F')$ for some H. By the definition of γ, this means that there exists a structure $\mathcal{A}'$ which differs from $\mathcal{A}$ only on the symbols in F' that don't appear in H such that $\mathcal{A}'$ satisfies F'. But by the Extended T-Learn rule, we have that $F' \models_T \gamma_{F'}(C)$, so there exists some structure $\mathcal{A}''$ which differs from $\mathcal{A}'$ only on the symbols in C that don't appear in F' such that $\mathcal{A}''$ satisfies $F' \wedge C$. Let H' be some formula containing all the symbols in H except those also appearing in C but not in F'. It follows that $F \models_T \gamma_{H'}(F' \wedge C)$.

Finally, consider the fourth property. We know that M' is of the form $M_0'\ l_1\ M_1'\ \ldots\ l_n\ M_n'$, and $l_1, \ldots, l_n$ are all the decision literals of M'. If the step is an

application of Decide or Restart, there is nothing to prove. For Forget, it easily follows since $M' = M''$ and F' and F'' are T-equivalent. Similarly, for Extended T-Learn, $M' = M''$ and $F' \subseteq F''$. 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.

T-Propagate: As above, we must show that $F'', l_1, \ldots, l_n \models l$, which holds since (i) $F', l_1, \ldots, l_n \models M'$, (ii) $M' \models l$, and (iii) F' and F'' are equivalent.

T-Backjump: Assume that, in the T-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 $M_0'\ l_1\ M_1'\ \ldots\ l_j\ M_j'\ l'$. 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) $F', l_1, \ldots, l_j \models M_0'\ l_1\ M_1'\ \ldots\ l_j\ M_j'$, (ii) $M_0'\ l_1\ M_1'\ \ldots\ l_j\ M_j' \models \neg C'$, (iii) $F' \models_T C' \vee l'$ and (iv) F' and F'' are equivalent. ∎

Note that it is very easy to generate non-terminating derivations for $\Longrightarrow_{\mathrm{XT}}$, for instance, by keeping learning and forgetting the same lemma, or by applying Restart indiscriminately. The next result shows a very general strategy for avoiding generating non-terminating derivations.

The main difference in the termination argument with respect to $\Longrightarrow_{\mathrm{FT}}$ is that, while Extended T-Learn can produce lemmas with new literals, it can only produce a finite number of them thanks to the properties of $\mathcal{L}$.

Theorem 10 (Termination of $\Longrightarrow_{\mathrm{XT}}$). *Every derivation Der of the form*
$\emptyset \parallel F = S_0 \Longrightarrow_{\mathrm{XT}} S_1 \Longrightarrow_{\mathrm{XT}} \ldots$ *is finite if the following two conditions hold:*

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

2. *For every subderivation of Der of the form*
   $S_{i-1} \Longrightarrow_{\mathrm{XT}} S_i \Longrightarrow_{\mathrm{XT}} \ldots \Longrightarrow_{\mathrm{XT}} S_j \Longrightarrow_{\mathrm{XT}} \ldots \Longrightarrow_{\mathrm{XT}} S_k$ *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 $S_j \Longrightarrow_{\mathrm{XT}} \ldots \Longrightarrow_{\mathrm{XT}} S_k$ than in $S_i \Longrightarrow_{\mathrm{XT}} \ldots \Longrightarrow_{\mathrm{XT}} S_j$, or*

   - *in $S_j \Longrightarrow_{\mathrm{XT}} \ldots \Longrightarrow_{\mathrm{XT}} S_k$ a new clause is learned that is not forgotten in Der.* □

PROOF. We begin by defining a well-founded strict partial ordering $\succ$ on states. 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 atomic formulas in $\mathcal{L}(F)$. By Lemma 9-(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 $n - length(N)$, that is, $m(N)$ is the number of atomic formulas of $\mathcal{L}(F)$ "missing" from N. Now define: $M \parallel F' \succ M' \parallel F''$ if

(i) there is some i with $0 \le i \le p, p'$ 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. We can extend $\succ$ to states by simply comparing the partial assignments and adding *FailState* as an additional minimal element. It is easy to see that all Basic DPLL Modulo Theories rules result in a state that is smaller with respect to $\succ$. The rules UnitPropagate, T-Backjump, and T-Propagate decrease by case (i) of the definition and Decide decreases by case (ii).

Now, assume there exists an infinite derivation Der fulfilling the requirements. We will show that this leads to a contradiction. In a subderivation of Der without Restart steps, at each step either the state decreases with respect to $\succ$ (by the application of a Basic DPLL Modulo Theories step) or it remains equal (by the Extended T-Learn and Forget steps). Therefore, since $\succ$ is well-founded and there is no infinite subderivation consisting of only Extended T-Learn and Forget steps, there must be infinitely many Restart steps in Der.

Furthermore, because $\mathcal{L}(F)$ is finite, there are only a finite number of possible clauses that can be learned and never forgotten. Thus, at some point, there must be an infinite number of Restart steps which occur with increasing periodicity.

