AFRL-RI-RS-TR-2017-007

CERTIFIED SATISFIABILITY MODULO THEORIES (SMT) SOLVING FOR SYSTEM VERIFICATION

NEW YORK UNIVERSITY

JANUARY 2017

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2017-007 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
STEVEN L. DRAGER
Work Unit Manager

/ S /
JOHN D. MATYJAS
Technical Advisor, Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JAN 2017	FINAL TECHNICAL REPORT	AUG 2013 – JUL 2016

4. TITLE AND SUBTITLE

CERTIFIED SATISFIABILITY MODULO THEORIES (SMT) SOLVING FOR SYSTEM VERIFICATION

5a. CONTRACT NUMBER
FA8750-13-2-0241

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)
Clark Barrett, Burak Ekici Liana Hadarean, Guy Katz, Chantel Keller, Alain Mebsout, Andrew Reynolds, Cesare Tinelli

5d. PROJECT NUMBER
HACM

5e. TASK NUMBER
SC

5f. WORK UNIT NUMBER
SV

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
New York University
70 Washington Square S
New York, NY 10012-1019

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
675 North Randolph Street
Arlington, VA 22203-2114

AFRL/RITA
525 Brooks Rd.
Rome, NY 13441

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2017-007

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Modern society relies increasingly on software. Many software systems, however, are unacceptably unreliable as they often contain conceptual or implementation errors and are therefore vulnerable to security attacks. It is now widely recognized that dramatically improving the reliability of computer software is going to be one of the most important scientific and technological challenges of this century. In model-based development, software systems, in particular embedded ones, are developed by first constructing a mathematical model of the system; then verifying desired functional properties against the model; and finally implementing the model. Increasingly, the property-checking phase can be handled formally and automatically using model-checking and verification techniques that rely on automated reasoning engines. Despite the success of these techniques, the complexity of the verification tools involved makes their trustworthiness an important issue. Incorrect results from the automated reasoning engines may compromise the whole verification process. In addition, even if the trustworthiness of a particular reasoning engine can be assured, a large verification task may require multiple reasoners to work together. Thus, the compositionality of trustworthiness is also a critical capability: tools must be able to trust and use the results of other tools. One approach for ensuring trustworthy results from a complex reasoning engine, and for supporting compositionality, is to have the engine emit an independently checkable proof. Compositionality can then be facilitated by using a proof format that can easily be processed by other verification tools. This report describes the results of efforts to do exactly this. CVC4, a modern, open-source solver for Satisfiability Modulo Theories (SMT), has been instrumented with the ability to generate independently checkable proofs for any verification condition it is able to prove. Additionally, a translator has been implemented to take proofs produced by CVC4 into Coq, an interactive theorem prover often used in large verification projects.

15. SUBJECT TERMS

Model-based development of software systems, model checking, Satisfiability Modulo Theories, CVC4

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**STEVEN L. DRAGER**
U	U	U	SAR	72	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

Table of Figures

Table of Tables

1 Summary

Modern society relies increasingly on software. Many software systems, however, are unacceptably unreliable. Software often contains conceptual or implementation errors and is vulnerable to security attacks. It is now widely recognized that dramatically improving the reliability of computer software is going to be one of the most important scientific and technological challenges of this century.

In model-based development, software systems, in particular embedded ones, are developed by first constructing a mathematical model of the system; then verifying desired functional properties against the model; and finally implementing the model. Increasingly, the property-checking phase can be handled formally and automatically using model-checking and verification techniques that rely on automated reasoning engines.

Despite the success of these techniques, the complexity of the verification tools involved makes their trustworthiness an important issue. Incorrect results from the automated reasoning engines may compromise the whole verification process. In addition, even if the trustworthiness of a particular reasoning engine can be assured, a large verification task may require multiple reasoners to work together. Thus, the compositionality of trustworthiness is also a critical capability: tools must be able to trust and use the results of other tools.

One approach for ensuring trustworthy results from a complex reasoning engine, and for supporting compositionality, is to have the engine emit an independently checkable proof. Compositionality can then be facilitated by using a proof format that can easily be processed by other verification tools.

This report describes the results of our efforts to do exactly this. We have instrumented CVC4, a modern, open-source solver for Satisfiability Modulo Theories (SMT) with the ability to generate *independently checkable* proofs for any verification condition it can prove. We have also implemented a translator from proofs produced by CVC4 to Coq, an interactive theorem prover often used in large verification projects.

2 Introduction

Many different tools for system analysis and verification exploit the reasoning capabilities of Satisfiability Modulo Theories (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 [1] and certified compilation [2]. 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 [3]–[5].

To illustrate this, we have integrated the CVC4 SMT solver with the Coq proof assistant. We have built on a pre-existing, third-party tool called SMTCoq and extended it for our purpose with the collaboration of one of the original SMTCoq developers (co-author Keller). SMTCoq is a communication tool between the Coq proof assistant and external SAT and SMT solvers. Based on a checker for generic first-order certificates implemented and proved correct in Coq, SMTCoq offers facilities both to check external SAT and SMT answers and to improve Coq 's automation using such solvers, in a safe way. While it originally supported only the SAT solver ZChaff and the SMT solver veriT for a combination of the theories of uninterpreted function symbols and linear integer arithmetic, SMTCoq was meant to be extendable to other solvers and theories with a reasonable amount of effort. Here we present our extensions to support CVC4 together with the theories of bit vectors and functional arrays.

The report is organized as follows. In Chapter 3, we describe the theory and overall approach for instrumenting an SMT solver to produce proofs. Next, in Chapter 4, we describe the results of the project which include (i) proof-producing solvers for three specific theories: equality with uninterpreted functions (EUF), arrays, and bit-vectors; and (ii) the SMTCoq tool that translates proofs produced by the SMT solver into theorems in the Coq proof assistant. We also report on an empirical evaluation and discuss related work. Chapter 5 concludes.

3 Methods, Assumptions, and Procedures

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 bit-vector 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 this chapter, we describe our approach to instrumenting SMT solvers to produce proofs. We have made three major contributions to the state of the art:

1. We have developed a general approach for fine-grained proof generation in DPLL($\mathcal{T}$)-style SMT solvers. This approach is not limited to one specific theory (e.g., fixed-width bit-vectors); in fact, it even supports proof generation for *combinations of theories*. We explain the approach in terms on an abstract description of DPLL($\mathcal{T}$) 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.

We start with a high-level description of the DPLL($\mathcal{T}$) framework for SMT solvers in Section 3.1. Next, in Section 3.2, we explain how proofs of unsatisfiability can be generated in a DPLL($\mathcal{T}$) setting. In Section 3.3 we discuss our approach to lazy proof production.

3.1 DPLL($\mathcal{T}$) -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($\mathcal{T}$) architecture [6], 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($\mathcal{T}$) Framework. We follow a recent abstract formalization of DPLL($\mathcal{T}$) -style SMT solvers by Reynolds *et al.* [7], which in turn is an elaboration of the one first introduced by Nieuwenhuis *et al.* [6]. 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 **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($\mathcal{T}$) 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* [8].[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 $\mathrm{Lit}_{\mathsf{F}}$ (resp., $\mathrm{Lit}_{\mathsf{M}}$) consists of all literals in F (resp., in M) and their complements. For $i = 1, \ldots, m$, the set $\mathrm{Lit}_{\mathsf{M}}|_i$ consists of the Σ_i-literals of $\mathrm{Lit}_{\mathsf{M}}$. $\mathrm{Int}_{\mathsf{M}}$ is the set of all *interface literals* of M: the equalities and disequalities between *shared constants*, where the set of shared constants is $\{c \mid \text{constant } c \text{ occurs in } \mathrm{Lit}_{\mathsf{M}}|_i \text{ and } \mathrm{Lit}_{\mathsf{M}}|_j, \text{ for some } 1 \leq i < j \leq m\}$. The index i for the rules Prop_i, Confl_i, Learn_i, and Expl_i ranges from 1 to m. In those rules, $\models_i$

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

denotes validity in the theory T_i. Clauses are implicitly processed modulo associativity, commutativity and idempotency of ∨.

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

$$\text{Fail}\ \frac{\text{C} \neq \emptyset \quad \bullet \notin \text{M}}{\text{fail}} \qquad \text{Prop}\ \frac{l_1 \vee \cdots \vee l_n \vee l \in \text{F} \quad \bar{l}_1, \ldots, \bar{l}_n \in \text{M} \quad l, \bar{l} \notin \text{M}}{\text{M} := \text{M}\ l}$$

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

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

$$\text{Expl}_i\ \frac{\text{C} = \{\bar{l} \vee D\} \quad \vDash_i l_1 \vee \cdots l_n \vee l \quad \bar{l}_1, \ldots, \bar{l}_n \prec_\text{M} l}{\text{C} := \{l_1 \vee \cdots \vee l_n \vee D\}}$$

$$\text{Confl}_i\ \frac{\text{C} = \emptyset \quad \vDash_i l_1 \vee \cdots \vee l_n \quad \bar{l}_1, \ldots, \bar{l}_n \in \text{M}}{\text{C} := \{l_1 \vee \cdots \vee l_n\}}$$

$$\text{Prop}_i\ \frac{l \in \text{Lit}_\text{F} \cup \text{Int}_\text{M} \quad \vDash_i l_1 \vee \cdots \vee l_n \vee l \quad \bar{l}_1, \ldots, \bar{l}_n \in \text{M} \quad l, \bar{l} \notin \text{M}}{\text{M} := \text{M}\ l}$$

$$\text{Learn}_i\ \frac{l_1, \ldots, l_n \in \text{Lit}_\text{M}|_i \cup \text{Int}_\text{M} \cup L_i \quad \vDash_i \exists \mathbf{x}\,(l_1[\mathbf{x}] \vee \cdots \vee l_n[\mathbf{x}])}{\text{F} := \text{F} \cup \{l_1[\mathbf{c}] \vee \cdots \vee l_n[\mathbf{c}]\}}$$

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

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

Modeling Solver Behavior. Rules c , 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 [9]. 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 $\text{Lit}_\text{F} \cup \text{Int}_\text{M}$. 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 [6]) that each T_i-solver provides an $\texttt{explain}_i$ method with the property that if l is a literal propagated by the solver, then $\texttt{explain}_i(l)$ returns a subset $\{\bar{l}_1, \bar{l}_2, \dots, \bar{l}_n\}$ of M, such that $\models_i l_1 \vee l_2 \vee \cdots \vee l_n \vee l$. The inclusion of the interface literals Int_M in rules Dec and Prop_i achieves the effect of the Nelson-Oppen combination method [10], [11]. Rule Learn_i models theory solvers following the splitting-on-demand paradigm [12]. 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.

3.2 Generating Proofs in DPLL($\mathcal{T}$)

One can prove that the transition rules defined in Section 3.1 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.

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

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.*

3.2.1 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 [13]: 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.

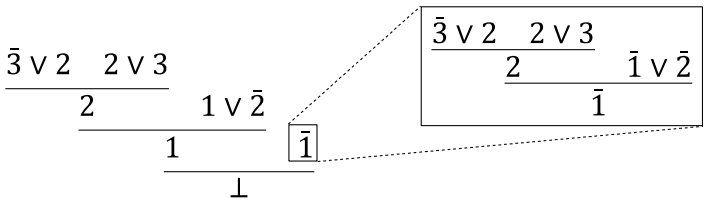

Figure 3 A refutaion tree (on the left) with a sub-proof for a learned clause (on the right)

Figure 3 depicts a refutation tree for the execution in Figure 2. The tree shows the final resolution proof once all the needed clauses have been learned. Its leaves are the input clauses $\overline{3} \vee 2$, $3 \vee 2$ and $1 \vee \overline{2}$, and the learned clause $\overline{1}$. 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 $\overline{1}$, 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 .

3.2.2 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 $\overline{l} \vee D$ 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,

[3] We give a few examples of theory-specific proofs for theory lemmas in Section 4.1, when we discuss specific theory solvers.

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.

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:*

$$\begin{array}{llll} 1: & c_3 = f(c_1) & 3: & c_5 = (a[c_3] := c_1)[c_4] \\ 2: & c_4 = f(c_2) & 4: & g(c_3, c_5) = g(c_4, c_1) \end{array}$$

where a is an array, $c_1, \dots, c_5$ 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:

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

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.

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

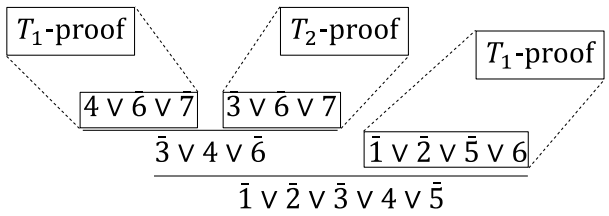

Figure 5 Using theory-specific proof in proving a lemma

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.

3.3 Lazy Proof Production

In the previous section we saw that in order to produce proofs in a DPLL($\mathcal{T}$) 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 [14], [15], 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 4.3) indicate that, in the cases of T_{UF} and T_{AX}, the lazy approach fairs better. We discuss some of the particulars of our implementation in Section 4.1.

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 [16] 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($\mathcal{T}$) framework with the following, general rule, which allows theory solvers to rewrite literals:

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

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.

3.4 LFSC

LFSC is an extension of the Edinburgh Logical Framework (LF) [17], a meta-framework based on an extension of simply-typed lambda calculus with dependent types. LF has been used extensively to encode various kinds of deductive systems. In general, a specific proof system P can be defined in LF by representing its proof rules as LF constants and encoding their premises and conclusions as a type. In this setting, a formal proof in the encoded proof system is represented as an LF term whose constants (in the sense of higher-order logic) are proof-rule names. A collection of type and term constant declarations is called a *signature* in LF. Checking the correctness of a proof then reduces to type checking: an LF proof checker takes as input both a signature S defining a proof system P and a proof term t encoding a proof in P. It verifies the correctness of the proof by checking that t is well-typed with respect to S. For example, the equality transitivity proof rule:

$$\frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}\ \mathsf{trans} \qquad (3.1)$$

in (unsorted) first-order logic can be encoded in LF as a constant with type:

$$\mathsf{trans} \quad : \Pi t_1, t_2, t_3 \colon \mathsf{tr}.\ \Pi p_1 \colon \mathsf{holds}\,(\mathsf{eq}\ t_1\ t_2).\ \Pi p_2 \colon \mathsf{holds}\,(\mathsf{eq}\ t_2\ t_3).\ \mathsf{holds}\,(\mathsf{eq}\ t_1\ t_3) \qquad (3.2)$$

where Π is the binder for the dependently typed product, tr is the type of first-order terms, eq is a binary function of type $\mathsf{tr} \times \mathsf{tr} \to \mathsf{form}$ (where form is the type of first-order formulas), and holds is a unary (dependent) type parametrized by a first-order formula.[4] As a proof constructor, the proof rule (3.1) takes as arguments terms t_1, t_2 and t_3, as well as proofs p_1 of $t_1 = t_2$ and p_2 of $t_2 = t_3$, and returns a proof of $t_1 = t_3$. The LF declaration in (3.2) encodes this in the type of the constant trans. One possible proof that $a = d$ follows from the premises $a = b$, $b = c$, and $c = d$ is represented by the (well-typed) term:

$$\lambda a, b, c, d \colon \mathsf{term}.\ \lambda p_1 \colon \mathsf{holds}\,(\mathsf{eq}\ a\ b).\ \lambda p_2 \colon \mathsf{holds}\,(\mathsf{eq}\ b\ c).\ \lambda p_3 \colon \mathsf{holds}\,(\mathsf{eq}\ c\ d).$$
$$(\mathsf{trans}\ a\ c\ d\ (\mathsf{trans}\ a\ b\ c\ p_1\ p_2)\ p_3)$$

[4] Intuitively, an LF expression of dependent type $\Pi\varphi \colon \mathsf{form}.\ \mathsf{holds}(\varphi)$ represents a proof that the formula φ holds.

Using the wild-card symbol _, the body of the innermost lambda term can be simplified to (trans _ _ _ (trans _ _ _ p_1 p_2) p_3), since the omitted arguments can be inferred automatically during type-checking.

Purely declarative proof systems like those defined in LF cannot always efficiently model the kind of complex reasoning usually employed by SMT solvers. LFSC addresses this issue by extending LF types with computational *side conditions*, explicit computational checks defined as programs in a small but expressive functional first-order programming language. The language has built-in types for arbitrary precision integers and rationals, ML-style pattern matching over LFSC type constructors, recursion, limited support for exceptions, and a very restricted set of imperative features. A proof rule in LFSC may optionally include a side condition written in this language. When checking the application of such a proof rule, an LFSC checker computes actual parameters for the side condition and executes its code. If the side condition fails, the LFSC checker rejects the rule application.

