Lazy Proofs for DPLL(T)-Based SMT Solvers

Guy Katz, Clark Barrett
New York University

Cesare Tinelli, Andrew Reynolds
The University of Iowa

Liana Hadarean
Synopsys Inc.

***Abstract*—With the integration of SMT solvers into analysis frameworks aimed at ensuring a system's end-to-end correctness, having a high level of confidence in these solvers' results has become crucial. For unsatisfiable queries, a reasonable approach is to have the solver return an independently checkable proof of unsatisfiability. We propose a lazy, extensible and robust method for enhancing DPLL(*T*)-style SMT solvers with proof-generation capabilities. Our method maintains separate Boolean-level and theory-level proofs, and weaves them together into one coherent artifact. Each theory-specific solver is called upon lazily, a posteriori, to prove precisely those solution steps it is responsible for and that are needed for the final proof. We present an implementation of our technique in the CVC4 SMT solver, capable of producing unsatisfiability proofs for quantifier-free queries involving uninterpreted functions, arrays, bitvectors and combinations thereof. We discuss an evaluation of our tool using industrial benchmarks and benchmarks from the SMT-LIB library, which shows promising results.**

I. Introduction

Many different tools for system analysis and verification exploit the reasoning capabilities of SMT solvers. Typically, these tools dispatch satisfiability queries to an SMT solver and then use the returned results to prove or disprove various system properties. Thus, one's ability to rely on the outcome of the analysis depends on the level of confidence in the results returned by the underlying SMT solver. Unfortunately, obtaining the high level of trust required for, e.g., safety-critical systems can be difficult, as the solvers themselves are highly complex tools and may contain errors.

One reasonable approach to increasing one's level of confidence in an SMT solver's answers is to have it produce solution certificates checkable by simpler, external tools. In the case of a satisfiable (quantifier-free) query, a natural certificate is a *satisfying assignment* for the input formula, which typically can be checked by straightforward means. In the unsatisfiable case, the natural counterpart of a satisfying assignment is a *proof* certificate, which details how to derive a contradiction from the input assertions using a reasonably small set of trusted inference rules. Proof certificates can then be checked by a small trusted proof-checker, thus removing the need to trust the SMT solver.

Proof certificates provide several additional benefits. For instance, they can be used for interpolant generation [23] and certified compilation [11]. Notably, they can be used also to improve the performance of *skeptical proof assistants*. The proof assistant discharges subgoals to the SMT solver and then uses the proof certificates to internally reconstruct a proof [1], [7], [13].

The project or effort depicted was sponsored by the Air Force Research Laboratory (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-13-2-0241 and FA8750-15-C-0113. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFRL or DARPA.

Instrumenting SMT solvers to generate proofs is a complex task. One challenge is that modern solvers reason about their input on multiple levels: typically an underlying SAT engine performs Boolean reasoning, whereas multiple dedicated *theory solvers* (e.g. array, arithmetic, and bitvector solvers) perform theory-specific deductions. The various components interact with each other in subtle ways—the theory solvers interact with the SAT engine and also with each other—and all of these interactions need to be properly captured in the produced proofs. Another challenge is to produce *fine-grained* proofs, i.e., proofs that are sufficiently detailed to be checked by simple means.

In previous work, we presented a proof generation technique for input queries in the logic of quantifier-free fixed-width bitvectors [15]. A main limitation there was that the technique was specifically tailored for that particular logic. In this work we make three major contributions that considerably enhance our previous approach:

1) We present and formalize a general approach for fine-grained proof generation in DPLL(*T*)-style SMT solvers. This approach is not limited to one specific theory (e.g, fixed-width bitvectors); in fact, it even supports proof generation for *combinations of theories*. We explain the approach in terms on an abstract description of DPLL(*T*) and also discuss ways to implement it in practice.
2) We demonstrate how our approach can be realized using *lazy proof generation*, which incurs a lower overhead. During search, an SMT solver will often generate a multitude of lemmas that are not actually needed to derive a contradiction from the input. Our lazy approach postpones proof construction for such lemmas until after the contradiction has been found, and then generates proofs just for those lemmas that were actually used.
3) We present lazy proof generation procedures for the *theory of uninterpreted functions with equality* and the *extensional theory of arrays*.

For evaluation purposes, we implemented our technique in CVC4, a state-of-the-art SMT solver [2]. We conducted extensive experiments using the relevant benchmarks from the SMT-LIB library [4]. Our tool was able to produce proofs in the vast majority of cases.

Before describing our work, we give a high-level description of the DPLL(T) framework for SMT solvers in Section II. Next, in Section III, we explain how proofs of unsatisfiability can be generated in a DPLL(T) setting. In Section IV we discuss our approach to lazy proof production, and in Section V we cover proof production for three theories: uninterpreted functions, arrays, and fixed-width bitvectors. An experimental evaluation of our approach is summarized in Section VI, followed by a discussion of related work in Section VII, and a few concluding remarks in Section VIII.

II. DPLL(T)-Based SMT Solvers

In its most general formulation, 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 SMT solvers based on the DPLL(T) architecture [21], which modularly combines a generic CDCL SAT solver (the *SAT engine*) with one or more reasoners (the *theory solvers*). Each theory solver decides the satisfiability of *constraints* (i.e., conjunctions of ground literals), in a specific background theory. Commonly supported theories include equality over uninterpreted functions (T_{UF}), linear arithmetic over the integers (T_{LIA}) or the reals (T_{LRA}), fixed-width bitvectors (T_{BV}), arrays (T_{AX}), and their combinations.

*Abstract DPLL(*T*) Framework.* We follow a recent abstract formalization of DPLL(T)-style SMT solvers by Reynolds *et al.* [22], which in turn is an elaboration of the one first introduced by Nieuwenhuis *et al.* [21]. We consider a background theory T that is a combination of m theories $T_1, \ldots, T_m$ with respective many-sorted (i.e., typed) signatures $\Sigma_1, \ldots, \Sigma_m$. For convenience, and without loss of generality, we assume that the theories have no predicate symbols besides equality[1] and that they all have the same set $\mathbf{S}$ of sort symbols. We also assume that the theories share no function symbols except for a set $\mathcal{C} = \bigcup_{S \in \mathbf{S}} \mathcal{C}_S$ of constant symbols (functions of arity 0), where each $\mathcal{C}_S$ is a distinguished infinite set of *free* (i.e., uninterpreted) constants of sort S.