Finally, if between two states there is at least one Basic DPLL Modulo Theories 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, there cannot be any subderivations with more than n Basic DPLL Modulo Theories steps between two Restart steps. This contradicts the fact that there are infinitely many Restart steps with increasing periodicity in Der. ∎

Lemma 11. *If $\emptyset \parallel F \Longrightarrow^*_{\mathrm{XT}} M \parallel F'$ and there is some conflicting clause in $M \parallel F'$, i.e., $M \models \neg C$ for some clause C in F', then either Fail or T-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 $M \models \neg C$ we have, due to Lemma 9-4, that $F', l_1, \ldots, l_n \models \neg C$. If we now consider any i in $1 \ldots n$ such that $F', l_1, \ldots, l_i \models \neg C$, and any j in $0 \ldots i-1$ 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 $\neg l_1 \vee \ldots \vee \neg l_j$, and note that M is also of the form $M' \, l_{j+1} \, N$. Then T-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 T-Backjump rule:

(i) $F', l_1, \ldots, l_j, l_i \models_T \neg C$ by Lemma 9-4 and $F', l_1, \ldots, l_j, l_i \models_T C$ because C is in F', so $F', l_1, \ldots, l_j, l_i$ is unsatisfiable. Thus, $F' \models_T C' \vee \neg l_i$.
(ii) $\neg l_i$ is undefined in M' (by Lemma 9-2);
(iii) l_i occurs in M. ∎

Property 12. *If* $\emptyset \parallel F \Longrightarrow^*_{\mathrm{XT}} M \parallel F'$ *and* M *is* T*-inconsistent, then either there is a conflicting clause in* $M \parallel F'$*, or else* Extended T-Learn *applies to* $M \parallel F'$*, generating a clause enabling some Basic DPLL Modulo Theories step.* □

PROOF. If M is T-inconsistent, then there exists a subset $\{l_1, \ldots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \ldots \vee \neg l_n$. Hence, the conflicting clause $\neg l_1 \vee \ldots \vee \neg l_n$ is either in $M \parallel F'$, or else it can be learned by one Extended T-Learn step, thus enabling either Fail or T-Backjump (by Lemma 11 above). Alternatively, if there are literals in $\mathcal{L}(M)$ that are undefined in M, then Extended T-Learn can be used to learn a clause containing such a literal, enabling Decide. ∎

Lemma 11 and Property 12 show that, for a state of the form $M \parallel F$, if there is some literal of F undefined in M, or there is some conflicting clause, or M is T-inconsistent, then a rule of Basic DPLL Modulo Theories is always applicable, possibly after a single Extended T-Learn step. Together with Theorem 10 (Termination), this shows how to compute a state to which the following main theorem is applicable.

Theorem 13. *Let* Der *be a derivation* $\emptyset \parallel F \Longrightarrow^*_{\mathrm{XT}} S$*, where* S *is (i) 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*-inconsistent.*

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

PROOF. For Property 1, if S is *FailState* it is because there is some state $M \parallel F'$ such that $\emptyset \parallel F \Longrightarrow^*_{\mathrm{XT}} M \parallel F' \Longrightarrow$ *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 $M \models \neg C$. By Lemma 9-4, $F' \models_T M$, so $F' \models_T \neg C$. But C is in F', so F' must be unsatisfiable. By the second part of Lemma 9-3, every model of F can be modified to obtain a model of F'. Since F' is unsatisfiable, F must also be unsatisfiable.

Assume S is not *FailState*. Then (i) all literals of F' are defined in M since otherwise, Decide would be applicable. Similarly, (ii) there is no clause C in F' such that $M \models \neg C$ because otherwise, by Lemma 11, either Fail or T-Backjump would apply. Together (i) and (ii) imply that all clauses of F' are defined and true in M which is T-consistent. Finally, by the first part of Lemma 9-3, $F' \models_T F$, and this implies that M is a T-model of F. ∎

For a given theory T, Theorems 10 and 13 show how to obtain a decision procedure for the T-consistency of formulas as long as we have a theory solver and can prove for it the existence of a suitable literal-generating function $\mathcal{L}$ such that the following holds: for every state of the form $M \parallel F$ that is final with respect to $\Longrightarrow_{\mathrm{B}}$, the theory solver is able to

1. determine that M is T-inconsistent,

2. determine that M is T-consistent, or

3. generate a new clause via Extended T-Learn that enables some Basic DPLL Modulo Theories step.

3 Avoiding case splitting within theory solvers

In this section, we show how rule-based theory solvers can be used in the context of Extended DPLL Modulo Theories.

Recall that theory solvers only need to deal with conjunctions (equivalently, sets) of literals. Then observe that any solver deciding the T-consistency of such conjunctions in a theory where this problem is NP-hard is bound to resort to some form of case splitting.[1] We show how the Extended T-Learn rule allows such solvers to avoid any internal case splitting, and we explain why, for rule-based solvers, the existence of $\mathcal{L}$ is reasonable.