As shown in Figure 7, when using LFSC , the trusted core includes both the (generic) LFSC checker and the specific LFSC signature which consists of a set of proof rules, each of which may have side conditions.

We refer the reader to [18] for a detailed description of the LFSC language and its formal semantics. Here we introduce LFSC syntax via examples to illustrate the main features of the framework.

Example 4. *An inference rule at the heart of SAT and SMT solvers is the propositional resolution rule:*

$$\frac{l_1 \vee \ldots l_n \vee l \quad \neg l \vee l'_1 \vee \ldots l'_m}{l_1 \vee \ldots l_n \vee l'_1 \vee \ldots \vee l'_m} \text{ Res}$$

where l's are literals. This rule alone is actually not enough to express resolution derivations as formal objects, since one also has to account for the associativity, commutativity and idempotency of the ∨ *operator. In LF, this problem can be addressed only by adding additional proof rules for those properties. Doing so makes it possible to move literals around in a clause and remove duplicate literals, but at the cost of requiring many proof rules for each resolution step, resulting in the generation of very large proofs. Alternative solutions [19] eschew the generic, declarative approach provided by meta-frameworks like LF and instead hard-code the clause data structure in the proof checker, requiring a proof-checker with higher complexity and lower generality.*

unit, var, lit, clause: type holds: clause → type cln: clause
ok: unit pos, neg: var → lit clc: lit → clause → clause

resolve (c_1, c_2: clause, v: var): clause = let p (pos v) in let n (neg v) in
  let _ (occurs p c_1) in let _ (occurs n c_2) in merge (remove p c_1) (remove n c_2)
Res: $\Pi c, c_1, c_2$: clause. holds c_1 → holds c_2 → Πv: var {(resolve c_1 c_2 v) ↓ c}. holds c

Figure 6 LFSC declarations encoding propositional resolution.

In contrast, an LFSC proof rule for resolution can use a side condition to encode that the resulting clause is computed by removing the complementary literals in the two input clauses and then merging the remaining literals. One encoding of the rule and its side condition, together with all the necessary types and constants, is shown in Figure 6*. In the figure and in the remainder of the chapter, we write* $\tau_1 \to \tau_2$ *to abbreviate as usual a type of the form* $\Pi x{:}\,\tau_1.\ \tau_2$ *where* τ_2 *contains no occurrences of* x*. Clauses are encoded essentially as nil-terminated lists of literals. They are built with the constructors* cln*, for the empty clause, and* clc*, for non-empty clauses. Literals are built from propositional variables using the constructors* pos *and* neg*, for positive and negative literals. Variables do not have constructors because LFSC variables can be used directly.*

The resolution rule Res *takes as input the clauses* c_1*,* c_2*, and* c*, together with a proof of* c_1 *of type* $holds\ c_1$*, one of* c_2 *of type* $holds\ c_2$*, and a variable* v *to be used as the resolved atom. The* $resolve$ *side condition function computes the resolvent of clause* c_1 *with* c_2*, provided that* c_1 *contains at least one occurrence of the positive literal* $(pos\ v)$ *and* c_2 *contains at least one occurrence of the negative literal* $(\text{neg}\ v)$*. The side condition* $\{(\text{resolve}\ c_1\ c_2\ v) \downarrow\ c\}$ *succeeds if* c *is the result of resolving* c_1 *and* c_2 *on* v*. In that case, the proof rule returns a proof of* c*. The definitions of the auxiliary functions* $occurs$*,* $remove$*, and* $merge$ *are omitted from* Figure 6 *due to space constraints.* $(\text{occurs}\ l\ c)$ *does nothing if the literal* l *is in the clause* c*; otherwise, it raises a failure exception;* $(\text{remove}\ l\ c)$ *returns the result of removing the literal* l *from the clause* c*;* $(\text{merge}\ c_1\ c_2)$ *returns the clause with no repeated literals resulting from merging clauses* c_1 *and* c_2*.*

LFSC has previously been successfully used to encode the constructs necessary for Boolean resolution, CNF conversion, and propositional abstraction of theory lemmas [18]. In this chapter, we did not cover these constructs, but instead focused on how to encode theory specific reasoning in LFSC .

4 Results and Discussion

In this chapter, we describe the results of the project. We begin with a description of the proof systems developed for specific theories. We then describe our implementation in CVC4, a state-of-the-art SMT solver [20]. We conducted extensive experiments using the relevant benchmarks from the SMT-LIB library [16]. Our tool was able to produce proofs in the vast majority of cases. We conclude with a description of the SMTCoq tool which uses proofs produced by CVC4 to produce proofs within the Coq proof assistant.

4.1 Proof Systems for SMT Theories

Recall that 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.

More generally, SMT proofs typically have a three-tiered structure: (i) a derivation of the internal CNF formula ψ from the input formula φ;[5] (ii) a resolution refutation of ψ in the form of a resolution tree whose root is the empty clause and whose leaves are either clauses from ψ or theory lemmas; and (iii) theory proofs of all the theory lemmas occurring in the resolution tree.

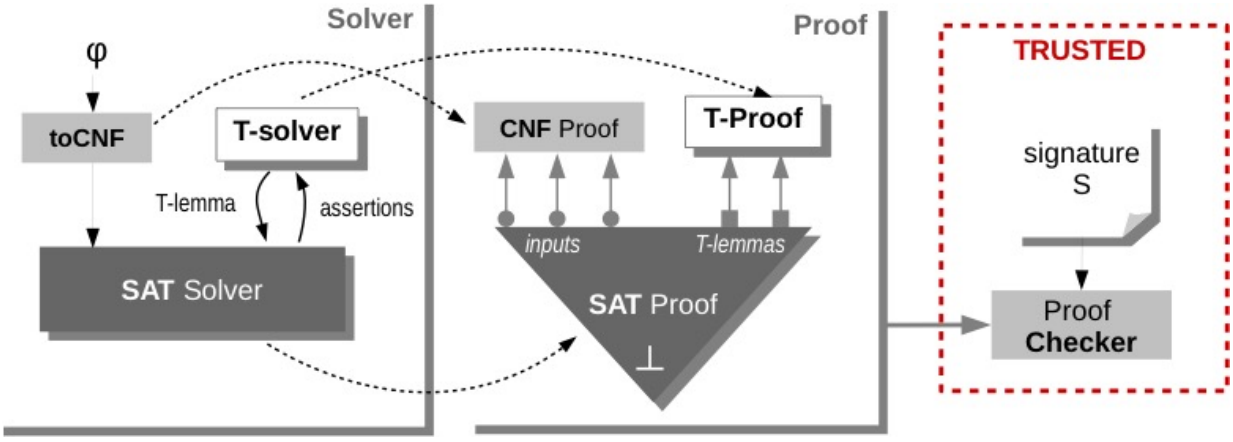

Figure 7 DPLL(T) architecture, SMT proof structure, and proof checker.

Figure 7 depicts the DPLL($\mathcal{T}$) architecture and how it relates to the structure of SMT proofs. Below, we describe proof production in three common theories: uninterpreted

[5] This step typically also includes the application of simplifying rewrite rules as discussed in Section 3.3. We ignore this issue in this chapter. Extending the approach here to include the many pre-processing rewrite rules used in real solvers is tedious but straightforward.

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.

4.1.1 Uninterpreted Functions

A general scheme for a proof-producing T_{UF}-solver was proposed by Fontaine *et al.* [21]. 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 $x = y$) [21]. 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 8 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 8 A refutation of { x = y, z = f(y), f(x) ≠ 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 4.3) suggests that the lazy approach is superior to the eager approach for T_{UF}.

4.1.2 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} [22] 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 literals, $a = b$ 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 9 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 9 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 4.3) suggest that, despite this extra bookkeeping, the lazy approach is superior to the eager approach for T_{AX}.

4.1.3 Bit-vectors

Proofs for the theory of fixed-width bit-vectors are of particular practical importance, with applications in both hardware and software verification. Previous work [23] shows how to reconstruct proofs from the Z3 SMT solver in HOL4 and Isabelle/HOL. However, due to the lack of detail in the Z3 bit-vector proofs, proof reconstruction is not always successful. In this section, we present a method of encoding and checking fine-grained SMT -generated proofs for the theory $\mathcal{T}_{\mathrm{bv}}$ of bit-vectors as formalized in the SMT-LIB 2 standard [16]. Proof generation and checking for the bit-vector theory poses several unique challenges. Algebraic reasoning is typically not sufficient by itself to decide most bit-vector formulas of practical interest, so often bit-vector (sub)-problems are solved by reduction to SAT. However, such reductions usually result in very large propositional proofs. In addition, the reduction itself must be proven correct. Encoding the $\mathcal{T}_{\mathrm{bv}}$ proof rules in LFSC helps address some of these challenges.

We make the following contributions: (i) we develop an LFSC proof system for the quantifier-free theory of fixed-width bit-vectors that includes proof rules for bit-blasting and allows for a two-tiered DPLL($\mathcal{T}$) proof structure; and (ii) we report experimental results on an extensive set of unsatisfiable SMT-LIB benchmarks in the QF_BV logic.

We discuss how bit-vector constraints are decided in CVC4 and how to generate proofs for them in Section 4.1.4, and Section 4.1.5 introduces the LFSC proof rules that are specific to the bit-vector theory.

4.1.4 Bit-vector proof generation in CVC4

Decision procedures for the theory $\mathcal{T}_{\mathrm{bv}}$ of bit-vectors almost always involve a reduction to propositional logic. One approach for encoding a bit-vector formula φ into an equisatisfiable propositional formula φ^{BB} is known as *bit-blasting*. For each variable v denoting a bit-vector of size n, bit-blasting introduces n fresh propositional variables, $v_0, \ldots v_{n-1}$, to represent each bit in the vector. To be able to encode this mapping in $\mathcal{T}_{\mathrm{bv}}$, we extend the $\mathcal{T}_{\mathrm{bv}}$ signature with a family of interpreted predicate symbols $(\mathsf{bitOf}_i : \mathsf{BV}_n \mapsto \mathsf{bool})_{0 \leq i < n}$, where bitOf_i takes a bit-vector x of width n and returns *true* iff the i^{th} bit of x is 1. Let φ be a bit-vector formula. For each atom a appearing in φ, let $bbAtom(a)$ denote a propositional formula consisting of the circuit representation of a. Let C^{BB} denote the conjunction of bit-blasting clauses obtained from converting to CNF the atom definitions:

$$C^{BB} \equiv CNF\left(\bigwedge_{a \in Atoms(\varphi)} a^{BB} \Leftrightarrow bbAtom(a)\right),$$

where a^{BB} is a fresh propositional variable representing atom a and CNF represents conversion to CNF. The formula $\varphi^{BB} := \varphi[a \mapsto a^{BB}]_{a \in Atoms(\varphi)} \wedge C^{BB}$ is a propositional

formula equisatisfiable with φ. Most state-of-the-art solvers for $\mathcal{T}_{\text{bv}}$ generate a formula like φ^{BB} and then rely on a single query to a SAT solver to check its satisfiability. Thus, a proof of unsatisfiability for φ could consist of: (i) a proof that φ is equisatisfiable with φ^{BB} in $\mathcal{T}_{\text{bv}}$, (ii) a propositional proof that φ^{BB} is equisatisfiable with $CNF(\varphi^{BB})$, and (iii) a monolithic, potentially very large, resolution-based refutation of $CNF(\varphi^{BB})$.

CVC4 incorporates an *eager* bit-vector decision procedure (cvcE) based on the approach sketched above. It also provides, as an alternative, a lazy DPLL($\mathcal{T}$) -style bit-vector solver (cvcLz) that maintains the word-level structure of the input terms and separates reasoning over the propositional structure of the input formula φ from bit-vector term reasoning [24]. In cvcLz, the bit-vector theory is treated like any other theory: the main DPLL($\mathcal{T}$) SAT engine SAT_{main} reasons on the propositional abstraction φ^{P} whereas a $\mathcal{T}_{\text{bv}}$ -solver decides conjunctions A of $\mathcal{T}_{\text{bv}}$ -literals.

Recall from Chapter 3.3 that the $\mathcal{T}_{\text{bv}}$ solver must *repeatedly* decide the satisfiability of the $\mathcal{T}_{\text{bv}}$ -literals A and return a $\mathcal{T}_{\text{bv}}$ -valid clause over the atoms of A if A is $\mathcal{T}_{\text{bv}}$ -unsatisfiable. We achieve this by relying on a second SATsolver, SAT_{bb}, to decide the satisfiability of each assignment . It does this by checking the propositional formula $A^{BB} \wedge C^{BB}$, where $A^{BB} = A[a \mapsto a^{BB}]_{a \in Atoms(A)}$. Note that this may be significantly smaller than the formula $\varphi[a \mapsto a^{BB}]_{a \in Atoms(\varphi)} \wedge C^{BB}$ checked in the eager approach.

If $A^{BB} \wedge C^{BB}$ is unsatisfiable, SAT_{bb} returns a set of literals $L^{BB} \subseteq A^{BB}$ that is inconsistent with C^{BB}. The clause $\neg L$ is a $\mathcal{T}_{\text{bv}}$ -valid lemma, and the $\neg L^{\text{P}}$ clause is added to SAT_{main}. We can efficiently use SAT_{bb} to check the satisfiability of C^{BB} with different assumptions A^{BB} by using the *solve with assumptions* feature of SAT solvers [25].

The lazy solver cvcLz in CVC4 also has several algebraic word-level sub-solvers. However, we do not yet support proof production for these sub-solvers, so in this chapter, we focus on the $\mathcal{T}_{\text{bv}}$ -lemmas generated by SAT_{bb}.

4.1.5 LFSC Bit-vector signature

In this section, we discuss proof generation for the lazy bit-vector solver described in Section 4.1.4. Figure 10 shows the overall structure of the $\mathcal{T}_{\text{bv}}$ proof by zooming in on the $\mathcal{T}_{\text{bv}}$ -lemmas that occur as leaves in the resolution SAT proof. We start with the bit-blasting proofs that each atom a is equivalent to its bit-blasted formula: $a \Leftrightarrow bbAtom(a)$. These proofs require no assumptions as $a \Leftrightarrow bbAtom(a)$ is $\mathcal{T}_{\text{bv}}$ -valid.[6] Next, the CNF proof establishes that the bit-blasting clauses C^{BB} follow from the atom definitions.[7] Note that this step also establishes the mapping from the $\mathcal{T}_{\text{bv}}$ -atom a to the abstract Boolean variable a^{BB} used in the SAT_{bb} SAT solver.

---

[6] Recall that $bbAtom(a)$ is a propositional formula encoding the semantics of atom a, and contains bitOf_i applications on the bit-vector variables in a.

[7] For details on how to use LFSC to encode proofs for CNF conversion, see [18]

Each $\mathcal{T}_{bv}$ -lemma has a corresponding resolution proof in SAT_{bb} with C^{BB} as leaves. The resolution proof constructs a clause over the a^{BB} SAT variables. To use this in SAT_{main}, we need to map the lemma to $\mathcal{T}_{bv}$ atoms, and then to the SAT variables a^{P} in SAT_{main}. In the figure, circles denote $\mathcal{T}_{bv}$ -atoms and diamonds the propositional variables that abstract them (either in SAT_{bb} or in SAT_{main}).

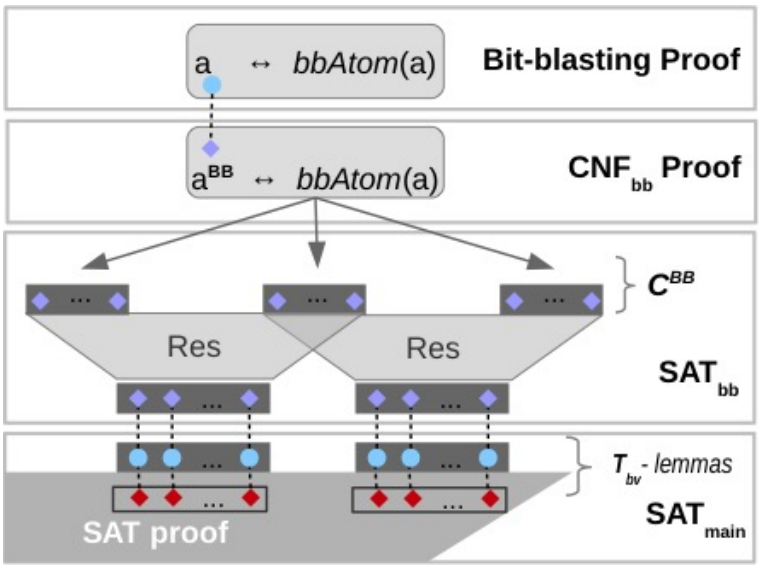

Figure 10 Bit-vector proof structure.

