Lazy Proofs for DPLL(T)-Based SMT Solvers

1 Introduction

SMT (satisfiability modulo theory) solvers input typically large formulas that contain both Boolean Logic and logic in different theories, and tell you whether the formulas are satisfiable or unsatisfiable.
Verification tools use these solvers to prove system properties. As a result, solver output must be trustable.
However, SMT solvers are really complicated tools that have tens of thousands of lines of code.
One solution to this is to have the SMT solver dispatch a certificate of proof of its result.
For satisfied formulas, this can be a model of satisfaction, that is, values for all the variables in the formula.
For unsatisfied formulas, it is a transformation of the formula into a siimple contradiction using a small set of inference rules, checkable by an external tool such as a proof checker.

Modern SMT solvers have a SAT solver to reason about Boolean logic, and multiple theory solvers for reasoning in each of the theories used.
All these communicate with each other in subtle ways and this must be accounted for while producing proofs.
Proofs must also be fine-grained, that is, they must be detailed enough so they can be checked by simple means.

Contributions made in this paper:

1. They formalize a *generalized* approach for fine-grained proof generation in DPLL(T)-style SMT solvers.

2. They present a lazy approach of proof generation that incurs a low overhead. While deriving a contradiction, the solver generates many lemmas, most of which aren't ultimately used to derive the contradiction. The proof generation only begins after the contradiction is derived, so these unnecessary lemmas don't need to be accounted for.

3. They present a lazy proof generation technique for the specific theories of uninterpreted functions with equality and the extensional theory of arrays.

The technique was implemented in CVC4 and tested on the SMT-Lib benchmark library.

2 DPLL(T)-Based SMT Solvers

SMT is the problem of determining the satisfiability of a set of formulas in some background theory T.
This work focuses on quantifier-free formulas and on DPLL(T) architecture solvers which combine a SAT engine and multiple theory solvers.

2.1 Abstract DPLL(T) Framework

The background theory T consists of m theories $T_1, ..., T_m$ with respective many-sorted signatures $\Sigma_1, .., \Sigma_m$.
A signature is defined as consisting of:

- sort symbols.
- predicate symbols with an associated arity.
- function symbols with an associated arity.

In this setting, all signatures share the set of sort symbols **S**, and equality is the only predicate.
The theories share no function symbols except for a set $C = \cup_{S \in \mathbf{S}} C_S$, where each C_S is a distinguished infinite set of free/uninterpreted constants of sort S. Thus, the only elements that distinguish the signatures are the constant symbols that don't belong to C and the non-nullary function symbols.

DPLL(T) solvers can be formalized abstractly as state transition systems defined by a set of transition rules.
The states of a system are either

- $fail$
- $\langle M, F, C \rangle$ where M is the current context; F is a set of ground clauses representing some form of the input formula; C represents the conflict clause which may be the empty set when there isn't a conflict.

If $M = M_0 \bullet M_1 \bullet ... \bullet M_n$, each M_i is the decision level, and $M^{[i]}$ denotes $M_0 \bullet ... \bullet M_i$.
Initial state : $\langle \phi, F_0, \phi \rangle$, where F_0 is the input formula.
Final state:

- $fail$, when F_0 is unsatisfiable in T.
- $\langle M, F, \Phi \rangle$ where M is satisfiable in T, F is equisatisfiable with F_0 in T, and $M \models_P F$.

Each atom of a clause $F \cup C$ is pure, that is, it has a signature Σ_i for some $i \in 1, ..., m$.
Int_M is the set of all *interface literals* of M: the (dis)equalities between shared constants.
Shared constants are the set represented by
$\{c | constant\ c\ occurs\ in\ Lit_{M|i} and Lit_{M|j}, for some 1 \leq i < j \leq m\}$.
$Lit_{M|i}$ consists of the Σ_i- literals of Lit_M.
In other words, Int_M is the set of (dis)equalities between elements of C.