3.1 Rule-based Theory Solvers

A large class of theory solvers can be defined using inference rules that describe how to take a set of literals and transform it in some way to get new sets of literals (or $\bot$, indicating T-inconsistency). Consider a theory T. For our purposes, let us assume that an inference rule has one of the following two formats:

$$\frac{\Gamma, \Delta}{\bot} \qquad\qquad \frac{\Gamma, \Delta}{\Gamma, \Delta_1 \quad \Gamma, \Delta_2 \quad \cdots \quad \Gamma, \Delta_n}$$

where the meta-variables Γ, Δ and Δ_i represent sets of literals. We call rules of the first kind *refuting* rules and rules of the second kind *progress* rules. Typically, Δ has side-conditions or is a schema, while Γ can represent any set of literals. Progress rules describe a local change based on a small number of literals (the ones in Δ), while all of the other literals (the ones in Γ) are unchanged. For example, the following progress rule says that in any set of literals, an equation (represented by the schema $x = y$) can be replaced by its symmetric equivalent ($y = x$):

$$\frac{\Gamma, x = y}{\Gamma, y = x}$$

A refuting rule is *sound* iff any legal instance δ of Δ is T-inconsistent. A progress rule is *sound* if whenever $\Delta, \Delta_1, \ldots, \Delta_n$ are instantiated with $\delta, \delta_1, \ldots, \delta_n$ respectively, δ is T-consistent iff $\bigvee_{i=1}^{n} \delta_i$ is T-consistent. We say that a set Φ of literals is *(ir)reducible* with respect to a set of derivation rules R if (n)one of the rules in R applies to it, i.e., if (no) some subset of Φ is a legal instance of Δ in a rule of R. A *strategy* is a function that, given a reducible set of literals Φ, chooses a rule from R to apply.

---

[1]In fact, conceivably a solver may be based on case splitting even if the above T-consistency problem is polynomial, for simplicity or convenience.

Given a set R of rules and a strategy S, a *derivation tree* for a set of literals Φ is a finite tree with root Φ such that for each internal node E of the tree, E is reducible and its children are the conclusions of the rule selected by S for E. A *refutation tree* (for Φ) is a derivation tree all of whose leaves are $\bot$. A *derivation* is a sequence of derivation trees starting with the single-node tree containing Φ, where each tree is derived from the previous one by the application of a rule from R to one of its leaves. A *refutation* is a finite derivation ending with a refutation tree. A strategy S is *terminating* if every derivation using S is finite. A strategy S is *complete* if whenever Φ is T-inconsistent, S produces a refutation for Φ.

It is not hard to see that a set R of sound inference rules together with a terminating and complete strategy S provide a decision procedure for the T-consistency of sets of ground literals. In fact, all decision procedures typically associated with applications of Satisfiability Modulo Theories can be described in this way. We will now describe how such decision procedures can be incorporated into the Extended Abstract DPLL Modulo Theories formalism.

3.2   Integration with Rule-based Theory Solvers

Recall that the original DPLL Modulo Theories framework requires that for every state $M \parallel F$ that is final with respect to Basic DPLL Modulo Theories, the theory solver can determine the T-consistency of M. Given a set of sound inference rules and a terminating and complete strategy, M can be checked for T-consistency simply by generating the derivation starting with M and determining whether it results in a refutation tree or not.

Note that this process may require a large derivation tree with many branches. The purpose of the Extended T-Learn rule is to allow the theory solver to avoid having to do any splitting itself. This can be accomplished as follows. Given a state $M \parallel F$ which is final with respect to Basic DPLL Modulo Theories, the theory solver begins applying rules starting with M. However, this time, as soon as a splitting rule is encountered (a progress rule with $n > 1$), the theory solver halts and uses Extended T-Learn to return one or more clauses representing the case split. The theory solver is then suspended until another final state $M' \parallel F'$ is reached.

The obvious remaining question is how to capture the case split with a learned clause. One way to do this that will work for any rule-based theory solver is to encode the number of possible case splits using Boolean constants.

Property 14. *Any rule-based decision procedure for a theory T can be integrated with the Extended DPLL Modulo Theories framework in such a way that no case splitting is required within the theory solver.*   □

PROOF. Suppose R is a set of sound rules for T and that S is a complete and terminating strategy for R. We show that every split rule can be replaced with an application of Extended T-Learn and that an appropriate $\mathcal{L}$ function exists.

Consider first an inference rule $r \in R$ that induces a 2-way case split. For some new Boolean constant p, we can use Extended T-Learn to add the clause $(p \vee \neg p)$. Assuming that

Extended T-Learn is only applied in a final state, the only DPLL rules that can be applied are Decide, Restart, or Forget. If the DPLL engine is following a terminating strategy (as described in Theorem 10), then we can assume without loss of generality that Decide is applied to choose either p or $\neg p$. The resulting state $M' \parallel F'$ is once again final with respect to Basic DPLL Modulo Theories, so the theory solver proceeds as follows: if $p \in M'$ in the new final state, the theory solver follows the left branch of the splitting rule r; otherwise, the right branch is taken. For rules in R with an n-way case split, the n choices can be encoded using $\lceil \log n \rceil$ new Boolean constants.