4.1.6 Encoding bit-vector formulas

sort : type	term : sort → type	BV : int → sort
form : type	true, false : form not : form → form	and, or, impl, iff : form → form → form = : Πs:sort. term s → term s → form
varBV : type	var2BV : Πn:int. varBV → term (BV n)	
bit : type	b0, b1 : bit	const2BV : Πn:int. constBV → term (BV n)
constBV : type	bvn : constBV	bvc : bit → constBV → constBV

Figure 11 Partial LFSC signature for the theory $\mathcal{T}_{bv}$ of bit-vectors

Figure 11 shows the LFSC constructs needed to represent formulas in the theory of bit-vectors. Note that the encoding distinguishes between formulas and terms: formulas are represented by the simple type form and terms by the dependent type term, parametrized by the sort of the term: Πs: sort. term s. Formulas are constructed with the usual logical operators and with an equality operator over terms which is parametric in the terms' sort. The int type is LFSC's own built-in infinite precision integer type. Bit-vector sorts are represented by the dependent type Πn: int. BV n where n is the width of the bit-vector. Bit-

vector constants are represented as lists of bits using the constBV type with the two constructors bvn and bvc, for the empty sequence and the list cons operator respectively. The constBV bit-vector constants are converted to bit-vector terms with the const2BV function. Bit-vector variables are represented as LFSC variables of type varBV and converted to terms with var2BV.

Example 5. *The bit-wise conjunction operator is encoded in LFSC as:*

$$\text{bvand}: \Pi n{:}\,\text{int}.\ \text{term}\ (\text{BV}\ n) \rightarrow \text{term}\ (\text{BV}\ n) \rightarrow \text{term}\ (\text{BV}\ n)$$

Similarly, the unsigned comparison operator $<$ is encoded as:

$$\text{bvult}: \Pi n{:}\,\text{int}.\ \text{term}\ (\text{BV}\ n) \rightarrow \text{term}\ (\text{BV}\ n) \rightarrow \text{form}$$

The $\mathcal{T}_{bv}$ formula $(t_1 = t_2\ \&\ t_3) \vee (t_1 < 0_{[3]})$ where & is bvand, $0_{[3]}$ is the zero bit-vector of size 3, and t_1, t_2, t_3 have type (term (BV 3)) *can be encoded in LFSC as*

```
(or    (= _ t1 (bvand _ t2 t3))
       (bvult _ t1 (const2BV 3 (bvc b0 (bvc b0 (bvc b0 bvn))))))),
```

with b0 *representing the zero bit.*

4.1.7 Bit-blasting

bbt : type bbtn : bbt bbtc : formula → bbt → bbt

bitOf : varBV → int → form bbTerm : Πn:int. term (BV n) → bbt → type

bb-var (v : varBV, n : int) : bbt =
if $n < 0$ *then* bbtn *else* (bbtc (bitOf v n) (bb-var v $(n-1)$))

bbVar : Πn:int. Πv:varBV.
Πvb:bbt {(bb-var v $(n-1)$) ↓ vb}. (bbTerm n (var2BV n v) vb)

bbAnd : Πn:int. $\Pi x, y$:term (BV n). $\Pi xb, yb, rb$:bbt.
Πxbb:bbTerm n x xb.
Πybb:bbTerm n y xb {(bb-bvand xb yb) ↓ rb}. bbTerm n (bvand n x y) rb

bbEq : Πn:int. $\Pi x, y$:term (BV n). $\Pi bx, by$:bbt. Πf:form.
Πbbx:bbTerm n x bx.
Πbby:bbTerm n y by {(bb-eq bx by) ↓ f}. thHolds (iff (= (BV n) x y) f)

Figure 12 Partial list of the LFSC bit-blasting rules for $\mathcal{T}_{bv}$.

Recall that a bit-blasting proof (see Figure 10) makes the connection between a bit-vector formula and its propositional logic encoding by proving for each bit-blasted atom a in the input formula, the following formula:

$$a \Leftrightarrow bbAtom(a).$$

We represent a bit-blasted bit-vector term of width n as a sequence of n formulas, with the i^{th} formula in the sequence corresponding to the i^{th} bit. The bbt type encodes bit-blasted terms and has two type constructors bbtn and bbtc as shown in Figure 12. We introduce the dependent type constructor bbTerm to encode the fact that the bit-vector term x: BV n. corresponds to a bit-blasted term y: bbt. For example, the following term encodes that $15_{[4]}$ is bit-blasted as $[true, true, true, true]$:

```
(bbTerm _   (const2BV 4 (bvc b1 (bvc b1 (bvc b1 (bvc b1 bvn )))))
            (bbtc true (bbtc true (bbtc true (bbtc true bbtn)))))
```

We can define proof rules for each piece of syntax in bit-vector terms and compose them in order to build up arbitrary bit-blasted terms. Figure 12 shows several such bit-blasting rules. The bbVar rule takes a bit-vector variable v, its width n, and a sequence of bit-blasted terms vb, and checks that the sequence computed by the side condition code in $\mathsf{bb-var}$ matches vb. The side condition code just builds a sequence of applications of the bitOf operator to v—with (bitOf $v\ i$) representing the $\mathcal{T}_{\mathsf{bv}}$ predicate bitOf_i introduced at the beginning of Section 4.1.4. Similarly, the rule that establishes how to bit-blast bit-wise conjunction (&) takes a proof xbb that xb is the bit-blasted term corresponding to x as well as a proof ybb for yb corresponding to y and returns a proof that $x\&y$ is bit-blasted to rb. The rb term is constructed by the side condition code $\mathsf{bb-bvand}$ (not shown) which works similarly to −var . The bbEq rule for equality $\mathcal{T}_{\mathsf{bv}}$ -atoms follows a similar pattern, but returns a formula instead of a Term . Note that bit-blasting proof rules do not take any $\mathcal{T}_{\mathsf{bv}}$ -assertions as assumptions: their conclusions are $\mathcal{T}_{\mathsf{bv}}$ -valid.

Example 6. *Encoding in LFSC the bit-blasting proof for the formula* $a_{[8]} = x_{[8]}\&y_{[8]}$ *requires the following proof rule applications:*

```
(bbEq _ _ _ _ _ _ (bbVar 8 a _ ) (bbAnd _ _ _ _ _ _ (bbVar 8 x _ ) (bbVar 8 y _ )))
```

Assuming previously defined variables $a, x,$ *and* y*, the above term has type* $thHolds(\varphi)$ *where* φ *is:*

$$(a_{[8]} = x_{[8]}\&y_{[8]}) \Leftrightarrow \bigwedge_{0 \le i < 8} (a_i \Leftrightarrow (\text{bitOf } v\ i) \wedge (\text{bitOf } v\ i)).$$

The bit-blasting LFSC proof rules rely on the side-condition code to build up the bit-blasted terms. This side-condition code thus becomes part of the trusted core and offers an efficient way to encode bit-blasting proofs.

4.1.8 Resolution in $\mathsf{SAT_{bb}}$

A resolution refutation can be obtained from a SAT solver by instrumenting it to store resolution proofs of all the clauses learned during search. The empty clause is then derived

by resolving input clauses and learned clauses. Recall that SAT_{bb} uses "solve with assumptions" to identify a subset $L^{BB} \subseteq A^{BB}$ that is inconsistent with C^{BB} and thereby produce the theory lemma $\neg L$. Because the assumption literals are implemented as decisions in SAT_{bb}, all clauses learned in SAT_{bb} follow from the bit-blasting clauses alone and can thus be reused in subsequent checks by SAT_{bb}. In particular, we can retrieve a resolution proof of the $\neg L^{BB}$ clause from SAT_{bb} starting from the bit-blasting clauses C^{BB} and using the stored resolutions of the learned clauses. We are careful to reuse the resolution proofs of learned clauses in multiple $\mathcal{T}_{\text{bv}}$ lemmas.

Stepping back and examining the overall $\mathcal{T}_{\text{bv}}$ proof structure, it looks like we could obtain one big resolution proof if we could plug the SAT_{bb} resolution trees into the SAT_{main} resolution tree. However, this cannot be done directly as the SATvariable a^{BB} abstracting $\mathcal{T}_{\text{bv}}$ -atom a in the resolution proof in SAT_{bb} is not the same as the a^{P} variable used to abstract the same atom in SAT_{main}. Therefore, we need a proof construct to map the proof of a clause c^{BB} to c^{P} (the dashed lines between SAT_{main} and SAT_{bb} in Figure 10).

In previous work on encoding SMT proofs in LFSC [18], we developed a specialized proof rule assump used to transform a $\mathcal{T}$ -proof of $\bigwedge_{i=0}^{n} \neg\, l_i \vDash_{\mathcal{T}} \bot$ to a proof of the clause $c^{\text{P}} = [l_1^{\text{P}}, \ldots, l_n^{\text{P}}]$ where we use the square brackets as a shorthand for the LFSC syntax for clauses. Chaining assump rules turns a term of type $\text{thHolds}(\neg l_1) \to \cdots \to \text{thHolds}(\neg l_n).\,\text{holds cln}$ into a term of type holds $[l_1^{\text{P}} \ldots l_n^{\text{P}}]$. Our goal here is to build a proof that takes as assumptions the negation of each literal l_i as well as a proof of the clause $c^{BB} = [l_1^{BB}, \ldots, l_n^{BB}]$ and returns a term of type holds cln. We will do this using the introUnit rule: [8]

introUnit: Πf: form. Πv: var. Πc: clause.
thHolds $f \to$ atom $v\ f \to$ (holds $[v] \to$ holds c) $\to$ holds c

This natural deduction style rule states that if formula f holds (thHolds f) and is abstracted by propositional variable v (atom $v\ f$), and if we can derive clause c from the unit clause corresponding to f (holds $[v] \to$ holds c), then we can derive clause c.

Example 7. *We show how to put these rules together to lift a proof of a clause in SAT_{bb} to a proof of the corresponding clause in SAT_{main}. In the sub-expression below, assume c has type* holds $[\neg a_1^{BB}, \neg a_2^{BB}]$ *and that* at_1 *and* at_2 *have types* $\text{atom}(a_1^{BB}, a_1)$ *and* $\text{atom}(a_2^{BB}, a_2)$*, respectively. The two resolution steps between the assumption unit clauses* u_1 *and* u_2 *derive the empty clause from c. Therefore, the computed type of the following term is* $\text{thHolds}(\text{not } a_1) \to \text{thHolds}(\text{not } a_2) \to \text{holds cln}$*, which is exactly what the assump rule requires:*

[8] For simplicity, introUnit only introduces literals in positive polarity. In reality, we also use a dual version that introduces literals in negative polarity.

λh_1: thHolds(not a_1). λh_2: thHolds(not a_2).
(introUnit _ _ _ h_1 at_1 (λu_1: (holds[a_1^{BB}]).
(introUnit _ _ _ h_2 at_2 (λu_2: (holds[a_2^{BB}]).
(Res _ _ (Res _ _ c u_1 v_1) u_2 v_2)))))

4.2 SMTCoq: communication between Coq and SMT solvers

SMTCoq[9] [3] is a tool that allows the Coq [26] proof assistant to communicate with external automatic solvers for Boolean satisfiability (SAT) and Satisfiability Modulo Theories (SMT). Its twofold goal is to:

- increase the confidence in SAT and SMT solvers: SMTCoq provides an independent and certified checker for SAT and SMT proof witnesses;
- safely increase the level of automation of Coq: SMTCoq provides starting safe tactics to solve a class of Coq goals automatically by calling external solvers and checking their answers (following a *skeptical* approach).

With our extensions, SMTCoq currently supports the SAT solver ZChaff [27] and the SMT solvers veriT [28] and CVC4 [20] for the quantifier-free fragment of the combined theory of linear integer arithmetic, equality with uninterpreted functions, bitvector arithmetic and functional arrays. There is a large variety of SAT and SMT solvers, with each solver typically excelling at solving problems in some specific class of propositional or first-order problems. While the SAT and SMT communities have adopted standard languages for expressing *input* problems (namely the DIMACS standard for SAT and the SMT-LIB [29] standard for SMT), agreeing on a common *output* language for proof witnesses has proven to be more challenging. Several formats [18], [30], [31] have been proposed but none has emerged as a standard yet. Each proof-producing solver currently implements its own variant of these formats.

To be able to combine the advantages of multiple SAT and SMT solvers despite the lack of common standards for representing proof certificates, SMTCoq has been designed to be modular along two dimensions:

- supporting new theories: SMTCoq's main checker is an extendable combination of independent *small checkers*;
- supporting new solvers: SMTCoq's kernel relies on a generic certificate format that can encode most SAT and SMT reasonings for supported theories; the encoding can be done during a *preprocessing* phase, which does not need to be certified.

[9] SMTCoq is distributed as free software at https://github.com/LFSC/smtcoq/tree/v1.3-darpa.

In this report, we emphasize the key ideas behind the modularity of SMTCoq , and validate this by reporting on the work of the integration of the SMT solver CVC4 [20], the theory of bit vectors and functional arrays. This work simultaneously aims at:

- offering CVC4 users the possibility to formally check CVC4 proofs in a trusted environment like Coq;
- bringing the power of a versatile and widely used SMT solver like CVC4 to Coq ;
- providing in Coq decision procedures for
  - bit-vectors: a theory widely used, for instance, for verifying circuits or programs using machine integers or bit-level representation of floating-point numbers, and
  - functional arrays: a theory which is used in program verification to encode programming languages arrays but also to represent memory.

4.2.1 The SMTCoq Tool

General Idea

The heart of SMTCoq is a checker for a generic format of certificates (close to the format proposed by Besson *et al.* [31]), implemented and proved correct inside Coq (see Figure 13). Taking advantage of Coq 's computational capabilities, the SMTCoq checker is executable inside Coq .

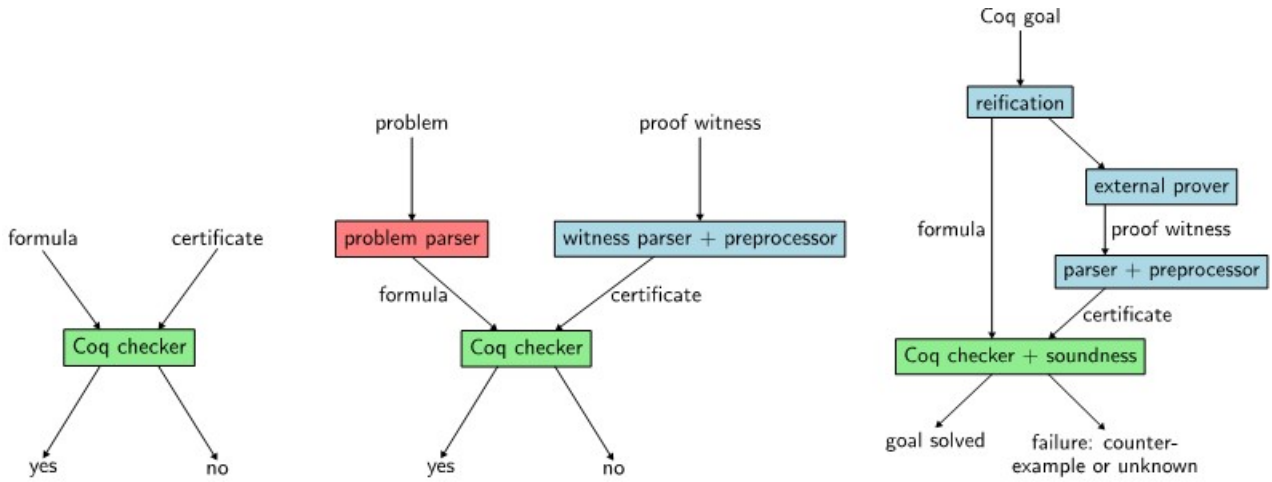

Figure 13 SMTCoq's main checker and its uses.

The Coq signature of this checker is the following:

```
checker : formula → certificate → bool
```

where the type `formula` represents the deep embedding (implementation of the syntax with an interpretation to it) in Coq of SMT formulas, and the type `certificate` represents SMTCoq 's format of certificates.

The checker's soundness is stated with respect to a translation function from the deep embedding of SMT formulas into Coq terms:

```
⟦•⟧ : formula → bool
```

that interprets every SMT formula into its Coq Boolean counterpart. The correctness of the checker is given by the following lemma:

which states that whenever the checker returns true for a given a formula and a certificate for it, the interpretation in Coq of the formula is a valid Boolean sentence.

The choice of the type of Booleans `bool` as the codomain of the translation function ⟦•⟧, instead of the type of (intuitionistic) propositions `Prop`, allows us to handle the checking of the classical reasoning made by SMT solvers without adding any axioms to Coq. The SSReflect [32] plugin for Coq can be used to bridge the gap between propositions and Booleans for the theories considered by SMTCoq. The major shortcoming of this approach is that it does not allow quantifiers inside goals sent to SMT solvers, although it does not prevent one from feeding these solvers universally quantified lemmas. The first use case of this correct-by-construction checker is to check the validity of a proof witness, or proof *certificate*, coming from an external solver against some input problem (Figure 13, middle). In this use case, the trusted base is both Coq and the parser of the input problem. The parse is part of the trusted based because we need to make sure we are effectively verifying a proof of the problem we sent to the external solver. However, this parser is fairly straightforward.