DPLL(T) solvers can be formalized abstractly as state transition systems defined by a set of transition rules. The states of the transition system are either the distinguished state fail or triples of the form $\langle M, F, C \rangle$, where

- M, the current *context*, is a sequence of literals and *decision points* $\bullet$,
- F is a set of ground clauses derived from the original input formula, and
- C is either the empty set or a singleton set containing a ground clause, the current *conflict clause*.

Each context M can be factored uniquely into a concatenation of the form $M_0 \bullet M_1 \bullet \cdots \bullet M_n$, where the M_i's contain no decision points. For every $0 \leq i \leq n$ we call M_i the *i'th decision level* of M, and denote with $M^{[i]}$ the subsequence $M_0 \bullet \cdots \bullet M_i$. Each atom of a clause in $F \cup C$ is *pure*, in the sense that it has signature Σ_i for some $i \in \{1, \ldots, m\}$. Note that two atoms in the same clause can have different signatures, and when they do they share at most the constants in $\mathcal{C}$. Input formulas can always be converted to this form while preserving satisfiability in T.

The initial state of the transition system is $\langle \emptyset, F_0, \emptyset \rangle$, where F_0 is a given set of clauses to be checked for satisfiability (i.e., the input formula). The expected final states are either fail, when F_0 is unsatisfiable in T, or $\langle M, F, \emptyset \rangle$ where M is satisfiable in T, F is equisatifiable with F_0 in T, and M propositionally entails F.

The possible behaviors of the system are defined by a set of non-deterministic transition rules that specify a set of successor states for any given state. These rules are depicted in Figure 1 in *guarded assignment form* [17].[2] A rule applies to a state s if all of its premises hold for s.

In the rules, M, F, and C denote, respectively, the context, clause set, and conflict component of the current state. The conclusion describes how each component is changed, if at all. We write $\bar{l}$ to denote the complement of literal l and $l \prec_{\mathsf{M}} l'$ to indicate that l occurs before l' in M. The function lev maps each literal of M to the (unique) decision level in which it occurs. The set Lit_{F} (resp., Lit_{M}) consists of all literals in F (resp., in M) and their complements. For $i = 1, \ldots, m$, the set $\text{Lit}_{\mathsf{M}}|_i$ consists of the Σ_i-literals of Lit_{M}. Int_{M} is the set of all *interface literals* of M: the equalities and disequalities between *shared constants*, where the set of shared constants is $\{c \mid \text{constant } c \text{ occurs in } \text{Lit}_{\mathsf{M}}|_i \text{ and } \text{Lit}_{\mathsf{M}}|_j, \text{ for some } 1 \leq i < j \leq m\}$. The index i for the rules Prop_i, Confl_i, Learn_i, and Expl_i ranges from 1 to m. In those rules, $\models_i$ denotes validity in the theory T_i. Clauses are implicitly processed modulo associativity, commutativity and idempotency of $\vee$.

Modeling Solver Behavior. Rules Dec, Prop, Expl, Confl, Fail, Learn, and Backj model the behavior of the SAT engine, which treats atoms as Boolean variables. In particular, Confl and Expl model the conflict discovery and analysis mechanism used by CDCL SAT solvers [18]. The remaining rules model the interaction between the SAT engine and the individual theory solvers within the overall SMT solver. The rules maintain the invariant that every conflict clause and learned clause is entailed in T by the initial clause set.

Generally speaking, the system uses the SAT engine 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 or entails some yet-undetermined literal from $\text{Lit}_{\mathsf{F}} \cup \text{Int}_{\mathsf{M}}$. In the first case, the theory solver returns an *explanation* of the unsatisfiability as a conflict clause, which is modeled by rule Confl_i. The propagation of entailed theory literals and the extension of the conflict analysis mechanism to them is modeled by rules Prop_i and Expl_i. We assume (as in [21]) that each T_i-solver provides an $\texttt{explain}_i$ method with the property that if l is a

[1] Other predicate symbols can be expressed as function symbols with return sort Bool, interpreted as the Booleans in each theory.

[2] To simplify the presentation, we do not consider here rules that model the forgetting of learned lemmas or restarts of the SMT solver.

$$\mathsf{Dec}\ \frac{l \in \mathrm{Lit}_{\mathsf{F}} \cup \mathrm{Int}_{\mathsf{M}} \quad l, \bar{l} \notin \mathsf{M}}{\mathsf{M} := \mathsf{M} \bullet l} \qquad \mathsf{Confl}\ \frac{\mathsf{C} = \emptyset \quad l_1 \vee \cdots \vee l_n \in \mathsf{F} \quad \bar{l}_1, \ldots, \bar{l}_n \in \mathsf{M}}{\mathsf{C} := \{l_1 \vee \cdots \vee l_n\}} \qquad \mathsf{Fail}\ \frac{\mathsf{C} \neq \emptyset \quad \bullet \notin \mathsf{M}}{\mathsf{fail}}$$

$$\mathsf{Prop}\ \frac{l_1 \vee \cdots \vee l_n \vee l \in \mathsf{F} \quad \bar{l}_1, \ldots, \bar{l}_n \in \mathsf{M} \quad l, \bar{l} \notin \mathsf{M}}{\mathsf{M} := \mathsf{M}\ l} \qquad \mathsf{Backj}\ \frac{\mathsf{C} = \{l_1 \vee \cdots \vee l_n \vee l\} \quad \mathsf{lev}\ \bar{l}_1, \ldots, \mathsf{lev}\ \bar{l}_n \leq\ i < \mathsf{lev}\ \bar{l}}{\mathsf{C} := \emptyset \quad \mathsf{M} := \mathsf{M}^{[i]}\ l}$$

$$\mathsf{Expl}\ \frac{\mathsf{C} = \{\bar{l} \vee D\} \quad l_1 \vee \cdots \vee l_n \vee l \in \mathsf{F} \quad \bar{l}_1, \ldots, \bar{l}_n \prec_{\mathsf{M}} l}{\mathsf{C} := \{l_1 \vee \cdots \vee l_n \vee D\}} \qquad \mathsf{Learn}\ \frac{\mathsf{C} \neq \emptyset}{\mathsf{F} := \mathsf{F} \cup \mathsf{C}}$$