Now we show the existence of an appropriate $\mathcal{L}$ function as follows. Given a set of literals Φ, let Φ' be the set of literals obtained by closing Φ under negation and consider all possible derivations starting with any subset of Φ'. For each such derivation, look at each branch of its final derivation tree and find the number of Boolean constants required to uniquely encode all case splits along that branch. Let N be the maximum number of required constants along all branches of all derivations and define $\mathcal{L}(\Phi)$ as the union of Φ' and a set of N distinct Boolean constants and their negations. Now, given a final state $M \parallel F$, if we always select from among the Boolean constants in $\mathcal{L}(M)$ when encoding splitting rules, then by construction, there will always be enough constants to finish every branch.

It is trivial to see that properties 1 and 2 of Definition 6 are satisfied by this definition. To ensure that properties 3 and 4 are satisfied, we require that for sets L containing the new Boolean constants, $\mathcal{L}(L)$ is the same as $\mathcal{L}(L')$ where L' is the subset of L obtained by removing any literals containing the new Boolean constants. ∎

Encoding case splits using new Boolean constants works for any rule-based theory solver and shows that it is always possible to integrate these solvers in such a way that the theory solver does not need to do any splitting. In practice, however, it is usually possible and desirable to encode splitting rules more directly. For example, a rule of the form:

$$\frac{\Gamma, \Delta}{\Gamma, \Delta_1 \quad \Gamma, \Delta_2 \quad \cdots \quad \Gamma, \Delta_n},$$

(where $n > 1$) corresponds to the following formula schema: $\neg(\Delta) \vee \bigvee_{i=1}^{n} \Delta_i$. Any instance of this schema can be converted into CNF and the resulting clauses sent to the DPLL engine via Extended T-Learn. [2] For this to work, one additional requirement is that the rules be *refining*. We say that an inference rule is *refining* if it is a refuting rule or if whenever $\Delta, \Delta_1, \ldots, \Delta_n$ are instantiated with $\delta, \delta_1, \ldots, \delta_n$ respectively, $\delta \models_T \gamma_\delta(\bigvee_{i=1}^{n} \delta_i)$. This is essentially a stronger version of soundness. It requires that any model of the premise can be refined into a model of some of the consequents. It is necessary in order to satisfy the side conditions of Extended T-Learn.

---

[2] In practice, if some Δ_i contains more than one literal, it is replaced with a new predicate symbol p_i, and the condition $p_i \leftrightarrow \bigwedge \Delta_i$ is added (this additional condition is easily converted into $m + 1$ clauses where m is the number of literals in Δ_i). This avoids a potential exponential blow-up when the formula is converted into CNF.

We must also check that an appropriate literal-generating function $\mathcal{L}$ exists. Assume we are given a set R of rules and a terminating strategy S. First, define $\mathcal{D}$ to be a function which, given a set Φ of literals returns all literals that may appear along any branch of the derivation tree with any subset of Φ at its root. And let $\mathcal{N}$ be a function which, given a set Φ of literals, returns all literals that can be formed from the atomic formulas in Φ. Now, we define a series of functions $\mathcal{L}_i$ as follows. Let $\mathcal{L}_0$ be the identity function and for $i > 0$, let $\mathcal{L}_i(\Phi) = \mathcal{N}(\mathcal{D}(\mathcal{L}_{i-1}(\Phi)))$. If for some $k > 0$, $\mathcal{L}_k = \mathcal{L}_{k+1}$, then we say that R is *literal-bounded under S*, and define $\mathcal{L} = \mathcal{L}_k$.

Property 15. *If R is a set of sound refining rules for a theory T, S is a strategy for R that is terminating and complete, and R is literal-bounded under S, then R can be integrated with the Extended DPLL Modulo Theories framework.* □

PROOF. We first show that $\mathcal{L}$ satisfies Definition 6. It is easy to see that Properties 1 and 2 in the definition are satisfied. Because $\mathcal{D}(\Phi)$ considers derivations starting with any subset of Φ, Property 3 must also be satisfied. Finally, because $\mathcal{L}$ is a fixed point of $\mathcal{L}_i$, it must be idempotent.

Now, we must show that whenever a state $M \parallel F$ is reached that is final with respect to Basic DPLL Modulo Theories, the theory solver can do one of the following: determine that M is T-consistent; determine that M is T-inconsistent; or introduce a new clause via Extended T-Learn that enables some Basic DPLL Modulo Theories step.

Given a state $M \parallel F$, we simply apply rules from R to M according to strategy S. If $\perp$ is derived, then by soundness, M is T-inconsistent. If an irreducible set of literals is derived, then by completeness, M must be T-consistent. If a splitting rule is reached, and $\Gamma, \Delta, \Delta_1, \ldots, \Delta_n$ are instantiated with $\phi, \delta, \delta_1, \ldots, \delta_n$ respectively, there are three possibilities:

1. For all i, $M \models \neg\delta_i$. In this case, we apply Extended T-Learn to learn $\neg(\delta) \vee \bigvee_{i=1}^{n} \delta_i$, which will result in one or more clauses that are conflicting in M, thus enabling either Fail or T-Backjump by Lemma 11.