The second use case is within a Coq tactic (Figure 13, right). We can give a Coq goal to an external solver and get a proof certificate for it. If the checker can validate the certificate, the soundness of the checker allows us to establish a proof of the initial goal. This process is known as *computational reflection* as it uses a computation (here, the execution of the checker) inside a proof. In this use case, the trusted base consists only of Coq: if something else goes wrong (e.g., the checker cannot validate the certificate), the tactic will fail, but nothing unsound will be added to the system.

In both cases, a crucial aspect for modularity purposes is the possibility to *preprocess* proof certificates before sending them to the SMTCoq checker, without having to prove anything about this preprocessing stage. Again, if the preprocessor is buggy, the checker will fail to validate the proof certificate (by returning `false`), which means that while nothing is

learned, nothing unsafe is added to Coq's context. This allows us to easily extend SMTCoq with new solvers: as long as the certificate coming from the new solver can be logically encoded into SMTCoq 's certificate format, we can implement this encoding at the preprocessing stage. As a result, SMTCoq 's current support for ZChaff , veriT and CVC4 is provided through the implementation of a preprocessor for each solver. These preprocessors convert to the same proof format, thus sharing the same overall checker.

Using a preprocessor is also beneficial for efficiency: proof certificates may be encoded more compactly before being sent to the SMTCoq checker, which may improve performance.

The Checker

We now provide more details on the checker of SMTCoq . As presented in Figure 14, it consists of a *main checker* obtained as the combination of several *small checkers*, each specialized in one aspect of proof checking in SMT (e.g., CNF conversion, propositional reasoning, reasoning in the theory of equality, linear arithmetic reasoning, bit-vector arithmetic, functional arrays and so on).

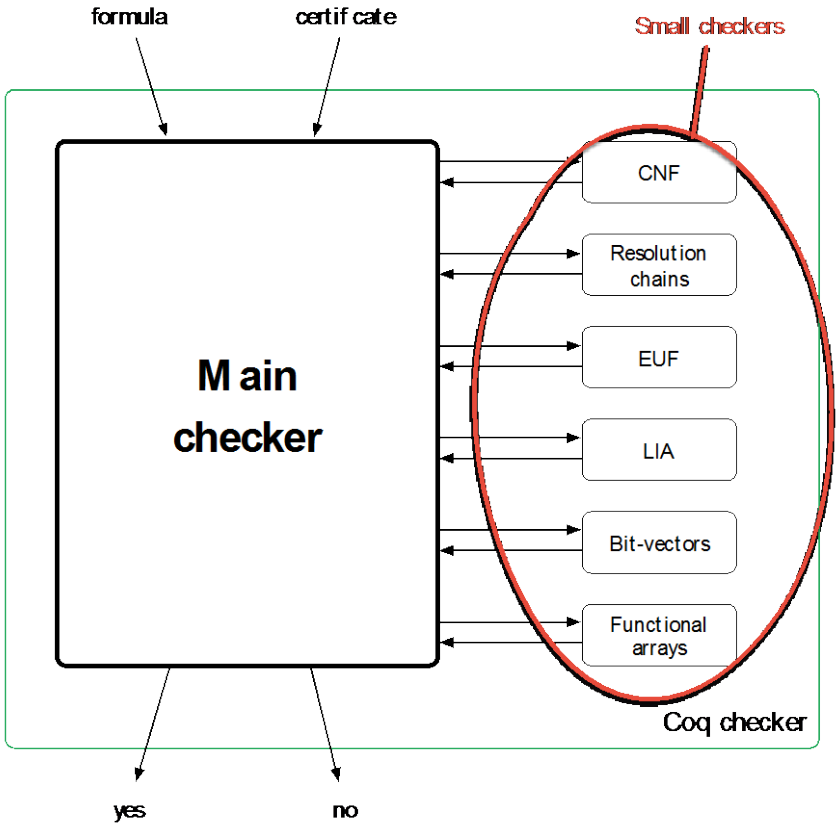

Figure 14 Internals of the Coq checker.

The type `certificate` is actually the aggregation of specialized types, one for each small checker. The role of the main checker is thus to dispatch each piece of the certificate to its dedicated small checker, until the initial formula is proved.

A small checker is a Coq program that, given a (possibly empty) list of formulas and a certificate step associated with it, computes a new formula:

```
small_checker : list formula → step → formula
```

The soundness of the `checker` comes from the soundness of each small checker, stated as follows:

```
Lemma small_checker_sound : ∀ f1 … fn c,
    ⟦f1⟧ ∧ … ∧ ⟦fn⟧ → ⟦small_checker [f1;...;fn] c⟧
```

meaning that the small checker returns a formula which is implied (after translation into Coq 's logic) by the conjunction of its premises. Note that the list of premises may be empty: in such a case, the small checker returns a tautology in Coq.

Here are some examples of small checkers.

- For propositional resolution chains, the checker takes as input a list of premises and returns a resolvent if it exists, or a trivially true clause otherwise. In this case, a certificate is not required as part of the small checker's input.
- For the theory of equality with uninterpreted functions (EUF), the checker takes as input a formula in this theory formulated as a certificate (corresponding to a theory lemma produced by the SMT solver), and returns the formula if it is able to check it, or a trivially true clause otherwise. In this case, no premises are given.
- For linear integer arithmetic (LIA), the checker works similarly to the EUF checker, but checks the formula using Micromega [33], an efficient decision procedure for this theory implemented in Coq.
- For bit-vectors, the checkers work like the EUF one but some of them have premises. Most rules concern bit-blasting bit-vector operators, and so the formulas manipulated by this checker use a special predicate to relate bit-vectors and their bit-level interpretation.
- For the theory of functional arrays with extensionality (A), the checker takes as input a formula of this theory (corresponding to an instance of one of the three axioms of arrays), returns it if it is able to check it, or the true clause otherwise. This checker only produces tautologies.

The only thing that small checkers need to share is the type `formula`, and its interpretation into Coq Booleans. Each small checker may then reason independently, using separate pieces of the certificate. Again, this is crucial for modularity: to extend SMTCoq with a new theory, one only has to extend the type `formula` with the signature of this theory and,

independently of the already existing checkers, implement a small checker for this theory and prove its soundness.

Notice that "small checker" can be understood in a very general sense: any function that, given a list of first-order formulas, returns an implied first-order formula, can be plugged into SMTCoq as a small checker. In principle, such a checker could even be as complex as an SMT solver, as long as it can be proved correct in Coq.

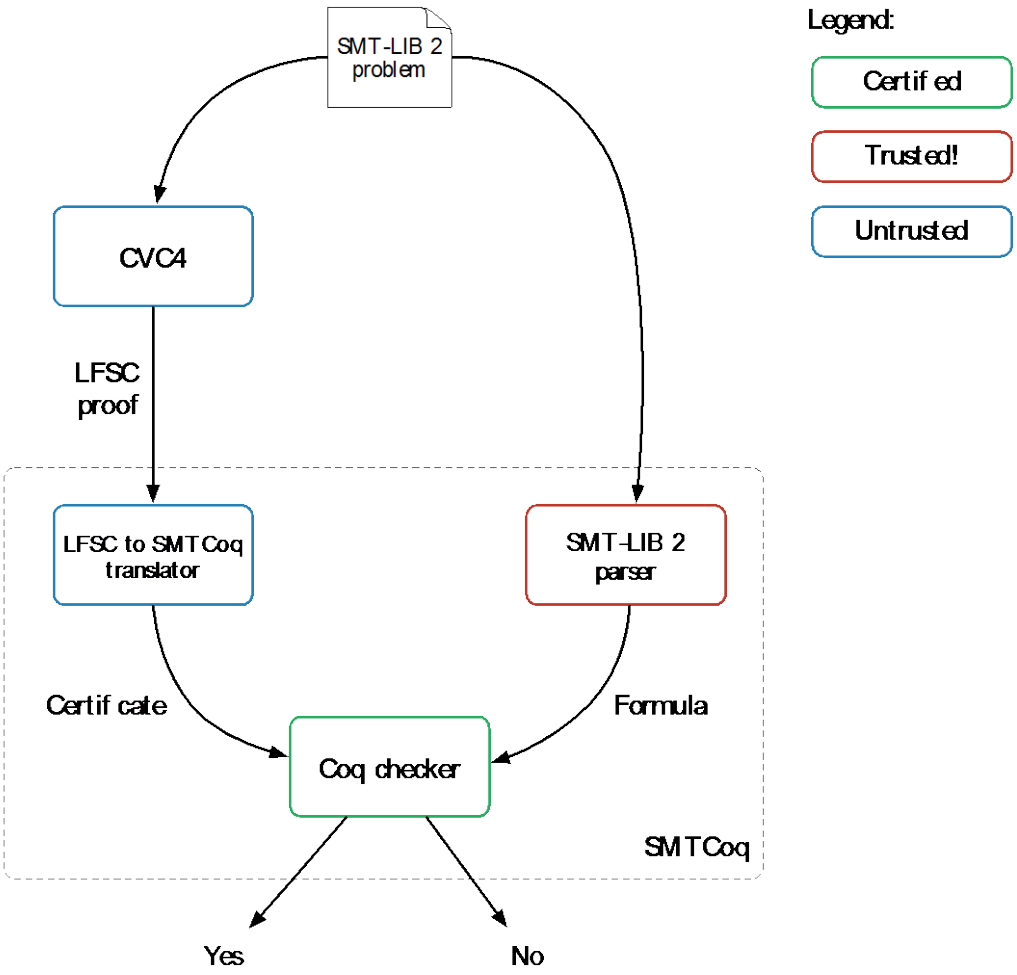

Figure 15 Integration of CVC4 in SMTCoq.

4.2.2 Extending SMTCoq to support CVC4

Approach for supporting CVC4

As detailed above, CVC4 is a proof-producing SMT solver, whose proof format is based on the Logical Framework with Side Conditions (LFSC) [18]. LFSC extends the Edinburgh Logical Framework [17] by allowing types with computational *side conditions*, explicit

computational checks defined as programs in a small but expressive functional first-order programming language. On the other hand, SMTCoq supports certificates in a restricted format. Our approach for supporting the SMT solver CVC4 in SMTCoq is to write a translator/preprocessor from the fragment of LFSC in which CVC4 produces its proofs to the internal certificate representation of SMTCoq, as shown in Figure 15.

LFSC proof witnesses

The LFSC language has built-in types for arbitrary precision integers and rationals, ML-style pattern matching over LFSC type constructors, recursion, a minimal support for exceptions, and a very restricted set of imperative features. One can define proof rules in LFSC as typing rules that may optionally include a side condition written in this language. When checking the application of such proof rules, an LFSC checker computes actual parameters for the side condition and executes its code; if the side condition fails, the LFSC checker rejects the rule application. The validity of an LFSC proof witness thus relies on the correctness of the proof rules which it uses. In particular, this includes any side conditions functions used in the rules. CVC4 defines its own proof rules for the various theories it supports in LFSC files called *signatures*. These include:

- sat.plf: Definitions for propositional variables, clauses, and rules for resolution. The resolution used in this signature is called *deferred resolution* and has a strong computational content.
- smt.plf: This file defines the language of terms and formulas used in the logic of SMT solvers as well as CNF-conversion rules and mappings from SMT atoms to propositional variables.
- th_base.plf: Constructors for functions types and applications are defined in this file. The rules of congruence of uninterpreted functions together with rules of equality are defined here also. These rules represent what is usually referred to as the EUF (equality over uninterpreted function) theory (or the empty theory) in SMT solvers.
- th_int.plf: This file only provides constructors and symbols for linear integer arithmetic. There are no rules yet as CVC4 doesn't produce proofs for arithmetic yet. We show later though that this is not a problem for SMTCoq.
- th_bv.plf: This file contains types and constructors for the theory of fixed-size bit vectors. All symbols and operators are declared here, while rules for *bit-blasting* these operators are defined in an accompanying file . These rules too have a strong computational content through the use of side-conditions.
- th_arrays.plf: This file contains the two operators read and write for the theory of functional arrays as well as rules that encode the usual axiomatization of this theory.

Example 8 (Simple rules in LFSC). *This is how the rule for elimination of disjunction (*or*) is written in LFSC . The judgment* th_holds *is declared for terms of type* formula *and means that the formula is valid (or holds) in all models. The corresponding formulation as an inference rule in mathematical notation is given on the right. There,* th_holds *is written as* ⊨.

```
(declare or_elim_2
  (! f1 formula
  (! f2 formula
  (! u1 (th_holds (not f2))
  (! u2 (th_holds (or f1 f2))
    (th_holds f1))))))
```

$$\textit{Or-Elim-2}\ \frac{\models \neg\, \mathtt{f2} \qquad \models \mathtt{f1} \vee \mathtt{f2}}{\models \mathtt{f1}}$$

Example 9 (Rules with side-conditions in LFSC). *We use the same notations as the previous example. This rule bit-blasts the bitwise and operation on bit vectors of* size n (bvand)*. An extra judgment is present here,* bblast_term _ x f *which means that* f *is the bit-level interpretation (or bit-blasted formula) corresponding to the term* x*.*

```
(program bblast_bvand ((x bblt) (y bblt)) bblt
  (match x
    (bbltn (match y (bbltn bbltn) (default (fail bblt))))
    ((bbltc bx x') (match y
                     (bbltn (fail bblt))
                     ((bbltc by y') (bbltc (and bx by) (bblast_bvand x' y')))))))

(declare bv_bbl_bvand (! n mpz
                      (! x (term (BitVec n))
                      (! y (term (BitVec n))
                      (! xb bblt
                      (! yb bblt
                      (! rb bblt
                      (! xbb (bblast_term n x xb)
                      (! ybb (bblast_term n y yb)
                      (! c (^ (bblast_bvand xb yb ) rb)
                           (bblast_term n (bvand n x y) rb)))))))))))
```

The corresponding inference rule in mathematical notation is shown below, where the judgment above is denoted by bbT*.*

$$\textit{BvBblBvand}\ \frac{\mathtt{bbT}\ n\ x\ xb \qquad \mathtt{bbT}\ n\ y\ yb}{\mathtt{bbT}\ n\ (bvand\ n\ x\ y)\ rb}\ rb = \mathtt{bblast\_bvand}\ xb\ yb$$

This rule uses a side condition, which means that the rule can only be applied when rb *is equal to* (bblast_bvand xb yb) *where* bblast_bvand *is the small recursive functional program given earlier.*

SMTCoq certificates

SMTCoq does not support LFSC for proof certificates. Instead, it uses a format is strongly inspired by (and related to) the one proposed in [31]. A proof certificate in SMTCoq is a sequence of proof steps, where each proof step is either:

1. an input clause, or
2. the application of a *rule* to a (potentially empty) list of *derived* clauses together with a resulting clause.

Each resulting clause is identified by a unique number. SMTCoq already has a set of predefined rules whose checker (which ensures that the application of the rule yields the specified conclusion) are proven correct in Coq. Some of these rules are deductive, while some other are given in the form of a tautology (*i.e.*, rules with no premises). This format of certificates is inherently linear. For instance, the following proof step says that the result of resolving the clauses identified by `9` and `5` is the clause $(b \wedge d) \vee \neg d$, which is given the identifier `11`.

```
11:(resolution ((and b d) (not d)) 9 5)
```

The key differences between LFSC and the SMTCoq format are presented in Table 1. The major difference lies in the presentation of the deduction rules. In SMTCoq , the small checkers deduce a new formula from already known formulas, possibly with the help of a piece of certificate that depends on the theory. The LFSC format is more uniform, thanks to the side conditions described above.

Table 1 Main differences between the LFSC and SMTCoq certificate formats.

	LFSC	SMTCoq
Rules	deduction + computation	deduction + certificate
Nested proofs	supported	not supported

Translation

To support LFSC, and so CVC4 , we have implemented (in OCaml) an untrusted preprocessor that transforms LFSC proofs into SMTCoq proofs. To this end, for some theories, we need to replay parts of the side conditions, in order to produce the corresponding SMTCoq premises, conclusion and piece of certificate that will be passed to the small checkers. This requires us to perform type checking on the LFSC object itself as well. This encoding, however, is relatively straightforward:

- for propositional reasoning, LFSC side conditions use the same logical content as SMTCoq rules;

- CNF conversion and EUF proofs are nested in LFSC, so they require some processing for the moment;
- for linear integer arithmetic, since SMTCoq relies on an existing decision procedure in Coq, it only needs to know what theory lemma is being proved, and can ignore the actual proof steps in the LFSC certificate.