$$\mathsf{Expl}_i\ \frac{\mathsf{C} = \{\bar{l} \vee D\} \quad \models_i l_1 \vee \cdots l_n \vee l \quad \bar{l}_1, \ldots, \bar{l}_n \prec_{\mathsf{M}} l}{\mathsf{C} := \{l_1 \vee \cdots \vee l_n \vee D\}} \qquad \mathsf{Confl}_i\ \frac{\mathsf{C} = \emptyset \quad \models_i l_1 \vee \cdots \vee l_n \quad \bar{l}_1, \ldots, \bar{l}_n \in \mathsf{M}}{\mathsf{C} := \{l_1 \vee \cdots \vee l_n\}}$$

$$\mathsf{Prop}_i\ \frac{\begin{array}{c} l \in \mathrm{Lit}_{\mathsf{F}} \cup \mathrm{Int}_{\mathsf{M}} \\ \models_i l_1 \vee \cdots \vee l_n \vee l \quad \bar{l}_1, \ldots, \bar{l}_n \in \mathsf{M} \quad l, \bar{l} \notin \mathsf{M} \end{array}}{\mathsf{M} := \mathsf{M}\ l} \qquad \mathsf{Learn}_i\ \frac{\begin{array}{c} l_1, \ldots, l_n \in \mathrm{Lit}_{\mathsf{M}}|_i \cup \mathrm{Int}_{\mathsf{M}} \cup L_i \\ \models_i \exists \mathbf{x}\, (l_1[\mathbf{x}] \vee \cdots \vee l_n[\mathbf{x}]) \end{array}}{\mathsf{F} := \mathsf{F} \cup \{l_1[\mathbf{c}] \vee \cdots \vee l_n[\mathbf{c}]\}}$$

Figure 1: State transition rules. In Learn_i, $\mathbf{x}$ is a (possibly empty) tuple of variables; $\mathbf{c}$ is a tuple of fresh constants from $\mathcal{C}$ of the same sort as $\mathbf{x}$.

M	F	C	Rule
	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2$	$\emptyset$	Dec
$\bullet\, 1$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2$	$\emptyset$	Prop $(1 \vee \bar{2})$
$\bullet\, 1\ \bar{2}$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2$	$\emptyset$	Prop $(2 \vee 3)$
$\bullet\, 1\ \bar{2}\ 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2$	$\emptyset$	Confl $(\bar{3} \vee 2)$
$\bullet\, 1\ \bar{2}\ 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2$	$\bar{3} \vee 2$	Expl $(2 \vee 3)$
$\bullet\, 1\ \bar{2}\ 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2$	2	Expl $(\bar{1} \vee \bar{2})$
$\bullet\, 1\ \bar{2}\ 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2$	$\bar{1}$	Learn $(\bar{1})$
$\bullet\, 1\ \bar{2}\ 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	$\bar{1}$	Backj $(\bar{1})$
$\bar{1}$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	$\emptyset$	Prop $(1 \vee \bar{2})$
$\bar{1}\ \bar{2}$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	$\emptyset$	Prop $(2 \vee 3)$
$\bar{1}\ \bar{2}\ 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	$\emptyset$	Confl $(\bar{3} \vee 2)$
$\bar{1}\ \bar{2}\ 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	$\bar{3} \vee 2$	Expl $(2 \vee 3)$
$\bar{1}\ \bar{2}\ 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	2	Expl $(1 \vee \bar{2})$
$\bar{1}\ \bar{2}\ 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	1	Expl $(\bar{1})$
$\bar{1}\ \bar{2}\ 3$	$1 \vee \bar{2},\ \bar{1} \vee \bar{2},\ 2 \vee 3,\ \bar{3} \vee 2,\ \bar{1}$	$\bot$	Fail
	fail		

Figure 2: An execution using only propositional rules.

literal propagated by the solver, then $\texttt{explain}_i(l)$ returns a subset $\{\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_n\}$ of M, such that $\models_i l_1 \vee l_2 \vee \cdots \vee l_n \vee l$. The inclusion of the interface literals $\mathrm{Int}_{\mathsf{M}}$ in rules Dec and Prop_i achieves the effect of the Nelson-Oppen combination method [10], [25]. Rule Learn_i models theory solvers following the splitting-on-demand paradigm [5]. 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 in the rule is a finite set consisting of *additional* literals, i.e., not present in the original formula in F, which may be generated by splitting-on-demand theory solvers.

III. Generating Proofs in DPLL(T)

One can prove that the transition rules defined in Section II are *refutation sound*: if an execution starting with $\langle \emptyset, F_0, \emptyset \rangle$ ends with fail, then F_0 is unsatisfiable in T. We discuss below how to generate unsatisfiability proofs from such executions.

Example 1: Figure 2 shows an example of an execution from an initial state to fail, using only propositional rules. In the figure, we abstract clause atoms by numbers to stress that they are treated purely propositionally by these rules. The *Rule* column shows the rule used for each transition, together with the clause the rule was applied to. We observe that Fail could have been applied right after the second application of Confl; however, we show instead a longer execution that regresses (with Expl) the conflict clause $\bar{3} \vee 2$ to the empty clause $\bot$. As we discuss later, the applications of Expl are needed for proof generation. Note that the second occurrence of $\bar{3} \vee 2$ as a conflict could have been avoided by learning the conflict clause 2 as soon as it was generated. Then, a shorter execution leading to fail would have been possible.

A. *Proof Generation for Propositional Unsatisfiability.*

Given a failed execution from an input set F_0 that uses only propositional clauses, as in Example 1, one can construct a proof that F_0 is (propositionally) unsatisfiable. Intuitively, we can understand a failed execution 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. Thus, a failed execution can be translated into a Boolean resolution proof in a straightforward manner.