2. For some i, $M \models \delta_i$, or else δ_i is undefined in M and $M \models \neg\delta_j$ for every $j \neq i$. In either case, no split is necessary and we simply proceed by applying rules of R to ϕ, δ_i.

3. The final case is when at least two of the δ_i are undefined in M. Then we apply Extended T-Learn to learn $\neg(\delta) \vee \bigvee_{i=1}^{n} \delta_i$ which is guaranteed to contain at least one clause that is not satisfied by M, thus enabling Decide. ∎

Example 16. As we saw in a previous example the theory of arrays requires case splitting. One (sound and refining) rule-based decision procedure for this theory is given in [SDBL01]. A careful examination of the decision procedure reveals the following:

1. Each term can be categorized as an *array* term, an *index* term, a *value* term, or a *set* term.

16

2. No new array terms are ever introduced by the inference rules.

3. At most one new index term for every pair of array terms is introduced.

4. Set terms are made up of some finite number of index terms.

5. The only new value terms introduced are of the form $read(a, i)$ where a is an array term and i is an index term.

It follows that the total number of possible terms that can be generated by the procedure starting with any finite set of literals is finite. Because there are only a finite number of predicates, it then follows that this set of rules is literal-bounded. ∎

Example 17. A rule-based decision procedure for a subset of set theory is given in [CZ00]. As before, the rules given are sound and refining. The rules are carefully constructed so that with the exception of one rule, none of them introduce new terms. The one exception introduces a new constant whenever two *original* constants are disequal to each other, and can be applied at most once for each such pair of original constants. Thus, at most one new constant for every pair of original constants can be introduced. As with the array procedure, it then follows that the total number of possible terms that can be generated starting with any finite set of literals is finite. Because there are only a finite number of predicates, this set of rules is also literal-bounded. ∎

In general, a similar analysis must be done for every theory before it can be integrated with the Extended DPLL Modulo Theories framework as described above. However, it is our experience that solvers for theories of interest are refining and literal-bounded and that this can usually be shown without too much effort.

4 Application to Satisfiability Modulo Multiple Theories

In this section, we focus on background theories T that are actually the union of two or more component theories $T_1, \ldots, T_n$, each equipped with its own solver.

We first show how to obtain an Abstract DPLL Modulo Theories transition system for the combined theory T as a *refinement* of the system XT described in Section 2 using only the solvers of the theories T_i. Then we show, more concretely, how to correspondingly refine the new DPLL(T) architecture into a DPLL($T_1, \ldots, T_n$) architecture in which each T_i-solver is directly integrated into the DPLL($X_1, \ldots, X_n$) engine. We discuss minimal requirements on the solvers that guarantee the total correctness of DPLL($T_1, \ldots, T_n$) computations.

We start with some necessary formal preliminaries. We will work here in the context of first-order logic with equality. For the rest of the section we fix $n > 1$ *stably infinite* theories[3] $T_1, \ldots, T_n$ with respective, mutually disjoint signatures $\Sigma_1, \ldots, \Sigma_n$. We will consider the theory $T = T_1 \cup \cdots \cup T_n$ with signature $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$. We are interested in the

---

[3]A theory T is stably infinite if every T-consistent quantifier-free formula F over T's signature is satisfiable in an infinite model of T.

T-satisfiability of ground formulas over the signature Σ extended with an infinite set K of *free constants*. For any signature Ω we will denote by $\Omega(K)$ the signature $\Omega \cup K$. We say that a ground clause or literal is *(i-)pure* if it has signature $\Sigma_i(K)$ where $i \in \{1, \ldots, n\}$. Given a CNF formula F of signature $\Sigma(K)$, by abstracting subterms with fresh constants from K, it is possible to convert F in linear time into an equisatisfiable CNF formula, all of whose atoms are pure. See [TH96], for instance, for details on this purification procedure. Roughly speaking, though, the idea is to recursively replace an argument t of a function or predicate symbol g in F by a fresh constant $k \in K$ and add to F the unit clause $k = t$ whenever $g \in \Sigma_i$ for some i and the root symbol of t is not in $\Sigma_i(K)$. From now on, we will limit ourselves with no loss of generality to pure formulas.

Following the Nelson-Oppen combination method, the various solvers will cooperate by exchanging entailed equalities over *shared* constants. Here we define more precisely what we mean by "shared." Let L be a set of pure literals over the signature $\Sigma(K)$. We say that a constant $k \in K$ is an *(ij-)shared constant* of L if it occurs in an i-pure and a j-pure literal of L for some distinct i and j. For $i = 1, \ldots, n$, we denote by L^i the set of all the $\Sigma_i(K)$-literals of L and by $\mathcal{S}_i(L)$ the set of all equalities between distinct ij-shared constants of L for every $j \neq i$. Note that for every $j \neq i$, $L^j \cap L^i$ contains at most equalities or the negation of equalities from $\mathcal{S}_i(L)$. An *arrangement for L* is a set containing for each equality $e \in \bigcup_i \mathcal{S}_i(L)$ either e or $\neg e$ (but not both), and nothing else.

The extended Abstract DPLL Modulo theories framework can be refined to take into account that T is a combined theory by imposing the following additional requirements on the XT system.