One difficulty in translating LFSC proofs to the SMTCoq format comes from the possibility in LFSC of using natural-deduction-style proofs, where one can nest one proof inside another. For instance, it is possible to have lemmas inside an LFSC proof whose witnesses are themselves LFSC proofs. The architecture of the main and small checkers of SMTCoq does not currently allow this sort of nesting: every clause produced by the small checkers needs to be a direct consequence of input clauses or clauses that were previously produced. To encode an LFSC proof into SMTCoq, our preprocessor thus linearizes nested proofs. The LFSC proofs generated by CVC4 are constructed in such a way that this does not cause a blow-up in practice; however, to support LFSC in general, we would need to extend SMTCoq certificates with nested proofs. Again, this extension should be made easier by the modularity inside the checker. It should impact only the main checker, and not the various small checkers already in SMTCoq.

Example 10. *This example shows the differences in proofs between the LFSC version and the SMTCoq one for a simple problem below, expressed in SMT-LIB 2 format:*

```
(set-logic QF_UF)
(declare-const a Bool)
(declare-const b Bool)
(declare-const c Bool)
(declare-const d Bool)
(assert (and a b))
(assert (or c d))
(assert (not (or c (and a (and b d)))))
(check-sat)
(exit)
```

The LFSC proof generated by CVC4 follows. One can notice the nesting of proof rules applications (e.g., `(not_not_intro _ (not_and_elim _ ...))`*).*

```
(check
 ;; Declarations
(% d (term Bool)
(% c (term Bool)
(% b (term Bool)
(% a (term Bool)
(% A3 (th_holds true)
(% A2 (th_holds (not (or (p_app c) (and (p_app a) (and (p_app b) (p_app d))))))
(% A1 (th_holds (or (p_app c) (p_app d)))
(% A0 (th_holds (and (p_app a) (p_app b)))
(: (holds cln)

 ;; Printing the global let map
(@ let1 (p_app a)
(@ let2 (p_app b)
(@ let3 (p_app c)
(@ let4 (p_app d)

 ;; In the preprocessor we trust
(th_let_pf _ (trust_f (iff (not (or let3 (and let1 (and let2 let4)))) (not (or let3 (and let1 (and
     let2 let4 ))))))(\ .PA220

(decl_atom let1 (\ .v2 (\ .a2
(decl_atom let2 (\ .v3 (\ .a3
(decl_atom let3 (\ .v4 (\ .a4
(decl_atom let4 (\ .v5 (\ .a5
(satlem _ _ (ast _ _ _ .a5 (\ .l11 (ast _ _ _ .a3 (\ .l7 (ast _ _ _ .a2 (\ .l5 (clausify_false
     (contra _ .l11 (or_elim_1 _ _ (not_not_intro _ .l7) (not_and_elim _ _ (or_elim_1 _ _
     (not_not_intro _ .l5) (not_and_elim _ _ (and_elim_2 _ _ (not_or_elim _ _ (or_elim_1 _ _
     (not_not_intro _ A2) (iff_elim_1 _ _ .PA220)))))))))))))))) (\ .pb8
(satlem _ _ (asf _ _ _ .a5 (\ .l10 (asf _ _ _ .a4 (\ .l8 (clausify_false (contra _ (or_elim_1 _ _
     .l8 A1) .l10)))))) (\ .pb6
(satlem _ _ (ast _ _ _ .a4 (\ .l9 (clausify_false (contra _ .l9 (and_elim_1 _ _ (not_or_elim _ _
     (or_elim_1 _ _ (not_not_intro _ A2) (iff_elim_1 _ _ .PA220)))))))) (\ .pb7
(satlem _ _ (asf _ _ _ .a3 (\ .l6 (clausify_false (contra _ (and_elim_2 _ _ A0) .l6)))) (\ .pb5
(satlem _ _ (asf _ _ _ .a2 (\ .l4 (clausify_false (contra _ (and_elim_1 _ _ A0) .l4)))) (\ .pb4

;; SAT proof
(satlem_simplify _ _ _ (R _ _ .pb6 .pb7 .v4) (\ .cl9
(satlem_simplify _ _ _ (Q _ _ (Q _ _ (Q _ _ .pb8 .cl9 .v5) .pb5 .v3) .pb4 .v2) (\ empty
     empty)))))))))))))))))))))))))))))))))))))))))
```

As an example of SMTCoq certificate, we take the proof generated by the SMT solver veriT (the `#i` *id's denote shared subterms).*

```
1:(input (#1:(and a b)))
2:(input (#2:(or c d)))
3:(input ((not #3:(or c #4:(and a #5:(and b d))))))
4:(and (a) 1 0)
5:(and (b) 1 1)
6:(or (c d) 2)
7:(not_or ((not c)) 3 0)
8:(not_or ((not #4)) 3 1)
9:(and_neg (#5 (not b) (not d)))
10:(not_and ((not a) (not #5)) 8)
11:(resolution (#5 (not d)) 9 5)
12:(resolution ((not #5)) 10 4)
13:(resolution (d) 6 7)
14:(resolution () 11 12 13)
```

Finally, the last listing shows a textual representation (in the same format as the one of veritT) of the translated version of the LFSC proof produced by CVC4.

```
1:(input ((not #1:(or c #2:(and a #3:(and b d))))))
2:(input (#4:(or c d)))
3:(input (#5:(and a b)))
4:(hole (#6:(= (not #1) (not #1))))
5:(equiv1 ((not (not #1)) (not #1)) 4)
6:(not_or ((not #2)) 1 1)
7:(not_and ((not a) (not #3)) 6)
8:(and_neg (#3 (not b) (not d)))
9:(resolution () 8 7)
10:(weaken ((not a) (not b) (not d)) 9)
11:(or (c d) 2)
12:(not_or ((not c)) 1 0)
13:(and (b) 3 1)
14:(and (a) 3 0)
15:(resolution (d) 11 12)
16:(resolution () 10 15 13 14)
```

The SMTCoq representation is more compact; however, the LFSC proof is more detailed.[10]

This translator (or preprocessor) is implemented as an OCaml functor which takes as argument a module providing facilities for translating terms, clauses and constructing rule applications. This functor is instantiated with a module that performs terms translation using the SMTCoq API directly, effectively enabling SMTCoq itself to accept certificates in the format of LFSC. It is also instantiated with a different module that simply prints terms and rules in the proof format of veriT. This allows us to provide a standalone translator

[10] The LFSC proof does omit terms in certain positions, replacing them by underscores, but these terms can be reconstructed automatically by type inference as needed.

from LFSC to veriT proofs which can be used independently The tool is called and is especially useful to debug and replay proofs in a step-by-step fashion.

4.2.3 Support for the Theory of Fixed-width Bit Vectors

We explained in Section 4.1 how we extended CVC4 to produce LFSC proofs for the quantifier-free fragment of the SMT theory of bit vectors. To check proof certificates in this theory, SMTCoq needed be extended as well. As explained in Section 4.2.1, to do that one needs to:

1. extend the Coq representation of formulas with the signature of the bit vector theory and the interpretation function into Coq terms;

2. implement (new) small checkers and their corresponding certificates for this theory, and prove their correctness.

Step 1 is a simple extension on the SMTCoq side. The major difficulty is that Coq itself has limited support for bit vectors. Its bit vector library provides only the implementation of bitwise operations (and not arithmetic operations), and no proofs. We have thus implemented a more complete library for this theory, which is detailed in Section 4.2.3. Step 2 involves implementing and adding new certified Coq programs, the small checkers. As already mentioned, however, because of SMTCoq's design, none of the previous small checkers and their proofs of correctness need to be changed as a result of this addition.

LFSC proofs for bit vectors produced by CVC4 mainly involve the following two kinds of deduction steps:

- *bit-blasting* steps that reduce the input bit vector formula to an equisatisfiable propositional formula;

- standard propositional reasoning steps (based on resolution).

The propositional steps can be handled directly by previous small checkers. For the bit-blasting steps, we have implemented new small checkers that relate terms of the bit vector theory with lists of Boolean formulas representing their bits, and proved the correctness in Coq for these small checkers.

LFSC proofs generated by CVC4 involve a third kind of step: formula simplifications based on the equivalence of two bit-vector terms or atomic formulas (for instance, by normalizing inequalities). Currently, these simplification steps are not provided a detailed LFSC subproof by CVC4, although there are plans to do so in the near future. In the current SMTCoq implementation then, we assume those steps, as in the LFSC proof coming from CVC4, or let the user prove them, in the case of tactics. Since those steps correspond to applications of CVC4 -defined rewriting and simplification rules, we plan for now to prove

the correctness of these rules once and for all at the Coq level, and to pre-process simplification steps into applications of these rules.

A Coq library for bit vector arithmetic: BVList

Our Coq library `BVList` consists of two *module types*, namely `BITVECTOR` and `RAWBITVECTOR`, that include the signatures of the abstract structure. Naively, they are just lists of Coq *parameters* and *axioms*. There is also a *functor module* `RAW2BITVECTOR` that takes an instance of `RAWBITVECTOR` and returns a module satisfying the `BITVECTOR` signature.

```
Module RAW2BITVECTOR (M: RAWBITVECTOR) <: BITVECTOR.
```

The notation <: says that we are making a module satisfying the `BITVECTOR` signature out of an instance `M` of type `RAWBITVECTOR`. In order to build bit vectors, this functor module uses the bit vector implementation coming from the input module `M`, together with a well-foundedness obligation on its size. They are respectively implemented as the fields `bv` and `wf` of the following Coq record:[11]

```
Record bitvector_ (n: N): Type ≜
  MkBitvector
    { bv :> M.bitvector;
      wf :  M.size bv = n
    }.
Definition bitvector ≜ bitvector_.
```

Observe that in the above implementation, the type of bit vectors is a *dependent type* which depends on the value of its parameter `n`, a Coq binary natural (the corresponding Coq type is noted `N` as opposed to the type of unary naturals `nat`).

We then make an instance module `RAWBITVECTOR_LIST` of `RAWBITVECTOR` where bit vectors are implemented as Coq's Boolean lists storing, in order, the individual bits of the bit vector (and so having a length equal to the bit vector size).

```
Definition bitvector ≜ list bool.
Definition size (a: bitvector) ≜ N.of_nat (List.length a).
Definition bits (a: bitvector) : list bool ≜ a.
```

As a consequence, any operation defined over bit vectors is encoded as an operation over Boolean lists. For instance, *bit vector and* is a binary operation that returns the list which is built by mapping *Boolean and* over the elements of input lists if input lists have the same size; *nil* otherwise.

---

[11] A Coq record is an inductive type with a single constructor, and associated projection functions for its fields.

```
Definition bv_and (a b : bitvector) : bitvector ≜
  match (size a) =? (size b) with
    | true ⇒ map2 andb (@bits a) (@bits b)
    | _    ⇒ nil
  end.
```

Using such kind of definitions, we can state and prove properties of bit vectors such as *the commutativity of and*:

```
Lemma bv_and_comm n a b : size a = n → size b = n → bv_and a b = bv_and b a.
```

This lemma can be proven by structural induction on the instance `a` and a case analysis on `b`. It is reflected in the interface `BITVECTOR` as

```
Axiom bv_add_comm : ∀ n (a b:bitvector n),
                    bv_eq (bv_add a b) (bv_add b a) = true.
```

Passing the module `RAWBITVECTOR_LIST` to the functor `RAW2BITVECTOR` as an argument, we get the module `BITVECTOR_LIST` satisfying the `BITVECTOR` signature. In this module, bit vectors are *dependently typed* with the values of Coq's binary naturals. This is not surprising since the module is built by the functor module. One can see the bit vector structure (a Coq record) by using the `destruct` tactic (in Coq 's proof editing mode) over an instance `a: bitvector n`:

```
{| bitvector_LIST.bv ≜ bv; bitvector_LIST.wf ≜ wf |}
```

where `bv` is an instance of type `RAWBITVECTOR_LIST.bitvector` which unfolds into a Boolean list while `wf` is of type `RAWBITVECTOR_LIST.size bv = n` which unfolds into the `Prop`[12] instance, `N.of_nat(length bv) = n`. We use this module to extend the structures of formulas of SMTCoq with bit vector types and operations as described earlier.

4.2.4 Support for the Theory of Functional Arrays with Extensionality

To support the theory of functional arrays, we have built a library on top of a version of Coq's `FMapList` [13] with type classes. Arrays are encoded as (finite) maps from keys to elements for the keys that are present in the map, with a default value for those keys not in the map. Since SMTCoq requires to have structural equality over terms, we have used such a formalization to have extensional equality over arrays that is reflected in the structural equality. In other words, two array terms denote the same array in the theory of arrays if and only if their Coq representations are structurally equal. This allows us to extend Coq

[12] `Prop`, the type of formulas, or *propositions*, in Coq, is a Coq universe just like `Set` and `Type`.

[13] See https://coq.inria.fr/library/Coq.FSets.FMapList.html for instance.

representation of formulas with the signature of the theory of arrays and with the interpretation into Coq terms.

This library provides a Coq type `farray` parameterized by two other types, one for the keys, and the other for the elements. In addition, this library requires the type of keys to have a *total order* (this is to ensure that maps are unique) and that the type of elements to be *inhabited* (this is to make sure that maps can have a default value). To this effect we provide type classes for the user to define its own instances (*e.g.*, for custom types).

```
Class EqbType T ≜ {
  eqb : T → T → bool;
  eqb_spec : ∀ x y, eqb x y = true ↔ x = y
}.

Class OrdType T ≜ {
  lt: T → T → Prop;
  lt_trans : ∀ x y z : T, lt x y → lt y z → lt x z;
  lt_not_eq : ∀ x y : T, lt x y → ~ eq x y
}.

Class Comparable T {ot:OrdType T} ≜ {
```

`EqbType` is a class of types with a Boolean equality that reflects in Leibniz equality (this is equivalent to saying that the type has decidable equality), `OrdType` is for types equipped with a partial order. Adding the function `compare` (through the class `Comparable`) makes this order total. Finally `Inhabited` only asks for the user to provide an element of the type (which is used as a default value in the map). We also provide predefined instances of these classes, ready to use, for the types manipulated by SMTCoq. This way there is no need to prove that *e.g.,* $\mathbb{Z}$ has a total order, is inhabited and has a decidable equality, every time we want to reason about arrays with integer indices. Similarly, we show that if both the elements (say of type `e`) and keys (say of type `k`) enjoy these properties then so does the type `farray k e`. This allows one to effectively reason in Coq about arrays of arrays of … (or multi-dimentional arrays) and use the checkers that we built in a transparent manner.

In the semantics of SMT-LIB 2, the standard language of SMT , arrays associate any key with an element. This means that the function `select` (also called get, or read) is *total*, as opposed to the semantic given by partial maps (the function `find` in Coq returns an `option elt`). In our interpretation, the function `select` is defined as below, *i.e.* if the key `i` is in the map then it returns the element to which it is associated, otherwise it returns the default value (given by the fact that `elt` is inhabited).

```
Definition select {key} {elt} (a: farray key elt) (i: key) : elt ≜
  match find i a with
  | Some v ⇒ v
  | None ⇒ default_value
  end.
```

This interpretation is consistent with the semantic of the theory of functional arrays with extensionality, but only on the *quantifier-free* fragment. In particular it realizes the three following axioms of the theory of arrays. In the following we use the notation $a[i]$ for select(a, i) and $a[i \leftarrow v]$ for store(a, i, v), these notations can be combined/chained to improve legibility.

$$\begin{aligned} \forall\ a\text{: farray key elt},\ i\text{: key},\ v\text{: elt}. \quad & a[i \leftarrow v][i] = i \\ \forall\ a\text{: farray key elt},\ i, j\text{: key},\ v\text{: elt}. \quad & i \neq j \implies a[i \leftarrow v][j] = a[j] \\ \forall\ a, b\text{: farray key elt}. \quad & (\forall\ i\text{: key}.\ a[i] = b[i]) \implies a = b \end{aligned}$$

This last axiom is known as extensionality (of equality) over arrays.

Remark.