Transition rules:

- Propagate:
  $$\frac{l_1 \vee ... \vee l_n \vee l \in F \quad \neg l_1, ..., \neg l_n \in M \quad l, \neg l \notin M}{M := Ml} \; Prop$$
  If all but one literal in a clause have been negated in the current context, positively assigned the last literal to satisfy the clause.

- Decide:
  $$\frac{l \in Lit_F \cup Int_M \quad l, \neg l \notin M}{M := M \bullet l} \; Dec$$
  Pick an unassigned literal either from the formula F, or from the interface literals and assign it positively.

- Conflict:
  $$\frac{C = \phi \quad l_1 \vee ... \vee l_n \in F \quad \neg l_1, ..., \neg l_n \in M}{C := \{l_1 \vee ... \vee l_N\}} \; Confl$$
  If there is a clause in F, all of whose literals have been negated by the current context, then that clause is the conflict clause.

- Explain:
  $$\frac{C = \{\neg l \vee D\} \quad l_1 \vee ... \vee l_n \vee l \in F \quad \neg l_1, ..., \neg l_n \prec_M l}{C := \{l_1 \vee ... \vee l_n \vee D\}} \; Expl$$
  If there is a clause in F that contains n literals that were assigned in the context before the literal l was assigned, such that l occurs in the conflict clause, resolve these two clauses to get the new conflict clause.

- Backjump:
  $$\frac{C = \{l_1 \vee ... \vee l_n \vee l\} \quad lev\ \neg l_1, ..., lev\ \neg l_n \leq i < lev\ \neg l}{C := \phi \quad M := M^{[i]} l} \; Backj$$
  If one literal in the conflict clause is in a level strictly above the levels of all other literals in M, then rewind the context to any level in between these two levels.

- Learn:
  $$\frac{C \neq \phi}{F := F \cup C} \; Learn$$
  If there is a conflict clause, it is entailed by input formula, so add it to the set of clauses to satisfy F.

- Fail:
  $$\frac{C \neq \phi \quad \bullet \notin M}{fail}\ Fail$$
  If there is a conflict, and there are no decision points to backjump on, then move to the fail state, that is, the input formula is unsatisfiable.

The rules above model the behavior of the SAT engine, which treats atoms as Boolean variables. The rules that follow model the interaction between the SAT solver and the theory solvers.

- *Propagate$_i$*
  $$\frac{l \in Lit_F \cup Int_M \quad \models_i l_1 \vee ... \vee l_n \vee l \quad \neg l_1, ..., \neg l_n \in M \quad l, \neg l \notin M}{M := Ml}\ Prop_i$$
  If all but one literal of a theory-i lemma are falsified by the current context, and the one literal is unassigned and also belongs to either the literals in F or the interface literals in M, then positively assign the one literal.

- *Conflict$_i$*
  $$\frac{C = \phi \quad \models_i l_1 \vee ... \vee l_n \quad \neg l_1, ..., \neg l_n \in M}{C := \{l_1 \vee ... \vee l_n\}}\ Confl_i$$
  If all the literals of a theory-i lemma are falsified by the current context, then the lemma is a conflict clause.

- *Explain$_i$*
  $$\frac{C = \{\neg l \vee D\} \quad \models_i l_1 \vee ... \vee l_n \vee l \quad \neg\, l_1, ..., \neg\, l_n \prec_M l}{C := \{l_1 \vee ... \vee l_n \vee D\}}\ Expl_i$$
  If one of the literals l in the conflict clause occurs in a theory-i lemma such that the negations of all other literals in that lemma were assigned in the current context before l was assigned positive, then resolve the conflict clause and the theory lemma.

- *Learn$_i$*
  $$\frac{l_1, ..., l_n \in Lit_{M|i} \cup Int_M \cup L_i \quad \models_i \exists \mathbf{x}(l_1[\mathbf{x}] \vee ... \vee l_n[\mathbf{x}])}{F := F \cup \{l_1[\mathbf{c}] \vee ... \vee l_n[\mathbf{c}]\}}\ Learn_i$$
  where $\mathbf{x}$ is a possibly empty tuple of variables, and $\mathbf{c}$ is a tuple of fresh constants from C - the set of shared constants between the sorts - of the same sort as $\mathbf{x}$; L_i is a finite set consisting of literals not present in the original formula F.
  If a theory-i lemma is valid for some tuple $\mathbf{x}$ of variables, instantiate it with a corresponding tuple of constants $\mathbf{c}$, and add it to F.