Refinement 1. We consider only derivations starting with states of the form $\emptyset \parallel F$, where each atom of F is a pure $\Sigma(K)$-atom.

Refinement 2. We consider only applications $M \parallel F \Longrightarrow M\,l \parallel F$ of T-Propagate and applications $M \parallel F \Longrightarrow M \parallel F, C$ of Extended T-Learn where l and each literal of C are pure.

Refinement 1 and 2 maintain the invariant that all the literals occurring in a state are pure, and so can be fed to the corresponding local solvers. Given these minimal requirements, it will be sufficient for T-Propagate to propagate only literals l that are i-pure for some $i = 1, \ldots, n$ and such that and $M^i \models_{T_i} l$, where the entailment $M^i \models_{T_i} l$ is determined by the T_i-solver. Similarly, Extended T-Learn will rely on the local solvers only to learn T_i-lemmas, i.e., i-pure clauses C such that $\emptyset \models_{T_i} \gamma_F(C)$. Note that we do allow lemmas C consisting of pure literals from different theories and such that $F \models_T \gamma_F(C)$, as lemmas of this sort can be computed even if one only has local solvers (consider for example the backjump clauses generated by standard conflict analysis mechanisms).

Refinement 3. The suitable literal-generating function $\mathcal{L}$ maps every finite set L of pure $\Sigma(K)$-literals to a finite set of pure $\Sigma(K)$-literals including $\bigcup_i \mathcal{S}_i(L)$.

To use the various solvers together in a refutationally complete way for T-consistency, it is necessary to make them agree on an arrangement. To do this efficiently, they should

be able to share the entailed (disjunctions) of equalities of shared constants. Refinement 3 then essentially states that theory lemmas can include shared equalities.

4.1 From DPLL(T) to DPLL($T_1, \ldots, T_n$)

Assuming the previous refinements at the abstract level, we now show in concrete how the DPLL(T) architecture can be specialized to use the various local solvers directly and to facilitate cooperation among them.

Except for trivial cases, none of the T_i-solvers are refutationally complete for T-consistency—they are complete at most for their theory T_i. In Section 2, however, we saw that in practice it is not necessary to have a refutationally complete T-solver to effectively generate derivations in XT satisfying Theorem 13. There, the minimal requirement was that, whenever the T-solver cannot determine that a given assignment M is T-consistent it must either (i) identify a T-inconsistent subset of M or (ii) produce a theory lemma containing literals of $\mathcal{L}(M)$ that are undefined in M. Here, we define a similar but local requirement on each T_i that does not even need refutational completeness for T_i-consistency (let alone T-consistency). If M is an assignment consisting of pure literals and $i \in \{1, \ldots, n\}$, we call *(default) completion* of M^i and denote by $\widehat{M^i}$ the smallest extension of M^i falsifying every shared equation for M^i that is undefined in M, that is, $\widehat{M^i} = M^i \cup \{\neg e \mid e \in \mathcal{S}_i(M), e \text{ undefined in } M\}$.

Requirement 1. For each $i = 1, \ldots, n$, the solver for T_i, given a state $M \parallel F$, must be able to do one of the following:

1. determine that $\widehat{M^i}$ is T_i-consistent, or

2. identify a T_i-inconsistent subset of M^i, or

3. produce an i-pure clause C containing at least one literal of $\mathcal{L}(M)$ undefined in M and such that $\emptyset \models_{T_i} \gamma_F(C)$.

The computational cost of the test in Point 1 of this requirement depends on the deduction capabilities of the theory solver.[4] Note, however, that the test can be deferred thanks to Point 3. The solver may choose not to generate the completion of M^i explicitly and test it for T_i-inconsistency, and instead generate a lemma for the engine containing one of the undefined equalities.

If a solver meeting Requirement 1 cannot determine the T_i-consistency of the completion $\widehat{M^i}$, it must be either because it has determined that a subset of $\widehat{M^i}$ (possibly of M^i alone) is in fact inconsistent, or that it needs more information about some of the undefined literals of $\mathcal{L}(M)$ first. However, once every literal of $\mathcal{L}(M)$ is defined in M, including the equalities in $\mathcal{S}_i(M)$, the solver *must* be able to tell whether M^i is T_i-consistent or not. This is a minimal requirement for any solver to be used in a Nelson-Oppen style combination procedure.

---

[4]For some solvers, such as the common ones for EUF or linear rational arithmetic, this additional cost is actually zero as these solvers already explicitly maintain complete information on all the entailed equalities between the known terms.

Usually though it is desirable for Nelson-Oppen solvers to also be able to compute (disjunctions of) shared equalities entailed by a given set of literals, so that only these equalities can be propagated to the other solvers, and guessing is minimized. For instance, if one solver communicates that a is equal to either b or c, then the other solvers do not have to consider cases where a is equal to some fourth constant. Requirement 1 allows that possibility, as illustrated by the following example.