Extensionality as defined in is expressible and provable in an intuitionistic setting (the one of Coq). In fact, the corresponding lemma is proven without any additional axioms.

```
Lemma extensionality : ∀ a b, (∀ i, a[i] = b[i]) → a = b.
```

However it is not the case for the form of extensionality that we need to represent proofs of CVC4 . The following, more useful lemma requires *classical axioms* to be proven.

```
Require Import Classical_Pred_Type ClassicalEpsilon.

Lemma extensionality2 : ∀ a b, a ≠ b → (∃ i, a[i] ≠ b[i]).
```

Notice that this not an issue in itself because SMT solvers use classical logic already. The corresponding rule in LFSC goes even as far as to provide a Skolem constant for the index at which the two arrays differ. This is also possible in Coq, using classical axioms again:

```
Definition diff_index_p : ∀ a b, a ≠ b → { i | a[i] ≠ b[i] } ≜ …
```

This helper function is used to define a function `diff` which returns the specific index at which `a` and `b` differ. One can then easily prove the following lemma that says that when two arrays are indeed distinct, then they differ in particular at the index returned by `diff`. (Notice that `diff` is a total function.)

```
Lemma select_at_diff: ∀ a b, a ≠ b → a[diff a b] ≠ b[diff a b].
```

Extending the structures of formulas of SMTCoq with array types and operations requires a similar level of modification as the one described in Section 4.2.3 for bit vectors. The main difference is that the representation of types is now recursive, and the different interpretations need these total ordering properties inside the definitions themselves.

The checker for the theory of arrays can handle three rules, one for each of the axioms presented in this section. The proof of the correctness of these checkers rely directly on the properties of the underlying interpretation we provide through the use of our library.

4.2.5 Proof Holes

CVC4 is currently not fully instrumented for proof production. In particular, a large number of internal term simplification rules (used to speed up reasoning) do not have yet corresponding proof rules. This means that some LFSC proofs can contain *holes*, *i.e.* proof steps which are not justified. Most of the time they can be easily identified in the proof object as applications of the (unsound) rules `trust` or `trust_f`.

```
(declare trust (th_holds false))
(declare trust_f (! f formula (th_holds f)))
```

The rule `trust` is clearly unsound as it allows one to derive false from any context. Rule `trust_f` is unsound as well as it allows one to derive that any formula at all. Although unsound, these rules offer the possibility to progressively build proof production support in a solver, allowing developers to *temporarily* omit justifications for some reasoning steps and fill them in later.

Holes in CVC4 Proofs

Currently, proofs generated by CVC4 have holes corresponding to any pre-processing step performed on the original (input) formulas of the problem. Pre-processing in SMT solvers is a crucial step (although very much heuristic) which greatly influence performance of the latters. It consists in replacing (or rewriting) sub-terms of formulas of the original problem by equivalent versions. Our translation to the proof format of SMTCoq uses holes in a similar fashion.

As shown in Section 4.2.2, the translation only works for a predefined set of rules and for proofs of a certain shape (the one produced by CVC4). If the translator encounters an unsupported rule, it will be replaced by a hole in the SMTCoq proof. This allows a loser coupling between the tools and makes our approach robust with respect to future small variations. For instance, CVC4 uses *rewrite steps* that appear in the proof of some theory lemmas (*e.g.* bit vectors) some of them have a corresponding rule in LFSC but there is at the moment no checker implemented in Coq for those. With our mechanism, these rewrite steps appear as holes and can thus be handled at a later stage outside of SMTCoq.

Supporting Partial Proofs in SMTCoq

SMTCoq was extended to provide a way to add a *hole* in the proof. This rule can have premises, and also requires a proof that its conclusion is a consequence of its premises. When SMTCoq is used as a checker for a proof witness associated to an SMT-LIB 2 problem,

the user is shown a warning message which says that the system has *assumed* the content of the hole to be true (if it is actually used by the proof).

This approach has the advantage of not impeding progress in the proof process even if automated SMT solvers only produce *partial* proofs.

4.2.6 The cvc4 Coq tactic

SMTCoq brings the power of SMT solvers to Coq. This is very useful for Coq user because it provides a level of automation that is greatly lacking in most interactive theorem provers (especially Coq). SMTCoq already provided two tactics built on top of the main Coq checker. The first one, `zchaff` calls the SAT solver ZChaff [27] to handle propositional (or Boolean) goals. The other, `verit` uses the proof producing SMT solver veriT [28] to give users an automated decision procedure for quantifier free goals involving a combination of the theories of equality over uninterpreted functions and linear integer arithmetic. These tactics do not compromise the soundness of Coq (see Figure 16).

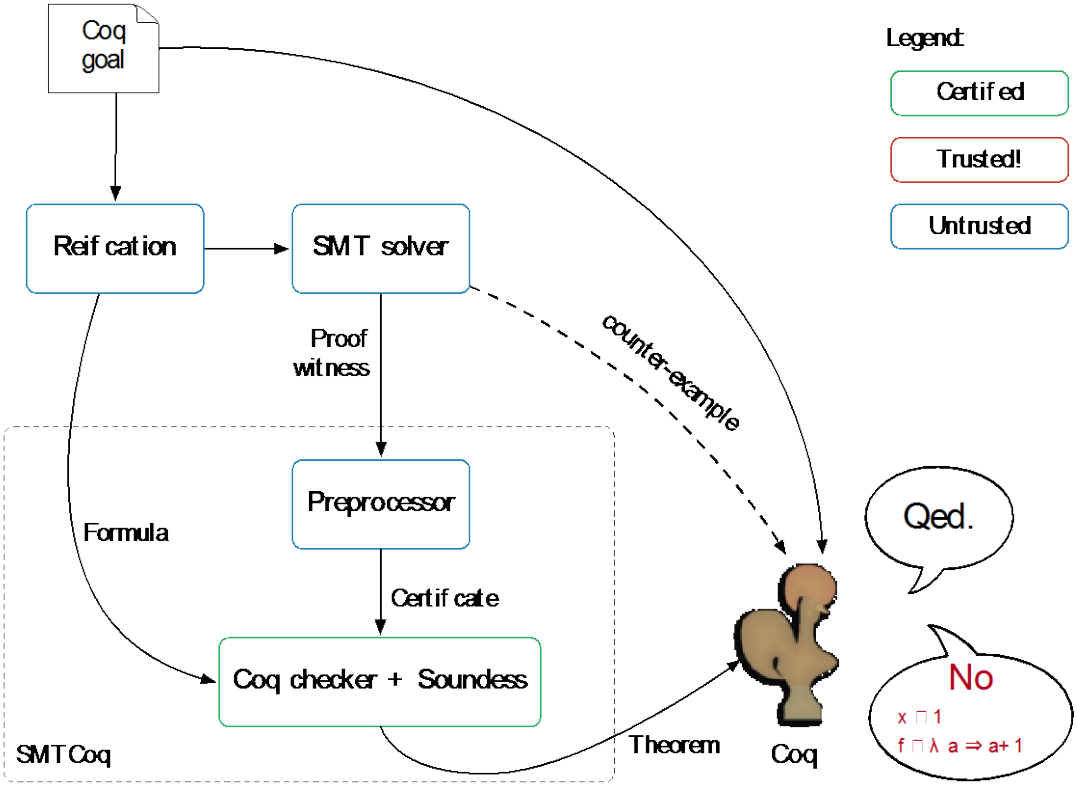

Figure 16 CVC4 tactic in SMTCoq.

We have added a new tactic, `cvc4`, that calls the SMT solver CVC4. This allows to discharge directly in Coq goals in the combination of the theories of equality over uninterpreted

functions, linear integer arithmetic, fixed-width bit vectors and functional arrays with extensionality.

This uses the same process of reification as explained in [34]. Of course, users that wish to use this tactic need to express their goals using our own library for bit vectors and functional arrays. However, we believe this is a reasonable requirement, especially considering the fact that these libraries have been constructed to be generic and easily extensible.

Practical Usage

For a user to be able to use the different SMTCoq tactics, he or she needs to first install the SMTCoq Coq library in a place that Coq can find (or add this path to Coq's load path). Only the first line in the snippet that follows is actually necessary, the other imports are here to provide concise notations for arrays and bit vectors.

```
Require Import SMTCoq Bool List.
Import ListNotations BVList.bit vector_LIST FArray.
Local Open Scope list_scope.
Local Open Scope farray_scope.
Local Open Scope bv_scope.
```

One needs to also have installed the solvers corresponding to the chosen tactic somewhere in their `PATH`. In addition for the `cvc4` tactic we require that the LFSC signatures (distributed with either CVC4 or SMTCoq) be placed in a directory which should be specified in the environment variable `LFSCSIGS`.

The `cvc4` tactic doesn't take any arguments and is to be invoked as any other Coq tactic. Like its sister tactic `verit`, it only works on goals of the form

$$\forall\ x_1\ \ldots\ x_n,\ b_1 = b_2$$

where `b_1` and `b_2` are two Coq terms of type `bool` ($\mathbb{B}$). This tactic can either succeed in proving the goal, fail, or succeed in proving the goal with a number of hypotheses which are then presented to the user as sub-goals to prove. Failure of the tactic can happen in the following scenarios:

- the solver answered `sat`, and can provide a counter-example to the goal;
- the solver crashed;
- the translation of the LFSC proof failed;
- the translated SMTCoq proof witness cannot be checked by the Coq checker (one or several proof steps fail).

Notation.

In the following Coq examples we use the notation:

```
Infix "⟶" ≜ implb (at level 60, right associativity) : bool_scope.
```

for Boolean implication. The symbols ∵ and □ stand for the vernacular Coq commands `Proof.` and `Qed.` respectively.

Examples

For instance, one can prove the following goal involving only Boolean reasoning with a single application of the Coq tactic `cvc4`.

```
Goal ∀ a, negb (a || negb a) = false.
∵
  cvc4.
■
```

In a similar fashion, the following goal involving arrays, uninterpreted functions and linear integer reasoning can also be solved by our tactic.

```
Goal ∀ (a b: farray Z Z)
       (v w x y: Z)
       (g: farray Z Z → Z)
       (f: Z → Z),
       equal a[x ← v] b && equal a[y ← w] b  ⟶
       Z.eqb (f x) (f y) || Z.eqb (g a) (g b).
∵
  cvc4.
■
```

This next example makes use of arrays indexed by bit vectors of size four. This is a typical example of the kind of reasoning that can be done when performing proof of programs manipulating the heap (often represented as arrays) and machine integers (represented as bit vectors).

```
Goal ∀ (bv1 bv2 : bitvector 4)
       (a b c d : farray (bitvector 4) Z),
    bv_eq #b|0|0|0|0| bv1  ⟶
    bv_eq #b|1|0|0|0| bv2  ⟶
    equal c b[bv1 ← 4]  ⟶
    equal d b[bv1 ← 4][bv2 ← 4]  ⟶
    equal a d[bv2 ← b[bv2]]  ⟶
    equal a c.
∵
    cvc4.
■
```

4.3 Evaluation

For evaluation purposes we implemented our proof generation approach in CVC4 [20]. 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($\mathcal{T}$) engine as described in Section 3.2, 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 3.3, 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.

We first compared the lazy and eager proof generation approaches for T_{UF} and T_{AX}. Figure 17 shows the results on all QF_UF and QF_AX benchmarks from the SMT-LIB library [16]. 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.

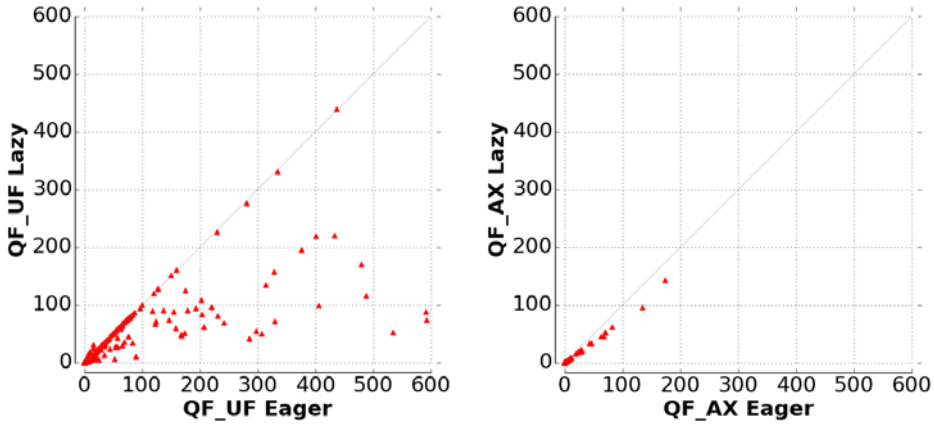

Figure 17 Eager vs. Lazy proof production runtimes, in seconds.

Table 2 Producing and checking proofs. All times are in seconds. Experiments were run with a 600 second timeout.

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

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 [16]: QF_UF, QF_AX, QF_BV , QF_UFBV, QF_ABV and QF_AUFBV. Table 2 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.

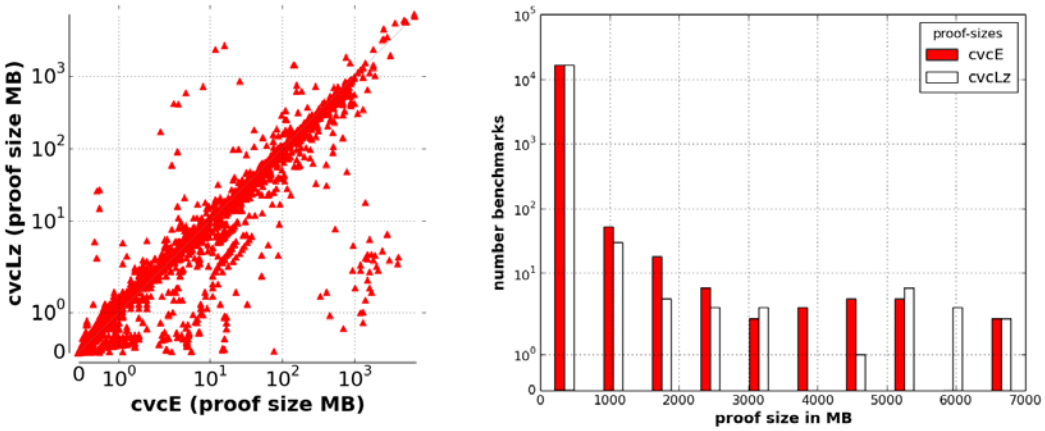

Figure 18 Proof sizes both cvcLz *and* cvcE.

Next, we selected all of the 17,172 unsatisfiable QF_BV benchmarks used in the 2015 SMT-COMP competition and evaluated the overhead of proof generation for both the lazy and the eager configurations of CVC4. CVC4 is a competitive bit-vector solver that placed second in the QF_BV division of the 2015 SMTCOMP by running and in parallel.[14] The proof generated by uses the same proof signature as but has a single monolithic resolution proof as opposed to the modular two-tiered structure of proofs.

Table 3 Overhead of proof generation and its impact on the number of problem solved.

	default		+log			+log+proof			+log+proof+check		
	solved	time (s)	solved	time (s)	%	solved	time (s)	%	solved	time (s)	%
cvcLz	16665	38575	16663	43684	11	16662	43729	14	14063	118544	973
cvcE	16601	65009	16583	78187	19	16582	78256	22	13734	137931	737

Table 3 shows the results for both solvers. We ran the following configurations: solving with proof generation disabled (default); solving with proofs enabled (i.e., the solver logs the information needed to produce the proof) but without actually producing proofs (+log); solving with proof generation including writing the proof object to disk (+log + proof); and solving with proof generation as well as proof checking (+log + proof + check). For the lazy solver cvcLz, the overhead of proof logging results in 2 fewer problems solved while adding an 11% overhead to solving time.[15] The additional overhead of stitching the proof together and outputting it to a file is only 3% of the solving time. For the eager solver cvcE, proof logging adds a higher overhead of 19% and solves 18 fewer problems than the default configuration of cvcE. The overhead of proof generation is higher for the eager solver than for the lazy one.

To ensure the correctness of the proofs we generated, we checked them using our LFSC proof checker. Within the 600 sec time limit, we were able to succesfully check 84% of the problems we could solve with and 82% of the ones solved with cvcE. Proof checking failed due to unsupported proof steps in our generated proof for 33 problems attempted by cvcLz, and for 92 attempted by cvcE. The other failures in proof checking were due to timeouts: proof checking is an order of magnitude slower than solving. We believe that with additional work on the LFSC proof checker, this can be improved.

Despite the slow checking times, we achieve higher proof checking rates for QF_BV than the proof reconstruction approach in Böhme et al. [23]. In that work, proofs could be produced for 735 of the 1377 QF_BV benchmarks available at the time. Out of these, the produced proofs were successfully checked only for 38.5% of the total; 48.4% timed out and 13.1% produced errors. The authors attribute the timeouts to the long time taken to reprove

[14] CVC4 solved 26001 problems in that division compared to 26260 problems solved by the winning solver, Boolector [35].

[15] Overhead in each column is measured by comparing the time taken to solve only those problems solved by *both* the default and the column configuration.

large-step Z3 inferences. Our experimental results indicate that fine-granularity bit-vector proofs enable proof checking for a significantly larger number of problems.