Observe, however, that a refutation tree provides only part of the full proof, since it only shows the unsatisfiability of the initial clause set *plus* some set of learned clauses. Thus, to complete the proof one also needs to prove that each learned clause is a consequence of the initial clause set. This can be performed similarly to how conflict analysis is performed in CDCL solvers [16]: every learned clause is the result of an application of the Confl rule and possibly a series of Expl rules. A sequence of resolution applications to the clauses to which these rules were applied produces the learned clause.

Figure 3 depicts a refutation tree for the execution in Figure 2. The tree shows the final resolution proof once all

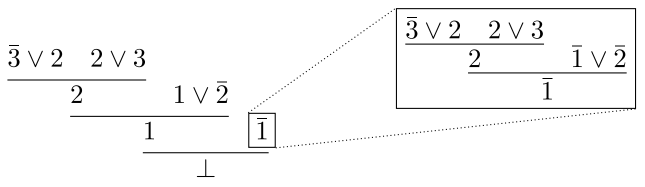

Figure 3: A refutation tree (on the left) with a sub-proof for a learned clause (on the right).

the needed clauses have been learned. Its leaves are the input clauses $\bar{3} \vee 2$, $3 \vee 2$ and $1 \vee \bar{2}$, and the learned clause $\bar{1}$. The tree itself is constructed simply by revisiting the applications of rules Confl and Expl that led to the conflict clause $\bot$, since each application of Expl produces a new conflict clause as the resolvent of the current conflict clause and an initial or learned clause. A separate proof is constructed for the learned clause $\bar{1}$, from the applications of Confl and Expl that generated it. In general, this recursive proof-tree generation process always terminates because each learned clause is derived from initial clauses and previously learned ones. It can be implemented in practice by keeping track of the various applications of Expl.

B. Proof Generation for Unsatisfiability Modulo Theories.

Executions ending in fail that involve the use of the non-propositional transition rules can also be seen as attempts to construct a refutation tree. This time, however, the leaves of the tree can include, in addition to initial and propositionally learned clauses, also *theory lemmas*—a name we give to clauses that come from the Confl_i, Learn_i, and Expl_i rules. Thus, the full proof tree requires combining propositional resolution proofs, produced by the SAT engine, with theory-specific proofs for each theory lemma.

To make this possible, we require each T_i-solver to provide a method $\texttt{provideProof}_i$ that takes as input a theory lemma and returns a proof of that lemma using theory-specific proof rules.[3] Then, a full proof tree can be constructed as before, by visiting the application of rules that led to the final conflict clause $\bot$. When visiting applications of Expl_i, the conflict clause $l_1 \vee \cdots \vee l_n \vee D$ is obtained by resolving $\bar{l} \vee D$ with the theory lemma $E = l_1 \vee \cdots l_n \vee l$. We then call $\texttt{provideProof}_i$ on E to obtain the missing part of the proof. Rule Confl_i adds a conflict clause $C = l_1 \vee \cdots \vee l_n$, which may end up as a leaf in a refutation tree. Thus, C is also a theory lemma and we call $\texttt{provideProof}_i$ on it if we encounter it during proof construction. Finally, rule Learn_i adds the clause $D = l_1[\mathbf{c}] \vee \cdots \vee l_n[\mathbf{c}]$ directly to F, with the consequence that D can act as an input clause. Thus, if we encounter it during proof construction, we call $\texttt{provideProof}_i$ on D to obtain its theory-specific proof.

Thanks to the use of pure literals in clauses and the controlled exchange of information between the various theory solvers through the use of interface literals, Expl_i and $\texttt{provideProof}_i$, which are local to the T_i-theory solver for each i, are enough to construct complex SMT proofs that involve several theories.

[3] We give a few examples of theory-specific proofs for theory lemmas in Section V, when we discuss specific theory solvers.

M	F	C	Rule
	...		
$1\ 2\ 3 \bullet \bar{4} \bullet 5$	F_0	$\emptyset$	Prop_1 $(\bar{1} \vee \bar{2} \vee \bar{5} \vee 6)$
$1\ 2\ 3 \bullet \bar{4} \bullet 5\ 6$	F_0	$\emptyset$	Prop_2 $(\bar{3} \vee \bar{6} \vee 7)$
$1\ 2\ 3 \bullet \bar{4} \bullet 5\ 6\ 7$	F_0	$\emptyset$	Confl_1
$1\ 2\ 3 \bullet \bar{4} \bullet 5\ 6\ 7$	F_0	$4 \vee \bar{6} \vee \bar{7}$	Expl_2 $(\bar{3} \vee \bar{6} \vee 7)$
$1\ 2\ 3 \bullet \bar{4} \bullet 5\ 6\ 7$	F_0	$\bar{3} \vee 4 \vee \bar{6}$	Expl_1 $(\bar{1} \vee \bar{2} \vee \bar{5} \vee 6)$
$1\ 2\ 3 \bullet \bar{4} \bullet 5\ 6\ 7$	F_0	C	Learn
$1\ 2\ 3 \bullet \bar{4} \bullet 5\ 6\ 7$	F_0, C	C	Backj
$1\ 2\ 3 \bullet \bar{4}\ \bar{5}$	F_0, C	$\emptyset$	...

$C = \bar{1} \vee \bar{2} \vee \bar{3} \vee 4 \vee \bar{5}$

Figure 4: An execution using theory rules.

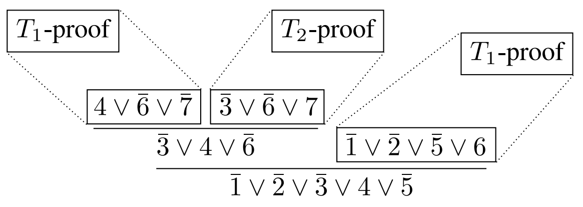
T_1-proof
T_2-proof
T_1-proof
$4 \vee \bar{6} \vee \bar{7}$
$\bar{3} \vee \bar{6} \vee 7$
$\bar{3} \vee 4 \vee \bar{6}$
$\bar{1} \vee \bar{2} \vee \bar{5} \vee 6$
$\bar{1} \vee \bar{2} \vee \bar{3} \vee 4 \vee \bar{5}$

Figure 5: Using theory-specific proofs in proving a lemma.