The rules maintain the invariant that every conflict clause and learned clause is entailed in T by the initial clause set.

2.2 Modeling Solver Behavior

Generally speaking, the system uses the SAT engine - rules *Propagate*, *Decide*, *Conflict*, *Explain*, *Backjump*, *Learn*, and *Fail* - to construct the context M

as a truth assignment for the clauses in F, as if those clauses were propositional. However, it periodically asks the solver of each theory T_i

- to check if the set of Σ_i- constraints in M is unsatisfiable in T_i using the $Conflict_i$ rule, or
- to check if the set of Σ_i- constraints in M entails yet-undetermined literal from $Lit_F \cup Int_M$ using the $Propagate_i$ or the $Explain_i$ rule.

We assume that each T_i- solver provides an $\texttt{explain}_\texttt{i}$ method with the property that if l is a literal propagated by the solver, then $\texttt{explain}_\texttt{i}(l)$ returns a subset $\{\neg l_1, \neg l_2, ..., \neg l_n\}$ of M, such that $\models_i l_1 \vee ... \vee l_n \vee l$. This way, we can use the $Explain_i$ rule on literals that have been propagated by theory lemmas, to rewind theory propagations and get to decisions.
For $i = 1, ..., m$ the set $Lit_{M|i}$ consists of the Σ_i- literals of Lit_M.

The set Lit_F (resp., Lit_M) consist of all literals in F (resp., in M) and their complements. For $i = 1, ..., m$, the set $L_{M|i}$ consists of the Σ_i-literals of Lit_M. Int_M is the set of all *interface literals* of M: the (dis)equalities between *shared constants*, where the set of shared constants is
$\{c \mid \text{constant } c \text{ occurs in } Lit_{M|i}, \text{ and } Lit_{M|j}, \text{ for some } 1 \leq i < j \leq m\}$. The inclusion of Int_M in rules *Decide* and $Propagate_i$ achieves the effect of the Nelson-Oppen theory combination method.

For example, consider the following formula.
F : $1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(2) \wedge x = 1$. The theories involved in this formula are the theory of equality over uninterpreted functions (E) and the theory of linear integer arithmetic (Z). Literals here aren't pure, that is, they belong to more than a single theory. To change that, variables are abstracted:
F : $a \leq x \wedge x \leq b \wedge f(x) \neq f(b) \wedge x = a \wedge a = 1 \wedge b = 2$.
$Lit_F = \{a \leq x,\ x \leq b,\ f(x) \neq f(b),\ x = a,\ a = 1,\ b = 2\}$
$L_{F|E} = \{f(x) \neq f(b),\ x = a\}$
$L_{F|Z} = \{a \leq x,\ x \leq b,\ x = a,\ a = 1,\ b = 2\}$
$Shared\ constants = \{a,\ x,\ b\}$
$Int_F = \{x = a\}$

Rule $Learn_i$ models theory solvers following the splitting-on-demand paradigm. When asked about the satisfiability of the set of Σ_i-literals in M, such solvers may return instead a *splitting lemma* - a clause encoding a guess that needs to be made about those literals before the solver can determine their satisfiability. The set L_i is a finite set consisting of additional literals, that is, not present in the original formula in F, which may be generated by splitting-on-demand theory solvers.
For example, when the array solver is asked whether the following is satisfiable:
$read(write(A, i, v), j) = read(A, j)$
it returns the following splitting lemma:
$read(write(A, i, v), j) \neq read(A, j) \vee i \neq j \vee read(A, i) = v$.