Finally, we compared the sizes of the proof files generated. Figure 18 (left) is a log-scale scatter plot comparing the sizes of the proofs generated by the two solvers. Overall, the proofs generated by the two-tiered lazy approach are smaller: adding the sizes of all the lazy generated proofs results in 276GB while for the eager solver it is 328GB. Figure 18 (right) shows, with the y-axis in log-scale, the distribution of the proof sizes over the benchmark selection. The majority of the benchmarks have relatively small proofs, well under 1GB.

We also evaluated [16] SMTCoq and its extensions on a set of around 500 benchmarks taken from the categories QF_AUFLIA of the SMT-LIB repository. Excluding the ones for which CVC4 fails to return a proof in less than 120 seconds, the ones where our version of Coq crashed due to a segmentation fault and the ones on which CVC4 crashes during its proof production phase, we are left with 251 successful experiments. Out of those, we get 240 proof files automatically checked by SMTCoq and 11 proofs rejected. The investigation on rejected proof certificates pointed us the failure of the micromega based checker which is used in SMTCoq to solve goals in linear integer arithmetic.

Table 4 shows the average run times (in seconds) for those accepted (240 files) together with the average number of holes that are left unproven.

Table 4 SMTCoq's experiments in QF_AUFLIA

CVC4 run time	Type-checking LFSC proof	Converting LFSC proof to SMTCoq format	Coq Checker runtime	Total	number of holes
0.089	0.267	0.167	1.760	2.283	1.055

We have split the times to accurately measure the time necessary for *type checking* the LFSC proof (by our own OCaml LFSC type checker), the time needed to convert the LFSC proof to and SMTCoq certificate, and finally the time for the certified Coq checker to check the certificate. As shown, the total average run time of the certified Coq LFSC checker is around 2.3 seconds which we think is acceptable especially for SMT-LIB benchmarks. Notice also that 1 hole which in turn generate an additional sub-goal (in average) is left unproven which is simply due to CVC4 's pre-processing step where no proof is generated.

[16] These experiments have been performed on an Intel i7-3630QM @2.40GHz machine with 8 GB memory running Ubuntu 16.04 LTS. Here are the software versions:

- CVC4 version at https://github.com/CVC4/CVC4/tree/edce1662b001dd6f229a25685fb4de6789ff008d
- Coq-8.5pl2
- SMTCoq version at https://github.com/LFSC/smtcoq/tree/v1.3-darpa

The SMT-LIB benchmarks for the theories QF_ABV and QF_AUFBV are simply too large to be checked by the SMTCoq checkers but we anticipate that uses of the `cvc4` tactic inside coq will be done on smaller goals as is generally the case with traditional Coq developments. Instead of experimenting on them, we have generated (by hand) 20 benchmarks mixing the theory of bit vectors, functional arrays and linear integer arithmetic and get all corresponding proofs certified by SMTCoq, with the following run times in seconds:

Table 5 SMTCoq's experiments in logic QF_AUFBVLIA

CVC4 run time	Type-checking LFSC proof	Converting LFSC proof to SMTCoq format	Coq Checker runtime	Total	number of holes
0.031	0.028	0.011	0.380	0.450	0.9

One point to notice here is that the `cvc4` tactic can get inefficient (taking more than a couple of seconds to respond) when the proof involves application of the rule for bit-blasting multiplication over bit vectors of size greater than 16 bits [17].

We have performed tests on a large variety of benchmarks the combination of the mentioned theories. We are confident that the `cvc4` tactic will manage to automatize a good number of goals as long as our bit vector library and our version of the functional arrays library are used.

4.4 Related Work

Various SMT solvers have taken different approaches to proof production over the years (see Barrett *et al.* [14] 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 [28], 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 [21]. However, veriT does not support lazy proof production or proofs for T_{AX} or T_{BV}.

The LFSC meta-framework has been successfully used for encoding proofs generated by SMT solvers for other theories in [18], [36], [37]. The current work extends this line of work to support LFSC proofs for the bit-vector theory. In [1] the authors show how to use LFSC to compute interpolants from unsatisfiability proofs in the theory of equality and uninterpreted function symbols. We believe this approach can be extended to generate bit-vector interpolants from LFSC bit-vector proofs. Other approaches for checking SMT-generated proofs include using custom checkers [38] or skeptical proof assistants based on higher-order logic [21], [39]–[41]. These approaches are based on translating SAT/SMT

[17] This same remark can also be made for SMT solvers that do bit-blasting of bit vector operators but on much smaller scale.

certificates to applications of the inference rules of the kernels of these proof assistants. In contrast, our approach in Coq is based on computational reflection: the certificate is directly processed by the reduction mechanism of Coq 's kernel.

Heule *et al.* implemented an efficient checker for state-of-the-art SAT techniques, verified in ACL2 [42], [43]. It is mainly based on a generalization of extended resolution [44], [45] and on reverse unit propagation [30]. SMTCoq currently handles only standard extended resolution for its propositional part.

The work whose scope is most similar to ours is an effort that was undertaken to reconstruct bit-vector proofs produced by Z3 within Isabelle/Hol [23]. The main difference in that work is that Z3 does not produce full proofs, but rather "proof sketches," essentially a record of propositional inferences plus a listing of theory lemmas used [15]. Specifically, Z3 provides some "large-step" inferences, lemmas that are valid in the theory of bit-vectors, without proof. As the authors remark, the coarse granularity of Z3's proofs makes proof reconstruction particularly challenging. A significant part of the proof checking time is spent re-proving large-step inferences that Z3 does not provide details for. In contrast, our approach is more fine-grained as it provides full details for every step. As we have shown, this enables our approach to check more proofs.

Based on an efficient encoding of a large subset of HOL goals into first-order logic, the Sledgehammer tactic [46] allows HOL-based proof assistants to efficiently and reliably help manual proving. Proofs are replayed using either the proof reconstruction mechanism described above or a built-in first-order prover. We hope that SMTCoq can help in adding such techniques into Coq and other Type Theory-based proof assistants, by providing a proof replay mechanism based on certificates.

5 Conclusion

Adding proof production capabilities to complex tools like SMT solvers can greatly increase our level of confidence in their results. We presented a technique that allows DPLL($\mathcal{T}$) -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}.

In the future, we plan to improve CVC4's ability to prove rewrite steps, as discussed in Section 3.3. 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.

SMTCoq has been designed to be modular in such a way that facilities its extension with new solvers and new theories. In particular, such extensions should not require any changes in existing checkers or in their proofs of soundness. Thus, while it may require some effort to certify new small checkers or to translate new proof formats into the SMTCoq format, such extensions require only local changes. Our current extensions to CVC4, bit-vector arithmetic and the theory of functional arrays validate this goal: indeed, the work so far covered mostly in implementing an untrusted preprocessor for certificates and adding new, independent checkers (see Table 6 for SMTCoq's list of features). One limiting aspect of SMTCoq is the lack of support for nested proofs, which we plan to add. Thanks to the modularity of the checker, we believe this feature too can be added locally.

Table 6 Support for solvers and theories in SMTCoq.

	SAT solver	SMT solvers	
Theory	**zChaff**	**veriT**	**CVC4**
Propositional logic	✓	✓	✓
EUF		✓	✓
Linear integer arithmetic		✓	✓
Bit-vectors			✓
Arrays			✓

We want the `cvc4` tactic to work also on goals in Coq's Prop universe. This will require, in a preprocessing step, to get the Boolean counterpart of a proposition (using SSReflect's reflect predicate [32]) and call the `cvc4` tactic afterwards.

We also intend to provide, in addition to our own bit-vectors library, support for Bedrock, namely words [47]. This will not bring any additional feature to the system but support the goals written in the format of words. To do so, we need to prove the bit-vector checkers once again using the facts of words in Coq.

References

[1] A. Reynolds, C. Tinelli, and L. Hadarean, "Certified interpolant generation for EUF," in *Workshop on satisfiability modulo theories*, 2011.

[2] J. Chen, R. Chugh, and N. Swamy, "Type-Preserving Compilation of End-to-End Verification of Security Enforcement," in *Proc. 10th acm conf. on programming language design and implementation (pldi)*, 2010, pp. 412–423.

[3] 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, vol. 7086, pp. 135–150.

[4] J. Blanchette, S. Böhme, and L. Paulson, "Extending Sledgehammer with SMT Solvers," *Journal of Automated Reasoning*, vol. 51, no. 1, pp. 109–128, 2013.

[5] B. Ekici, G. Katz, C. Keller, A. Mebsout, A. Reynolds, and C. Tinelli, "Extending SMTCoq, a Certified Checker for SMT," in *Proc. 1st int. workshop on hammers for type theories (hatt)*, 2016, pp. 21–29.

[6] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(*T*)," *Journal of the ACM (JACM)*, vol. 53, no. 6, pp. 937–977, 2006.

[7] A. Reynolds, C. Tinelli, A. Goel, and S. Krstić, "Finite Model Finding in SMT," in *Proc. 25th int. conf. on computer aided verification (cav)*, 2013, pp. 640–655.

[8] S. Krstić and A. Goel, "Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL," in *Proc. 6th int. symposium on frontiers of combining systems (frocos)*, 2007, pp. 1–27.

[9] J. Marques-Silva and K. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[10] C. Tinelli and M. Harandi, "A New Correctness Proof of the Nelson—Oppen Combination Procedure," in *Proc. 1st int. symposium on frontiers of combining systems (frocos)*, 1996, pp. 103–120.

[11] 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 (AMAI)*, vol. 55, nos. 1-2, pp. 63–99, 2009.

[12] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Splitting On Demand in SAT Modulo Theories," in *Proc. 13th int. conf. on logic for programming, artificial intelligence, and reasoning (lpar)*, 2006, pp. 512–526.

[13] M. Heule and A. Biere, “Proofs for Satisfiability Problems,” *All about Proofs, Proofs for All*, vol. 55, no. 1, pp. 1–22, 2015.

[14] C. Barrett, L. de Moura, and P. Fontaine, “Proofs in Satisfiability Modulo Theories,” *All about Proofs, Proofs for All*, vol. 55, no. 1, pp. 23–44, 2015.

[15] N. Bjørner and L. de Moura, “Proofs and Refutations, and Z3,” in *Proc. 14th int. conf. on logic for programming, artificial intelligence and reasoning (lpar)*, 2008.

[16] C. Barrett, P. Fontaine, and C. Tinelli, “The Satisfiability Modulo Theories Library (SMT-LIB).” http://www.SMT-LIB.org, 2015.

[17] R. Harper, F. Honsell, and G. D. Plotkin, “A framework for defining logics,” *J. ACM*, vol. 40, no. 1, pp. 143–184, 1993.

[18] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli, “SMT Proof Checking Using a Logical Framework,” *Formal Methods in System Design*, vol. 42, no. 1, pp. 91–118, 2012.

[19] N. Wetzler, M. J. Heule, and W. A. Hunt Jr, “DRAT-trim: Efficient checking and trimming using expressive clausal proofs,” in *Theory and applications of satisfiability testing*, 2014.

[20] C. Barrett, “CVC4,” in *Proc. 23rd int. conf. on computer aided verification (cav)*, 2011, pp. 171–177.

[21] P. Fontaine, J. Marion, S. Merz, L. Nieto, and A. Tiu, “Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants,” in *Proc. 12th int. conf. on tools and algorithms for the construction and analysis of systems (tacas)*, 2006, pp. 167–181.

[22] L. de Moura and N. Bjørner, “Generalized, Efficient Array Decision Procedures,” in *Proc. 9th int. conf. on formal methods in computer-aided design (fmcad)*, 2009, pp. 45–52.

[23] S. Böhme, A. C. J. Fox, T. Sewell, and T. Weber, “Reconstruction of Z3’s Bit-Vector Proofs in HOL4 and Isabelle/HOL,” in *Certified programs and proofs - first international conference, CPP 2011, kenting, taiwan, december 7-9, 2011. proceedings*, 2011, vol. 7086, pp. 183–198.

[24] L. Hadarean, K. Bansal, D. Jovanovic, C. Barrett, and C. Tinelli, “A tale of two solvers: Eager and lazy approaches to bit-vectors,” in *Conference on computer aided verification*, 2014.

[25] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *Theory and applications of satisfiability testing*, 2004.

[26] B. Barras, “The Coq proof assistant: reference manual,” INRIA, 2000.

[27] Y. S. Mahajan, Z. Fu, and S. Malik, “Zchaff2004: An efficient SAT solver,” in *Theory and applications of satisfiability testing, 7th international conference, SAT 2004, vancouver, bc, canada, may 10-13, 2004, revised selected papers*, 2004, vol. 3542, pp. 360–375.

[28] T. Bouton, D. de Oliveira, D. Déharbe, and P. Fontaine, "veriT: An Open, Trustable and Efficient SMT-Solver," in *Proc. 22nd int. conf. on automated deduction (cade)*, 2009, vol. 5663, pp. 151–156.

[29] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Proceedings of the 8th international workshop on satisfiability modulo theories (edinburgh, uk)*, 2010.

[30] A. V. Gelder, "Producing and verifying extremely large propositional refutations - Have your cake and eat it too," *Ann. Math. Artif. Intell.*, vol. 65, no. 4, pp. 329–372, 2012.

[31] F. Besson, P. Fontaine, and L. Théry, "A Flexible Proof Format for SMT: a Proposal," in *PxTP 2011: First international workshop on proof eXchange for theorem proving august 1, 2011 affiliated with cade 2011, 31 july-5 august 2011 wrocław, poland*, 2011, pp. 15–26.

[32] G. Gonthier and A. Mahboubi, "An introduction to small scale reflection in coq," *J. Formalized Reasoning*, vol. 3, no. 2, pp. 95–152, 2010.

[33] F. Besson, "Fast Reflexive Arithmetic Tactics the Linear Case and Beyond," in *TYPES*, 2006, vol. 4502, pp. 48–62.

[34] C. Keller, "A Matter of Trust: Skeptical Communication Between Coq and External Provers," PhD thesis, École Polytechnique, 2013.

[35] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Tools and algorithms for the construction and analysis of systems*, 2009.

[36] A. Reynolds, L. Hadarean, C. Tinelli, Y. Ge, A. Stump, and C. Barrett, "Comparing proof systems for linear real arithmetic with LFSC," in *Workshop on satisfiability modulo theories*, 2010.

[37] D. Oe, A. Reynolds, and A. Stump, "Fast and Flexible Proof Checking for SMT," in *Workshop on satisfiability modulo theories*, 2009.

[38] M. Moskal, "Rocket-Fast Proof Checking for SMT Solvers," in *Proc. 14th int. conf. on tools and algorithms for the construction and analysis of systems (tacas)*, 2008, pp. 486–500.

[39] S. McLaughlin, C. Barrett, and Y. Ge, "Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite," in *Proc. 3rd workshop on pragmatics of decision procedures in automated reasoning (pdpar)*, 2005, pp. 43–51.

[40] T. Weber, "SAT-based Finite Model Generation for Higher-Order Logic," PhD thesis, Institut für Informatik, Technische Universität München, Germany, 2008.

[41] S. Böhme and T. Weber, "Fast lcf-style proof reconstruction for Z3," in *Interactive theorem proving, first international conference, ITP 2010, edinburgh, uk, july 11-14, 2010. proceedings*, 2010, vol. 6172, pp. 179–194.

[42] M. Heule, W. A. H. Jr., and N. Wetzler, “Verifying refutations with extended resolution,” in *Automated deduction - CADE-24 - 24th international conference on automated deduction, lake placid, ny, usa, june 9-14, 2013. proceedings*, 2013, vol. 7898, pp. 345–359.

[43] N. Wetzler, M. Heule, and W. A. H. Jr., “Mechanical verification of SAT refutations with extended resolution,” in *Interactive theorem proving - 4th international conference, ITP 2013, rennes, france, july 22-26, 2013. proceedings*, 2013, vol. 7998, pp. 229–244.

[44] G. Tseitin, “On the Complexity of Proofs in Propositional Logics,” in *Seminars in mathematics*, 1970, vol. 8, pp. 466–483.

[45] O. Kullmann, “On a generalization of extended resolution,” *Discrete Applied Mathematics*, vols. 96-97, pp. 149–176, 1999.

[46] L. C. Paulson and J. C. Blanchette, “Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers,” in *The 8th international workshop on the implementation of logics, IWIL 2010, yogyakarta, indonesia, october 9, 2011*, 2010, vol. 2, pp. 1–11.

[47] A. Chlipala, “Mostly-automated verification of low-level programs in computational separation logic,” in *ACM sigplan notices*, 2011, vol. 46, pp. 234–245.

A Appendix

A.1 Implementation of SMTCoq

This section describes the organization of the SMTCoq repository and locations of source code and modules.

Sources are contained in the directory src which can be found at top-level. The directories examples and unit-tests contain respectively example files of usage for SMTCoq and regression tests for the different tactics and vernacular commands that the plugin provides.

The rest of the section describes the organization of `src`.