Example 2: Suppose T is the combination of the theory of uninterpreted functions (T_1 is T_{UF}) and the theory of arrays with extensionality (T_2 is T_{AX}), and consider an initial clause set F_0 containing the atoms:

$$1 : c_3 = f(c_1) \quad 3 : c_5 = (a[c_3] := c_1)[c_4]$$
$$2 : c_4 = f(c_2) \quad 4 : g(c_3, c_5) = g(c_4, c_1)$$

where a is an array, $c_1, \ldots, c_5$ are shared constants, and f and g are uninterpreted functions. The expression $a[i]$ denotes the result of *reading* an array a at index i, and $a[i] := b$ denotes the result of *writing* value b at index i of a. Suppose that literals $1, 2, 3$ occur as unit clauses in F_0 while 4 occurs in some longer clause. Then, a possible execution from F_0 might look like the one in Figure 4 where 5, 6, and 7 are the following interface literals:

$$5 : c_1 = c_2 \qquad 6 : c_3 = c_4 \qquad 7 : c_5 = c_1 \ .$$

If that execution eventually ends in fail and uses the learned clause $C = \bar{1} \vee \bar{2} \vee \bar{3} \vee 4 \vee \bar{5}$, then a proof certificate for F_0 will need a proof of C. The proof tree for C generated from the given execution is shown in Figure 5, with the proofs of the various theory lemmas omitted. Note that C, which has both Σ_1- and Σ_2-literals, is valid in T. However, it is not a lemma of either component theory. Proving it valid in T really requires a collaboration between the two theory solvers.

In practice, concrete implementations of this framework do not pass to the SAT engine the theory lemmas used in Expl_i steps, to avoid polluting the engine with unnecessary clauses. This means that in the example above, for instance, to obtain a proof for the learned clause C, we must be able to reconstruct the theory lemmas used in each Expl_i step. To do this, we record for each learned clause a *proof sketch*: a list of theory propagations, each performed by a specific theory solver, that together justify the learned clause. A clause's proof sketch can be used later to produce a full proof as needed: each individual propagation is converted into a theory lemma via a call to the relevant solver's $\texttt{explain}_i$ method, and then a proof for that propagation is obtained via a call to $\texttt{provideProof}_i$. These intermediate proofs are then composed into a proof for the learned clause, using resolution as in the example above. By keeping these proof sketches we have enough information to construct complete proofs later on. This process facilitates lazy proof generation for learned clauses, as we discuss next.

IV. Lazy Proof Production

In the previous section we saw that in order to produce proofs in a DPLL(T) setting, each T_i-solver must be able to justify the theory lemmas it generates. In this section, we discuss a complementary question: *when* should it provide these justifications?

One approach, found in some solvers that support various forms of proof production [3], [6], is to prove each theory lemma *eagerly*, at the time it is generated. This has the advantage that proof production for each theory step typically incurs only a small overhead, and often boils down to recording the internal deductive process that the theory solver follows when generating the lemma. However, this greedy approach can be inefficient. During the solution phase, theory solvers usually produce numerous lemmas that end up not being used in deriving the empty clause, and so do not make it into the final refutation tree. Hence, any proofs produced for such lemmas are a waste of effort. As an alternative, we advocate a *lazy* approach where no proofs for theory lemmas are generated until the final refutation tree has been found. Then, the $\texttt{provideProof}_i$ methods are invoked only for those theory lemmas that appear as leaves in the tree.

For many of the benchmarks we tried, only a fraction of the thousands of theory lemmas generated during the solving phase are used in the final proof, so the savings from producing proofs for theory lemmas lazily can be significant. A disadvantage is that theory lemmas occurring in the final proof end up being processed twice: once when they are originally generated, and then again when producing the proof. Typically, this means that in addition to generating the proof, the theory solver will have to redo the deductive work that was required to generate each lemma in the first place.

Choosing an appropriate strategy depends on the particular theory solver in question. For some theory solvers reproving lemmas is cheap, making the lazy approach more suitable; for others, an eager approach may yield better results. Our experiments (in Section VI) indicate that, in the cases of T_{UF} and T_{AX}, the lazy approach fairs better. We discuss the particulars of our implementation in Section V.

Lazy Proofs and Rewrite Rules. Modern SMT solvers make use of a large arsenal of rewrite rules aimed at simplifying formulas. These rules specify how and when to replace atoms and terms with simpler but equivalent versions, and applying them can significantly improve the performance of solvers. However, the simplification of even a single atom that appears in a theory lemma can interfere with lazy proof production, as illustrated by the following example, encountered while attempting to produce proofs for the SMT-LIB benchmarks [4] in the theory T_{ABV} combining arrays and bitvectors.

Example 3: Suppose that the T_{AX}-solver generates the theory lemma $L_1 : (b{+}1 = 1) \vee ((a[b{+}1] := x)[1] = a[1])$, where a is an array and b is a fixed-width bitvector (for conciseness, we give here the lemma in non-purified form). Intuitively, this lemma says that if $b + 1 \neq 1$, then writing x to $a[b + 1]$ does not alter the value of $a[1]$. L_1 is valid in T_{AX}, and so the T_{AX}-solver should be able to prove it.

In the lazy approach, the T_{AX}-solver is not asked to provide a proof for L_1 right away. Now, suppose that during subsequent processing of the theory lemma, a bitvector rewrite rule is invoked, simplifying the atom $b + 1 = 1$ to $b = 0$, and consequently transforming lemma L_1 into $L_2 : (b = 0) \vee ((a[b + 1] := x)[1] = a[1])$. This lemma is valid in T_{ABV}, but not in T_{AX}. Thus, when the time comes to produce a proof and the T_{AX}-solver is asked to prove L_2, it will fail to do so.