Example 18. Assume, just for simplicity, that for every M, $\mathcal{L}(M)$ is no more than $M \cup \bigcup_i \mathcal{S}_i(M)$, which entails that each T_i-solver is refutationally complete for T_i-consistency. Then consider an assignment M where M_i is T_i-consistent, for some i, and let $e_1, \ldots, e_n$ be equalities in $\mathcal{S}_i(M)$ undefined in M such that $l_1, \ldots, l_m \models \bigvee_k e_k$ for some $\{l_1, \ldots, l_m\} \subseteq M^i$. In this case, $\widehat{M^i}$ is clearly T_i-inconsistent. However, since M^i alone is consistent, by Requirement 1 the T_i-solver must return a lemma containing one or more undefined literals of $\mathcal{L}(M)$.

Now, if the solver can in fact compute (deterministically, with no internal case splits!) the clause $\bigwedge_j l_j \Rightarrow \bigvee_k e_k$, that clause will be the ideal lemma to return. Otherwise, it is enough for the solver to return *any* lemma that contains at least one shared equality e (in the worst case, even a tautology of the form $e \vee \neg e$ will do). Intuitively, this marks a progress in the computation because eventually one of the shared equalities will be added to M (for instance, by an application of Decide), reducing the number of undefined literals in $\mathcal{L}(M)$. ∎

Requirement 1 and the earlier refinements are enough to guarantee that we can use the local T_i-solvers directly—as opposed to building a solver for the combined theory T—to generate derivations satisfying Theorem 13.

The first thing we need for Theorem 13 is that we can always derive from a state $\emptyset \parallel F$ a final state S with respect to Basic DPLL Modulo Theories ($\Longrightarrow_B$). This is indeed possible in the current setting because of the only two basic rules that are theory based, the first, T-Propagate, is never needed to reach a final state—as we saw in Section 2, therefore any restrictions to its application are immaterial; the second, T-Backjump, can always use as backjump clause one constructed directly from the state's assignment and the conflicting clause, without any further involvement of the theory solvers, again as shown in the proof of Lemma 11.

The second thing we need for Theorem 13 is that whenever the final state S has the form $M \parallel F'$, the assignment M is T-consistent. Property 20 below shows that this is indeed possible. Although none of the local solvers is able to determine by itself whether M is T-consistent, it can do that in cooperation with the other solvers, thanks to Requirement 1 and this previously mentioned combination result:

Proposition 19 ([TH96]). *A set L of pure $\Sigma(K)$ literals is T-consistent iff there is an arrangement A for L such that $L^i \cup A$ is T_i-consistent for all $i = 1, \ldots, n$.* □

It is then not difficult to show that, under the assumptions in this section, the following property holds

20

Property 20. *Each derivation of the form $\emptyset \parallel F \Longrightarrow^*_{\text{XT}} M \parallel G$ where $M \parallel G$ is final wrt. Basic DPLL Modulo Theories can be extended in finitely many steps using the solvers for $T_1, \ldots, T_n$ to a derivation of the form $\emptyset \parallel F \Longrightarrow^*_{\text{XT}} M \parallel G \Longrightarrow^*_{\text{XT}} S$ where S is either FailState or a state $M' \parallel G'$ with a T-consistent M'.* $\quad\quad\square$

PROOF. Consider the state $M \parallel G$. If each T_i-solver can determine that the set $\widehat{M^i}$ is T_i-consistent we know that there exists an arrangement A of all the shared constants of M such that $M^i \cup A$ is T_i-consistent for all i. We can then conclude by Proposition 19 that M is T-consistent. Otherwise, if for some i in $\{1, \ldots, n\}$ the solver for T_i cannot determine that $\widehat{M^i}$ is T_i-consistent, one of the cases of Requirement 1 applies. In the first case, the solver identifies a T_i-inconsistent subset $\tilde{M}_i$ of M_i. The clause $\neg \tilde{M}_i$, conflicting in $M \parallel F'$, can be added to that state with an application of Extended T-Learn, as one can easily verify. Since the clause is conflicting, the derivation can be continued with an application of Fail or T-Backjump thanks to Lemma 11. In the second case, the solver produces an i-pure clause C such that $\emptyset \models_{T_i} \gamma_G(C)$ and C contains at least one literal of $\mathcal{L}(M)$ that is undefined in M. This clause too can be added to the state $M \parallel F'$ with an application of Extended T-Learn, as one can again easily verify. Since C contains at least one undefined literal, the derivation can continue in this case as well with the application of a rule of the basic system B. In both cases, the applicability of a rule of B allows the derivation to be continued while satisfying the preconditions of Theorem 4. It follows that repeating the process sketched above will never diverge into an infinite derivation. Eventually then there will be no way to continue the derivation further, which means that one way or another the derivation will reach the state *FailState* or a state of the form $M' \parallel G'$ such that the solver of each T_i is able to say that the set $\widehat{M^i}$ is T_i-consistent, which implies, again by Proposition 19, that M' is T-consistent. $\quad\blacksquare$

4.2 Convex theories

We mentioned in Section 1 that the Nelson-Oppen method is particularly efficient if (i) the component theories T_i are all *convex*, i.e., such that for all sets L of ground $\Sigma_i(K)$-literals and all sets E of ground K-equalities, $L \models_{T_i} \bigvee_{e \in E} e$ iff $L \models_{T_i} e$ for some $e \in E$ and (ii) the T_i solvers are able to compute T_i-entailed shared equalities. In that case, no expensive case splits over shared equalities are needed, and the equality propagation process can be stopped as soon as there are no more entailed shared equalities. This situation can be neatly modeled in our framework if we slightly revise the precondition of the T-Propagate rule on the state $M \parallel F$ to allow the propagation of entailed equalities occurring in $\bigcup_i \mathcal{S}(M)$ in addition to the literals occurring (possibly negated) in F.[5] It is not difficult to see that this extension preserves all the properties of the framework.