In other words, the equation $read(write(A, i, v), j) = read(A, j)$ holds in 2 situations: when i and j are distinct, or when they are equal but $write(A, i, v)$ changes nothing.

3 Generating Proofs in DPLL(T)

The transition rules defined above are refutation sound: if an execution starting with $\langle \phi, F_0, \phi \rangle$ ends with $fail$, then F_0 is unsatisfiable in T.

3.1 Proof Generation for Propositional Unsatisfiability

A failed execution can be understood as trying to construct a refutation tree: a tree of clauses built from the leaves, which are either clauses in F_0 or learned clauses, down to the root $\bot$, where each non-leaf node is a propositional resolvent of its children. Additionally, one needs to prove that each learned clause is a consequence of the initial clause set.

For example, consider the formula F in CNF expressed as the set:
$\{a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b\}$

M	**F**	**C**	**Rule**	**Step**
	$a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b$	ϕ	Dec	1
$\bullet a$	$a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b$	ϕ	Prop ($a \lor \neg b$)	2
$\bullet a \neg b$	$a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b$	ϕ	Prop ($b \lor c$)	3
$\bullet a \neg b\ c$	$a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b$	ϕ	Confl ($\neg c \lor b$)	4
$\bullet a \neg b\ c$	$a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b$	$\neg c \lor b$	Expl ($b \lor c$)	5
$\bullet a \neg b\ c$	$a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b$	b	Expl ($\neg a \lor \neg b$)	6
$\bullet a \neg b\ c$	$a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b$	$\neg a$	Learn ($\neg a$)	7
$\bullet a \neg b\ c$	$a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b, \neg a$	$\neg a$	Backj ($\neg a$)	8
$\neg a$	$a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b, \neg a$	ϕ	Prop ($a \lor \neg b$)	9
$\neg a \neg b$	$a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b, \neg a$	ϕ	Prop ($b \lor c$)	10
$\neg a \neg b\ c$	$a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b, \neg a$	ϕ	Confl ($\neg c \lor b$)	11
$\neg a \neg b\ c$	$a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b, \neg a$	$\neg c \lor b$	Expl ($b \lor c$)	12
$\neg a \neg b\ c$	$a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b, \neg a$	b	Expl ($a \lor \neg b$)	13
$\neg a \neg b\ c$	$a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b, \neg a$	a	Expl ($\neg a$)	14
$\neg a \neg b\ c$	$a \lor \neg b, \neg a \lor \neg b, b \lor c, \neg c \lor b, \neg a$	$\bot$	Fail	15

The following is a proof tree for the unsatisfiability of F.

$$\dfrac{\dfrac{\dfrac{\neg c \lor b\ (4) \qquad b \lor c\ (5)}{b} \qquad \neg a \lor \neg b\ (6)}{\neg a\ (14)} \qquad \dfrac{\dfrac{\neg c \lor b\ (11) \qquad b \lor c\ (12)}{b} \qquad a \lor \neg b\ (13)}{a}}{\bot}$$

Steps 1 to 7 constitute the left subtree of the proof tree that adds $\neg a$ as a learned clause to F. Steps 8 to 14 constitute the left subtree of the proof tree. Conflict clauses and explanations represent propositional resolution between clauses and the proof tree has a node for every application of the Conflict and the Explain rules, and one at the root for the Fail rule.

3.2 Proof Generation for Unsatisfiability Modulo Theories

The non-propositional rules are can also be seen as trying to construct a refutation tree. However, in addition to clauses from the initial formula, and clauses that are propositionally learned, leaves can have additional clauses that are valid in the particular theory - these are called *theory lemmas*.
The rules $Conf_i$, $Learn_i$, and $Expl_i$ add theory lemmas to the refutation proof tree, while $Prop_i$ uses theory lemmas to propagate literals.
The theory lemmas added to the proof tree must be proven, and this is done by the corresponding theory solver. Each T_i-solver provides a method $\texttt{provideProof}_{\texttt{i}}$ that takes as input a theory lemma and returns a proof of that lemma using theory specific proof rules.