We can overcome this difficulty as follows. First, we extend the abstract DPLL(T) framework with the following, general rule, which allows theory solvers to rewrite literals:

$$\mathsf{Rewrite}_i \quad \frac{\mathsf{C} = \{l \vee D\} \qquad \models_i \bar{l}_1 \vee \cdots \vee \bar{l}_n \vee (l = l') \quad l_1, \ldots, l_n \in \mathsf{M}}{\mathsf{C} := \{l' \vee D\}}$$

We call the clause $\bar{l}_1 \vee \cdots \vee \bar{l}_n \vee (l = l')$ above a *rewrite lemma*. During the solution phase, we keep track of the application of these rewrite rules to theory atoms. Whenever a theory atom that participates in a lemma is rewritten, we record this information in the lemma's proof sketch. Then, if and when we need to prove the (rewritten) lemma, we can separately prove the original lemma and each specific rewrite lemma used to rewrite it, and then combine their proofs into a proof for the rewritten lemma. In our example above, when we need to prove L_2, we first have the T_{AX}-solver prove the original lemma L_1, and then separately ask the T_{BV}-solver to provide a proof for the equivalence $(b + 1 = 1) = (b = 0)$. These two proofs can then be combined to prove L_2, which is the actual leaf in the refutation tree. Observe that this technique is applicable even if there is a series of rewrites involving multiple theory solvers, because, according to the $\mathsf{Rewrite}_i$ rule, each rewrite lemma used is valid in some individual theory.

Besides enabling proof production when rewrite rules are applied, this process also has a beneficial effect on *modularity*: it separates proofs for rewrite rules from those of the theory lemmas, thus simplifying proof production and improving proof legibility.

V. Theory-Specific Proofs

In the purely propositional case (as in Example 1), a proof can always be constructed that consists of a sequence of applications of Boolean resolution, starting from the input clauses. In the non-propositional case, we saw that each theory solver must provide proofs for its theory lemmas. This requires additional instrumentation in the theory solvers as well as additional deduction rules and axioms beyond Boolean resolution. In this section, we discuss the construction of proof-producing theory solvers for three common theories: uninterpreted functions with equality (T_{UF}), arrays with extensionality (T_{AX}) and fixed-width bitvectors (T_{BV}). In all theory solvers, it is more convenient to prove a theory lemma $l_1 \vee \cdots \vee l_n$ by first proving the unsatisfiability of the set $\{\bar{l}_1, \ldots, \bar{l}_n\}$; so we focus on the latter kind of proof here.

Uninterpreted Functions. A general scheme for a proof-producing T_{UF}-solver was proposed by Fontaine *et al.* [14]. We follow a similar approach, briefly summarized below. Decision procedures for T_{UF} are normally based on congruence closure: the solver maintains an *equality graph* which partitions the terms appearing in the input constraints into equivalence classes. As the search progresses, equivalence classes get merged. Unsatisfiability is derived when two terms a and b from an input constraint $a \neq b$ end up in the same equivalence class.

To produce a refutation tree, the T_{UF}-solver keeps track of all previously performed merges of equivalence classes. When it is asked to prove that $a = b$ is a consequence of some of the input constraints (contradicting the input constraint $a \neq b$), it backtracks through these merges and constructs a chain $a = x_1 = \cdots = x_n = b$, where each link is the result of an input constraint or an application of the congruence rule (deriving, for instance, $f(x) = f(y)$ from $x = y$) [14]. This chain can then be transformed into a proof tree whose leaves are input assertions and whose internal nodes are generated by the application of one of the following rules:

Transitivity:	from $x = y$ and $y = z$ derive $x = z$
Congruence:	from $\mathbf{x} = \mathbf{y}$ derive $f(\mathbf{x}) = f(\mathbf{y})$
Symmetry:	from $x = y$ derive $y = x$

Figure 6 depicts a refutation of the negation of the T_{UF} theory lemma $(x \neq y) \vee (z \neq f(y)) \vee (f(x) = z)$ using those rules.

$$\dfrac{f(x) \neq z \qquad \dfrac{\dfrac{x = y}{f(x) = f(y)}\ \text{Cong.} \qquad \dfrac{z = f(y)}{f(y) = z}\ \text{Symm.}}{f(x) = z}\ \text{Trans.}}{\bot}$$

Figure 6: A refutation of $\{x = y,\, z = f(y),\, f(x) \neq z\}$.

A convenient way to implement eager T_{UF} proof production is to instrument the T_{UF}-solver's `explain` function to produce, apart from an explanation clause, also a proof for that clause. However, T_{UF} is a prime candidate for lazy proof production: since the decision procedure in this case is very efficient, reproving previous lemmas is cheap. In the lazy approach, during proof construction, if we encounter a T_{UF} theory lemma $l_1 \vee \ldots \vee l_n$, we assert its negation to a *fresh* proof-producing instance of the T_{UF}-solver. This solver then constructs the proof as it derives a contradiction. Our experimental evaluation (see Section VI) suggests that the lazy approach is superior to the eager approach for T_{UF}.

Arrays with Extensionality. We now show how we can build on the procedure for T_{UF} to produce proofs for T_{AX}. An efficient decision procedure for T_{AX} [12] uses congruence closure and maintains an equality graph, similarly to the T_{UF} case; however, it merges equivalence classes also as the result of array-specific axioms (proof rules with no premises):

1) Read-over-write 1: for any array a, indices i and j and element x, if $i \neq j$ then $(a[i] := x)[j] = a[j]$.
2) Read-over-write 2: $(a[i] := x)[i] = x$.

The first axiom guarantees that writing to index i does not change the value at a different index j, and the second guarantees that written values persist. A third axiom states that disequal arrays must differ in at least one cell:

3) Extensionality: for any two arrays a and b, if $a \neq b$ then there exists a k such that $a[k] \neq b[k]$.

Observe that, unlike in the T_{UF} case, an unsatisfiable set of constraints here does not have to include one of the form $a \neq b$, since disequalities can also be deduced by the extensionality axiom. A contradiction is reached when two contradictory literal, $a = b$ and $a \neq b$, are derived.

Instrumenting a T_{AX}-solver to produce proof trees based on these axioms again consists of collecting the reasons for the merges of equivalence classes. In particular, any application of Read-over-write 1 and Extensionality contains a sub-proof for the axiom's guard—respectively, $i \neq j$ and $a \neq b$.

Figure 7 depicts a refutation of the negation of the T_{AX} theory lemma $(i = j) \vee ((a[j] := y)[i] \neq x) \vee (a[i] = x)$ using the first read-over-write (*RoW*) axiom.

$$\dfrac{\dfrac{i \neq j \quad (a[j] := y)[i] = x}{a[i] = x}\ \text{RoW 1} \qquad a[i] \neq x}{\bot}$$

Figure 7: Refutation of $\{i \neq j,\, (a[j] := y)[i] = x,\, a[i] \neq x\}$.