A.1.1 Top-level architecture of SMTCoq

SMTCoq sources are contained in this directory. A few Coq files can be found at top-level.

configure.sh

This script is meant to be run when compiling SMTCoq for the first time. It should also be run every time the Makefile is modified. It takes as argument an optional flag `-native` which, when present, will set up the sources to use the *native Coq* libraries. Otherwise the standard version 8.5 of Coq is used.

SMTCoq.v

This is the main SMTCoq entry point, it is meant to be imported by users that want to use SMTCoq in their Coq developments. It provides (exports) the other SMTCoq modules as well as declares the OCaml plugin for adding the new vernacular commands and tactics.

Trace.v

This file defines the types of certificates and steps (atomic certificate pieces) as well as the *main checkers*.

The first section `trace` gives a generic definition of a main checker parameterized by the type of individual steps and a function to check individual steps `check_step` (small checkers). Correctness of the main checker is proved under the assumption that the small checker is correct.

These generic definitions are applied to construct main checkers for resolution (module `Sat_Checker`), CNF conversion (module `Cnf_Checker`) and satisfiability modulo theories (module `Euf_Checker`). They each define an inductive type `step` to represent certificate steps. For instance, in the case of the resolution checker, the only possible step is to apply the resolution rule so steps are defined as:

```
Inductive step ≜
   | Res (_:int) (_:resolution).
```

The main theorems for these modules are named `checker_correct`. For instance the main result for the SMT checker (`Euf_Checker`) is formulated as follows:

```
Lemma checker_correct : ∀ d used_roots c,
    checker d used_roots c = true →
    ~ valid t_func t_atom t_form d.
```

which means that if the checker returns true on the formula `d` and the certificate `c` then `d` is not valid (*i.e.* `c` is a refutation proof certificate for `d`).

State.v

This module is used to define representations for the global state of the checker. A state is an array of clauses:

```
Module S.
  Definition t ≜ array C.t.
...
End S.
```

on which we define resolution chain operations `set_resolve` that modify the state.

Variables, literals and clauses are defined respectively in modules `Var`, `Lit` and `C`. Binary resolution is defined between two clauses in `C.resolve`.

SMT_terms.v

This Coq module defines reification types for formulas (`Form.form`), types (`Typ.type`) and atoms/terms (`Atom.atom`). Formulas are given an interpretation in Coq's Booleans, types are interpreted in Coq types (for instance, type `TZ` is interpreted as Coq's mathematical integers `Z`) and atoms are interpreted as Coq terms of type the interpretation of their type (for instance an atom whose type is `TZ` is interpreted as an integer of `Z`).

Some important lemmas.

A function `cast` allows to change the encoded type of a term of type `Typ.type` when we know two types are equal (the inductive `cast_result` provides the conversion function).

```
Fixpoint cast_refl A:
      cast A A = Cast (fun P (H : P A) ⇒ H).
```

This is the lemma to use to remove cast constructions during the proofs.

```
Lemma i_eqb_spec : ∀ t x y, i_eqb t x y ↔ x = y.
```

This other lemma says that Boolean equality over interpretation of types is the equivalent to Leibniz equality. This is useful to allow rewriting.

Atom (as well as formulas) are encoded by integers, and mapping is preserved by an array `t_atom`. Another array maintains interpretations of encodings. The following lemma states that these two relates:

```
Lemma t_interp_wf : ∀ i,
   t_interp.[i] = interp_aux (PArray.get t_interp) (t_atom.[i]).
```

Misc.v

This module contains miscellaneous general lemmas that are used in several places throughout the development of SMTCoq.

versions

This directory contains everything that is dependent on the version of Coq that one wants to use. `standard` contains libraries for the standard version of Coq and `native` contains everything related to native Coq. Note that some libraries are already present in the default libraries of native Coq, in this case they have a counterpart in `standard` that replicates the functionality (without using native integers or native arrays).

A particular point of interest is the files smtcoq_plugin_standard.ml4 and smtcoq_plugin_native.ml4. They provide extension points for Coq by defining new vernacular commands and new tactics. For instance the tactic `verit` tells Coq to call the OCaml function `verit.tactic` (which in turns uses the Coq API to manipulate the goals and call the certified checkers).

```
TACTIC EXTEND Tactic_verit
| [ "verit" ] -> [ Verit.tactic () ]
END
```

spl

This directory contains everything related to simplifications of input formulas as well as the Coq machinery to handle step checkers that use assumptions (and generate sub-goals).

- Arithmetic.v: Arithmetic simplifications
- Operators.v: Simplifications of SMT-LIB 2 operators (atomic disequalities and distinct operators)
- Syntactic.v: Flattening and normalization of propositional structures
- Assumptions.v: Small checker for assumptions

extraction

This is the extracted version of the SMTCoq checker, that can be run outside Coq. It still needs to be fixed for the new additions and extensions.

classes

The definitions of interpretations of terms and types of SMTCoq requires some additional constraints that are encoded as Coq type-classes. This directory contains definitions and properties of these classes SMT_classes.v as well as predefined useful instances of these classes SMT_classes_instances.v.

These classes are:

- `EqbType`: types with a Boolean equality that reflects in Leibniz equality
- `DecType`: types with a decidable equality
- `OrdType`: class of types with a partial order
- `Comparable`: augmentation of class of partial order with a compare function to obtain a total order
- `Inhabited`: class of inhabited types (used to obtain default values for types)
- `CompDec`: a class that merges all previous classes

A.1.2 Small checkers

Small Coq checkers are organized in sub-directories that reflect the theories they handle. Small checkers for propositional logic, equality over uninterpreted functions and linear integer arithmetic all use preexisting standard Coq libraries (Bool, Arith, Z, BinPos, ...) to formalize the underlying interpretation of these theories. The theories of fixed-width bit-vectors and functional unbounded arrays are formalized in new custom Coq libraries (that are distributed with SMTCoq).

Computational small checkers have the following signature:

```
Definition checker (s : S.t) (p1 ... pn : int) (l1 ... lm : lit) : C.t ≜ ...
```

where `s` is the state of the main checker, `p1`, ..., `pn` are positions (there can be none) of deduced clauses that appear in the state `s` and `l1`, ..., `lm` are literals. The function `checker` returns a clause that is `deducible` from the already deduced clauses in the state `s`.

It states that the clause returned by `checker` is valid. In most cases for the small checkers, when they fail they return a trivially true clause (`C._true`).

cnf

Small checkers for CNF (conjunctive normal form) are defined in the module Cnf.v. In essence they implement a Tseitin conversion.

For instance, the checker `check_BuildDef` returns a tautology in clausal form (the validity of the clause is not dependent on the validity of the state) and the checker `check_ImmBuildDef` is a generic encoding of conversion rules that have a premise (which appears in the state).

euf

The checkers for EUF (equality over uninterpreted functions) are defined in the module Euf.v.

The first one checks application of the rule of transitivity. `check_trans` takes as argument the result of the rule application as well as list of equalities of the form `a = b`, `b = c`, …, `x = y`, `y = z`.

The other checker takes care of applications of the congruence rule. Functions in SMT-LIB have a given arity and they are interpreted as Coq functions. The checker for congruence can check rule applications with a number of equalities corresponding to the arity of the function.

lia

Checking linear arithmetic lemmas that come from the SMT solver is performed using the already existing `Micromega` solver of Coq. The corresponding checker is implemented in module Lia.v.

bva

The small checkers for bit-vector operations can be found in module Bva_checker.v. They implement the rules for bit-blasting operators of the theory of fixed width bit-vector.

There are small checkers for:

- bit-wise operators (`bvand`, `bvor`, `bvxor`, `bvnot`)
- equality
- variables
- constants
- extraction
- concatenation

- arithmetic operations (addition, negation, multiplication)
- comparison predicates (signed/unsigned)
- extensions (zero/signed)

The theory of fixed width bit-vectors is realized by an implementation provided in BVList.v. There, bit-vectors are interpreted by lists of Booleans. The type of bit-vectors is a dependent type:

```
Parameter bitvector : N → Type.
```

In the implementation, a bit-vector is a record that contains a list of Booleans `bv`, *i.e.* the lists of its bits, as well as a proof of well formedness `wf`, *i.e.* a proof that the size of the list `bv` is the parameter `n` of the type.

```
Record bitvector_ (n:N) : Type ≜
    MkBitvector
      { bv :> M.bitvector;
        wf : M.size bv = n
      }.
```

array

The theory of unbounded functional arrays with extensionality is realized in Coq by a custom type that can be found in FArray.v.

```
Definition farray (key elt : Type) _ _  ≜
```

The type `farray` is parameterized by the type of keys (or indexes) of the array and the type of the elements. `key` must be a type equipped with a partial order and `elt` must be inhabited.

```
Record slist ≜
  {this :> Raw.farray key elt;
   sorted : sort (Raw.ltk key_ord) this;
   nodefault : NoDefault this
   }.

Definition farray ≜ slist.
```

An array is represented internally by an association list for its mappings with additional constraints that encode the fact that the list is sorted and that there are no mapping to the default value.

This library also provides useful properties on these arrays. Notably extensionality which is required by the theory of arrays in SMT solvers:

```
Lemma extensionality : ∀ a b, (∀ i, select a i = select b i) → a = b.
```

The extensionality rule that is used by the checker is a bit different and requires classical axioms to be proven. This is done in section `Classical_extensionality` which provides an alternative version without contaminating uses of the library.

There are three small checkers for arrays. They check application of the axioms (in the theory sense) of the theory of arrays, two for *read over write* and one for *extensionality*

A.1.3 OCaml implementation of the plugin

Part of SMTCoq are implemented in OCaml. This concerns functionalities which are not certified such as the reification mechanism, the parsers, pre-processors and the definitions of tactics.

This part communicates directly with Coq by using the OCaml Coq API.

trace

This directory contains the implementation of certificates and the representation of SMT-LIB formulas in SMTCoq.

- coqTerms.ml contains imports from Coq of terms to be used directly in OCaml. These include usual Coq terms but also ones specific to SMTCoq.
- smtAtom.mli contains the definitions for the types of atoms in SMTCoq but also provides smart constructors for them. The modules defined in this file have functions to reify Coq terms in OCaml and to translate back OCaml atoms and types to their Coq counterpart interpretation.
- smtForm.mli plays the same role as `smtAtom` but on the level of formulas.
- smtCertif.ml contains definitions for an OCaml version of the steps of the certificate. These are the objects that are constructed when importing a certificate from an SMT solver for instance.
- smtTrace.ml contains functions to build the Coq version of the certificate from the OCaml one.
- smtCommands.ml constitute the bulk of the implementation of the plugin. It contains the OCaml functions that are used to build the Coq vernacular commands (`Verit_checker`, `Lfsc_checker`, …) and the tactics. It also contains functions to reconstruct Coq counter-examples from models returned by the SMT solver.
- smtCnf.ml implements a CNF conversion on the type of SMTCoq formulas.
- smtMisc.ml contains miscellaneous functions used in the previous modules.

smtlib2

This directory contains utilities to communicate directly with SMT solvers. This includes a lexer/parser for the SMT-LIB 2 format (smtlib2_parse.mly) a conversion module from SMT-LIB 2 to formulas and atoms of SMTCoq (smtlib2_genConstr.ml) and a way to call and communicate with SMT solvers through pipes (smtlib2_solver.mli).

zchaff

Files in this directory allow to call the SAT solver ZChaff. It contains a parser for the sat solver input files and ZChaff certificates. The implementation for the Coq tactic `zchaff` can be found in zchaff.ml.

verit

This directory contains the necessary modules to support the SMT solver veriT. In particular it contains a parser for the format of certificates of veriT (veritParser.mly) and an intermediate representation of those certificates (veritSyntax.mli). This module also implements a conversion function from veriT certificates to SMTCoq format of certificates. This pre-processor is a simple one-to-one conversion.

The file (verit.ml) contains the functions to invoke veritT and create SMT-LIB 2 scripts. This is used by the definition of the tactic `verit` of the same file.

lfsc

This directory contains the pre-processor for LFSC proofs to SMTCoq certificates (as well as veriT certificates). The files ast.ml and builtin.ml contain an OCaml implementation of a type checker for LFSC proofs. This directory also contains a parser and lexer for LFSC (*c.f.*, lfscParser.mly).

The pre-processor is implemented in the module converter.ml) as a *functor*. Depending on the module (for terms and clauses conversions) that is passed in the functor application, we obtain either a pre-processor from LFSC proofs to SMTCoq certificates directly or a converter from LFSC proofs to veriT certificates.

Note.

A standalone version of the converter can be obtained by issuing `make` in this directory. This produces a binary `lfsctosmtcoq.native` that can be run with an LFSC proof as argument and produces a veriT certificate on the standard output. Finally, the tactic `cvc4` is implemented in the file lfsc.ml. It contains functions to call the SMT solver CVC4, convert its proof and call the base tactic of `smtCommands`.

List of Symbols, Abbreviations, and Acronyms

- Σ - Used to denote a signature in many-sorted first order logic.
- $\bullet$ - A decision point in a context.
- $\overline{l}$ - The logical complement of l.
- $l \prec_{\mathsf{M}} l'$ - l occurs before l' in M.
- $\models_i$ - Denotes validity in the theory T_i.
- $a[i]$ - The result of reading an array a at index i.
- $a[i] := b$ - The result of writing value b at index i of a.
- $\mathcal{C}$ - A set of constant symbols.
- CDCL - Conflict-directed Clause-learning. Refers to a modern algorithm for Boolean satisfiability.
- Coq - A skeptical proof assistant - see https://coq.inria.fr.
- CNF - Conjunctive Normal Form.
- CVC4 - An open-source SMT solver available at http://cvc4.cs.nyu.edu.
- DIMACS - A textual format for expressing Boolean satisfiability problems.
- DPLL - The Davis-Putnam-Logemann-Loveland algorithm, a complete algorithm for Boolean satisfiability.
- DPLL($\mathcal{T}$) - An architecture for SMT solvers in which a DPLL-based SAT solver interacts with a theory solver for theory $\mathcal{T}$.
- EUF - Equality with Uninterpreted Functions.
- $\texttt{explain}_i$ - A function provided by a theory solver which maps a propagated literal to a valid clause responsible for the propagation.
- fail - A distinguished abstract state signifying unsatisfiability.
- lev - A funciton mapping each literal of M to the unique decision level in which it occurs.
- Lit - A function which returns all of the literals in its argument and their complements.
- $\text{Lit}_{\mathsf{M}}|_i$ - The Σ_i-literals of Lit_{M}.

- Int_{M} - The set of all interface literals of M: the equalities and disequalities between shared constants, where the set of shared constants is $\{c \mid \text{constant } c \text{ occurs in } \text{Lit}_{\text{M}}|_i \text{ and } \text{Lit}_{\text{M}}|_j, \text{ for some } 1 \leq i < j \leq m\}$
- LF - The Edinburgh Logical Framework. A proof format.
- LFSC - Logical Framework with Side Conditions. An extension of LF that supports computational side conditions.
- $\texttt{provideProof}_i$ - A function provided by a theory solver which maps a theory lemma to a proof for the theory lemma.
- M_i - The i'th decision level.
- $M^{[i]}$ - The subsequence $M_0 \bullet \cdots \bullet M_i$.
- $\langle M, F, C \rangle$ - An abstract state consisting of the context M, a set F of clauses, and a set C containing conflicts.
- QF_ABV - The SMT-LIB logic for quantifier-free array and bit vector formulas.
- QF_AUFBV - The SMT-LIB logic for quantifier-free array and uninterpreted functions and bit vector formulas.
- QF_AX - The SMT-LIB logic for quantifier-free array formulas.
- QF_BV - The SMT-LIB logic for quantifier-free bit vector formulas.
- QF_UFBV - The SMT-LIB logic for quantifier-free uninterpreted function and bit vector formulas.
- QF_UF - The SMT-LIB logic for quantifier-free uninterpreted function formulas.
- **S** - A set of sort symbols.
- SAT - Boolean satisfiability.
- SMT - Satisfiability Modulo Theories.
- SMT-COMP - The SMT competition, held annually (see www.smtcomp.org).
- SMT-LIB - The Satisfiability Modulo Theories benchmark library.
- SMTCoq - A tool that translates SMT proofs into Coq proofs.
- Z3 - An SMT solver developed at Microsoft Research.
- ZChaff - A SAT solver developed at Princeton University.
- T - Used to represent a generic logical theory.
- T_{AX}- The theory of arrays.

- T_{ABV} - The theory of arrays combined with the theory of fixed-width bitvectors.
- T_{BV}- The theory of fixed-width bitvectors.
- T_{LIA}- The theory of linear arithmetic over the integers.
- T_{LRA}- The theory of linear arithmetic over the reals.
- T_{UF}- The theory of equality with uninterpreted functions.
- veriT - An SMT solver developed at INRIA, France.