When only some of the component theories are convex, an even better extension is to allow the propagation of entailed shared *dis*equalities as well, because this too, even if not strictly needed, may ease the lemma generation burden of the solvers for the non-convex

---

[5]Note that an equality between shared constants need not occur in F.

theories. With these extensions then, we can have solvers for convex theories use only T-Propagate to propagate entailed shared (dis)equalities to the other solvers, and solvers for non-convex theories use a flexible combination of T-Propagate, to propagate single shared (dis)equalities and Extended T-Learn, to propagate proper disjunctions of shared (dis)equalities.

5   Conclusions and further work

We have proposed a new version of DPLL(T) in which theory solvers can delegate all case splits to the DPLL engine. This can be done *on demand* for solvers that can encode their internal case splits into one or more clauses, possibly including new constants and literals. We have formalized this in an extension of Abstract DPLL and proved it correct. We think that the new insights gained by this formalization will help us and others when incorporating these ideas into our respective SMT solvers.

We have also introduced a DPLL($T_1, \ldots, T_n$) architecture for combined theories, which also fits naturally into the extended Abstract DPLL framework. This refinement is crucial in practice because most SMT applications are based on combinations of theories.

Our splitting on demand approach leads to significantly simpler theory solvers. The price to pay is an increase in the complexity of the DPLL(X) engine, which must be able to deal with a dynamically expanding set of literals. However, we believe that doing this once and for all is better than paying the price of building a far more complex theory solver for each theory that requires case splits. Moreover, the requirement of being able to deal dynamically with new literals and clauses is needed in general for flexibility in applications, not just for our approach.

As future work, we plan to evaluate the approach experimentally. We also plan to investigate theory-dependent splitting heuristics, and effective ways for a theory solver to share such heuristic information with the DPLL(X) engine.

Example 21. Consider the conjunction

$$f(g(c_1), g(a)) \neq f(g(c_2), g(b)) \ \land \ a = h(c_1) \ \land \ b = h(c_2) \ \land \ c_1 \leq c_2 \ \land \ c_2 \leq c_1$$

combining EUF and Difference Logic (DL) over the real numbers. This conjunction is T-unsatisfiable, since from the EUF part (the first three equations) one can infer $c_1 \neq c_2$, and, from the remaining (DL) part, $c_1 = c_2$. In an ideal Nelson-Oppen-like combination setting, a solver for the combined theory is obtained by having each T-solver deal each with its part and communicate to each other all inferred equalities between *shared* symbols, in this case, c_1 and c_2. But it is well-known that requiring all theory solvers to generate all equality consequences between the shared symbols is too expensive. Here, it would also suffice, say, to have the EUF solver generate both the equality and the disequality consequences, but, as the example already suggests, generating disequalities from EUF is also expensive. However, cheap tests for generating a relatively small number of *possible* disequality consequences may exist.

Therefore, again, it would be desirable that $Solver_\tau$ can *demand* DPLL(X) to split on literals such as $c_1=c_2$. ∎

References

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

[BB04]   Clark W. Barrett and Sergey Berezin. CVC lite: A new implementation of the cooperating validity checker category b. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04 (Boston, Massachusetts)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer, 2004.

[BBC⁺06]   Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Silvio Ranise, Peter van Rossum, and Roberto Sebastiani. Efficient theory combination via boolean search. *Information and Computation*, x(x):xx–xx, 2006. To appear. Cf. conference paper at CAV'05.

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

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

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

[CZ00]   Domenico. Cantone and Calogero G. Zarba. A new fast tableau-based decision procedure for an unquantified fragment of set theory. In Ricardo Caferra and Gernot Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, volume 1761 of *Lecture Notes in Computer Science*, pages 127–137. Springer, 2000.

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

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

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

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

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

[SDBL01] Aaron Stump, David L. Dill, Clark W. Barrett, and Jeremy Levitt. A decision procedure for an extensional theory of arrays. In *Proceedings of the 16th IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 29–37. IEEE Computer Society, June 2001. Boston, Massachusetts.

[TH96]   Cesare Tinelli and Mehdi T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In *Procs. Frontiers of Combining Systems (FroCoS)*, Applied Logic, pages 103–120. Kluwer Academic Publishers, March 1996.

[TR05]   Cesare Tinelli and Silvio Ranise. SMT-LIB: The Satisfiability Modulo Theories Library, July 2005. http://goedel.cs.uiowa.edu/smtlib/.