Eager proof production can be achieved as in the T_{UF} case. For lazy proof production, we can again instantiate a fresh copy of the solver for every lemma that we need to prove. However, in this case, reproving lemmas from scratch does not suffice. The problem is due to the Extensionality axiom. Consider a case where we need to reprove an instance $(a = b) \vee (a[k] \neq b[k])$ of that axiom, where k is a free constant witnessing the disequality $a \neq b$. If we attempt to lazily prove this lemma by instantiating a fresh T_{AX}-solver and asserting to it the set $\{a \neq b,\, a[k] = b[k]\}$, it will be unable to refute it (simply because, by itself, it is not unsatisfiable). This problem can be overcome by some simple bookkeeping during the solution phase: whenever the Extensionality axiom is used, we record that k is a witness for $a \neq b$; later, during lazy proof production, we ensure that the same k is used to witness $a \neq b$ in the fresh solver. Again, our experiments (see Section VI) suggest that, despite this extra bookkeeping, the lazy approach is superior to the eager approach for T_{AX}.

Bitvectors. We discuss proof generation for the theory T_{BV} of fixed-width bitvectors thoroughly in our previous work [15], so we provide here only a short recap, for completeness. Theory solvers for this theory make extensive use of *bit-blasting*: they transform a bitvector formula φ into an equisatisfiable propositional formula φ^{BB}, in which fresh Boolean variables represent the values of individual bitvector bits. An internal SAT solver then checks the satisfiability of φ^{BB}, and a proof for its unsatisfiability can be translated into a proof for the unsatisfiability of the original φ.

A small example appears in Figure 8. It depicts a bit-blasting refutation for the negation of T_{BV} lemma $(b_1 \neq b_2) \vee (b_2 \neq 10) \vee (b_1 \neq 00)$, where b_1 and b_2 are bitvectors of size 2. The three equalities in the lemma are bit-blasted, via application of the *BB* rule, to derive equalities over some of their constituent bits (denoted here by an array-like notation); these equalities are then used to derive a contradiction.

$$\cfrac{\cfrac{\cfrac{b_1 = b_2}{b_1[1] = b_2[1]}\ \text{BB} \qquad \cfrac{b_2 = 10}{b_2[1] = 1}\ \text{BB}}{b_1[1] = 1}\ \text{Trans.} \qquad \cfrac{b_1 = 00}{b_1[1] = 0}\ \text{BB}}{\bot}$$

Figure 8: A refutation of $\{b_1 = b_2,\, b_2 = 10,\, b_1 = 00\}$.

Bitvector lemmas are proved semi-lazily, in the following sense. During the solution phase, the T_{BV}-solver's internal SAT solver is instrumented to eagerly record any conflict that it discovers. Later, when a lemma needs to be proved because it appears in the refutation tree, the bit-level conflicts that prove it have already been recorded and can be used. While most of the work is thus done eagerly, one part is still performed lazily: the proof of the bit-blasting process itself, i.e., the part of the proof connecting φ to φ^{BB}, is reproduced lazily only for participating lemmas.

Our motivation for eagerly recording the internal SAT solver's conflicts is that reproducing a T_{BV} theory lemma with no information would require re-bit-blasting and re-solving, a potentially very expensive process.

VI. Evaluation

For evaluation purposes we implemented our proof generation approach in CVC4 [2]. Proof generation for T_{BV} was implemented as part of previous work [15]. For this evaluation, we extended CVC4 with both eager and lazy proof generation capabilities for T_{UF} and T_{AX}. We also completed the instrumentation of the DPLL(T) engine as described in Section III, enabling it to handle any combination of the three theories above. Support for proving rewrite rules is still under development, and so for the purposes of this evaluation rewrite rules are treated as axioms, i.e. are given without fine-grained justification. However, the rewrite rules do appear in separate lemmas outside the main proof as discussed in Section IV, and their usage in other parts of the proof is checked for correctness. All changes have been integrated into the master branch of CVC4, which is available online through CVC4's GitHub repository at https://github.com/CVC4.

CVC4 outputs the proofs it generates as terms in the *Logical Framework with Side Conditions* (*LFSC*) [24]. Based on a simply typed λ-calculus with dependent types, LFSC reduces proof checking to type checking: proof rules are encoded as (higher-order) constants, with their premises and conclusions encoded as types, and a proof is a term whose constants are proof-rule names. An LFSC checker takes as input a proof term t and a *signature* S, a collection of type and constant declarations that includes the various proof rules, and checks that t is well-typed with respect to S. We extended the signature S from Hadarean *et al.* [15] to support the T_{UF} and T_{AX} rules mentioned in Section V.

We first compared the lazy and eager proof generation approaches for T_{UF} and T_{AX}. Figure 9 shows the results on all QF_UF and QF_AX benchmarks from the SMT-LIB library [4]. For QF_UF benchmarks, the eager approach was slower than the lazy one on almost all instances and incurred an average performance overhead of 30%. For QF_AX benchmarks, the eager approach was 25% slower on average. Both cases thus indicate a clear advantage for the lazy approach.

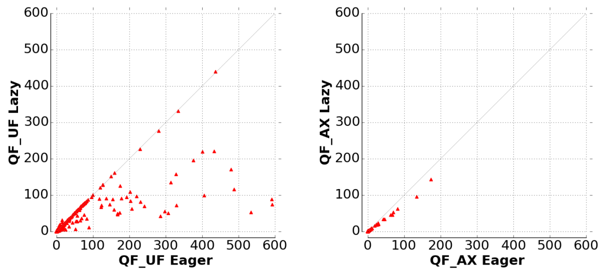

Figure 9: Eager vs. Lazy proof production runtimes, in seconds.

We then ran a more extensive experiment to test our ability to correctly generate and check proofs (lazily for the T_{UF} and T_{AX} solvers) for unsatisfiable benchmarks from all the relevant logics (including theory combinations) in the SMT-LIB library [4]: QF_UF, QF_AX, QF_BV, QF_UFBV, QF_ABV and QF_AUFBV. Table I shows the results. The *Default* columns describe the performance of CVC4 with proof production disabled; the *Generate and Check Proof* and *Generate Proof* columns describe performance when producing a proof with and without checking it, respectively. Also shown in the table are results on a set of industrial QF_ABV benchmarks encoding symbolic execution problems, which were provided to us by collaborators from GrammaTech, Inc. These results appear in the row labeled *Symbolic Execution*.

CVC4 was able to produce proofs for over 99% of all instances that it could solve without proof generation. We were similarly able to check most of the generated proofs using LFSC's external proof checker. In the future, we plan to improve proof checking time by optimizing the LFSC checker and using more efficient LFSC encodings for our proofs.

Benchmark Category	Default		Generate Proof		Generate and Check Proof	
	Solved	Time	Solved	Time	Solved	Time
QF_UF	4083	7523	4067	19097	4029	61650
QF_AX	277	450	264	3170	260	3193
QF_BV	20517	49884	20430	67072	17602	132975
QF_UFBV	12	1391	12	2623	4	170
QF_ABV	4487	16223	4410	19900	4127	22768
QF_AUFBV	31	93	31	245	30	1751
Symbolic Execution	94	1735	89	4364	71	2348

Table I: Producing and checking proofs. All times are in seconds. Experiments were run with a 600 second timeout.

VII. Related Work

Various SMT solvers have taken different approaches to proof production over the years (see Barrett *et al.* [3] for a recent survey). To the best of our knowledge, the only other SMT solver that is both actively maintained and able to produce independently-checkable proofs is veriT [9], which supports eager proof-production for T_{UF} and the theory of linear arithmetic. Our approach for eager proof production in T_{UF} is similar to that of veriT [14]. However, veriT does not support lazy proof production or proofs for T_{AX} or T_{BV}.

The Z3 solver produces *proof traces*, essentially a record of propositional inferences plus a listing of theory lemmas used [6]. Extending such a proof trace to a full proof requires an external tool capable of proving theory lemmas independently, which can be quite challenging, for instance for bitvector theory lemmas [8]. Our approach differs from Z3's approach in that it produces full, fine-grained proofs that are checkable by simple checkers.

The LFSC format [24] allows us to use a generic LFSC checker to check proofs. Other approaches for checking SMT-generated proofs include using custom checkers [20] or skeptical interactive theorem provers such as HOL Light [19] or Isabelle/HOL [14].

VIII. Conclusion and Future Work

Adding proof production capabilities to complex tools like SMT solvers can greatly increase our level of confidence in their results. We presented here a technique that allows DPLL(T)-style SMT solvers to produce unsatisfiability proofs for queries involving combinations of theories. Our approach requires that each theory solver provide proofs for its theory-specific deductions; and these sub-proofs are then interwoven into a complete, cohesive proof by the main SAT engine. Our approach is modular and extensible in the sense that any new proof-producing solver can be readily integrated with existing ones. We also explored *lazy* proof generation and demonstrated its advantages for T_{UF} and T_{AX}.

For the near future, we plan to improve CVC4's ability to prove rewrite steps, as discussed in Section IV. Another planned enhancement is the addition of proof support for arithmetic and *quantified logics*—with the aim of eventually being able to produce proofs for unsatisfiable formulas in the full input language supported by CVC4.

References

[1] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *CPP*, 2011.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, 2011.

[3] C. Barrett, L. de Moura, and P. Fontaine. Proofs in Satisfiability Modulo Theories. *All about Proofs, Proofs for All*, 2015.

[4] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). http://www.SMT-LIB.org.

[5] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting On Demand in SAT Modulo Theories. In *LPAR*, 2006.

[6] N. Bjørner and L. de Moura. Proofs and Refutations, and Z3. In *LPAR*, 2008.

[7] J. Blanchette, S. Böhme, and L. Paulson. Extending Sledgehammer with SMT Solvers. *J. of Automated Reasoning*, 2013.

[8] S. Böhme, A. Fox, T. Sewell, and T. Weber. Reconstruction of Z3's Bit-Vector Proofs in HOL4 and Isabelle/HOL. In *CPP*, 2011.

[9] T. Bouton, D. Caminha B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an Open, Trustable and Efficient SMT-Solver. In *CADE*, 2009.

[10] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, and R. Sebastiani. Delayed Theory Combination vs. Nelson-Oppen for Satisfiability Modulo Theories: a Comparative Analysis. *Annals of Mathematics and Artificial Intelligence*, 2009.

[11] J. Chen, R. Chugh, and N. Swamy. Type-Preserving Compilation of End-to-End Verification of Security Enforcement. In *PLDI*, 2010.

[12] L. de Moura and N. Bjørner. Generalized, Efficient Array Decision Procedures. In *FMCAD*, 2009.

[13] B. Ekici, G. Katz, C. Keller, A. Mebsout, A. Reynolds, and C. Tinelli. Extending SMTCoq, a Certified Checker for SMT. In *HATT*, 2016.

[14] P. Fontaine, J. Marion, S. Merz, L. Nieto, and A. Tiu. Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. In *TACAS*, 2006.

[15] L. Hadarean, C. Barrett, A. Reynolds, C. Tinelli, and M. Deters. Fine-grained SMT Proofs for the Theory of Fixed-width Bit-vectors. In *LPAR*, 2015.

[16] M. Heule and A. Biere. Proofs for Satisfiability Problems. *All about Proofs, Proofs for All*, 2015.

[17] S. Krstić and A. Goel. Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In *FROCOS*, 2007.

[18] J. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 1999.

[19] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite. In *PDPAR*, 2005.

[20] M. Moskal. Rocket-Fast Proof Checking for SMT Solvers. In *TACAS*, 2008.

[21] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *J. of the ACM*, 2006.

[22] A. Reynolds, C. Tinelli, A. Goel, and S. Krstić. Finite Model Finding in SMT. In *CAV*, 2013.

[23] A. Reynolds, C. Tinelli, and L. Hadarean. Certified Interpolant Generation for EUF. In *SMT*, 2011.

[24] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. SMT Proof Checking Using a Logical Framework. *Formal Methods in System Design*, 2012.

[25] C. Tinelli and M. Harandi. A New Correctness Proof of the Nelson-Oppen Combination Procedure. In *FROCOS*, 1996.