Formal Methods in Computer-Aided Design 2024

Extending DRAT to SMT

S Hitarth *¶, Cayden Codel †, Hanna Lachnitt ‡, and Bruno Dutertre §
*Hong Kong University of Science and Technology, Hong Kong
¶IMDEA Software Institute, Madrid, Spain
Email: hitarth.singh@connect.ust.hk
†Carnegie Mellon University, Pittsburgh, PA, USA
Email: ccodel@cs.cmu.edu
‡Stanford University, Stanford, CA, USA
Email: lachnitt@stanford.edu
§Amazon Web Services, Santa Clara, CA, USA
Email: dutebrun@amazon.com

***Abstract*—The soundness of Satisfiability Modulo Theories (SMT) solvers is critical in many applications. One way to ensure soundness is to have solvers generate proofs that can be independently verified. Unfortunately, generating proofs can have a significant overhead. We propose a new proof format (*e*DRAT) that extends the well-known DRAT format from SAT to SMT. *e*DRAT proofs can be generated with little overhead and can be verified by combining existing tools for propositional reasoning with specialized theory checkers. We instrument the CVC5 solver to generate *e*DRAT proofs and we develop checkers for two SMT theories. Our checkers include an untrusted elaborator written in Rust and a formally verified component written in Lean that validates results from the elaborator. Empirical evaluation shows that *e*DRAT has a much lower proof generation overhead than other formats supported by CVC5, and it has comparable or better proof checking times.**

I. Introduction

Satisfiability Modulo Theories (SMT) solvers are used as back-ends in a variety of applications including software verification and testing [24], [18], [19], the verification of distributed systems [26], model checking [20], [9], [8], and security policy analysis [2], [29]. The soundness of SMT solvers is critical for these applications, especially because SMT solvers have become increasingly complex over the years and are therefore subject to bugs.

When an input formula is satisfiable, solvers can produce a model that can generally be checked, but this approach is not applicable when the SMT solver says that the input formula is unsatisfiable. To increase trust in the unsat case, the SMT community has developed solvers that generate a proof that can be independently validated by a trusted checker. Solvers such as CVC5 [3], OpenSMT [23], SMTInterpol [10], veriT [7] and Z3 [12] can produce proofs, for at least some of the logical theories they support. Some have had proof support for many years.

Several proof formats have been proposed for SMT [33], [30], [22], but none has emerged as a standard. One limitation of these formats is that they require fine-grained proofs with small inference steps. While this simplifies proof checking, generating such detailed proofs is expensive and slows down solvers, and the resulting proofs can be very large and slow to validate.

To address these concerns, we propose *extended DRAT* (*e*DRAT), a new SMT proof format that extends DRAT [21], a standard proof format for Boolean satisfiability. Proofs in *e*DRAT are coarse-grained and clausal. They include Boolean resolution steps (as in DRAT and its predecessor DRUP) and SMT-specific clauses called *theory lemmas*.

Along with *e*DRAT, we present VALIDO, a modular and extensible toolchain for checking *e*DRAT proofs. Proof checking with VALIDO is a two-step process. First, we validate the propositional part of the proof with `DRAT-trim` [34] to extract an unsat core. Second, we check that all the theory lemmas in the unsat core are valid using theory-specific checkers. Currently, VALIDO supports two SMT theories: QF_LRA and QF_UF. The VALIDO theory checkers for these two theories have two components:

- An *elaborator* does most of the heavy lifting. It validates theory lemmas using theory-specific decision procedures and generates an *unsatisfiability certificate* for the negation of each lemma.
- A *certificate validator* checks the unsatisfiability certificates produced by the elaborator.

The validator is the only trusted component, as the validity of a certificate is enough to ensure that a lemma is valid, irrespective of how the certificate was generated. To achieve a high degree of confidence in the correctness of our toolchain, we use the Lean theorem prover [14] to develop and prove the soundness of our validators.

Because theory lemmas are individually validated, we can precisely identify incorrect lemmas when proof validation fails. This can aid debugging and guide the search for a minimal counterexample.

We have instrumented the CVC5 solver to generate *e*DRAT proofs, and we have evaluated VALIDO on SMT-LIB benchmarks. Empirical results show that *e*DRAT proof generation has low overhead (less than 10%), as opposed to between 2x and 17x for two other formats supported by CVC5. *e*DRAT proofs are generally smaller, and proof checking time is comparable to or better than with the alternative proof formats.

Our toolchain can generate eDRAT proofs for any theory that cvc5 supports, including theories with quantifiers and theory

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_8

combinations. Our current proof-checking pipeline, VALIDO, is only implemented for QF_UF and QF_LRA. However, validating a theory lemma boils down to solving a (small) SMT problem. For instance, we could leverage CVC5 itself (or some other proof-producing solver) as an elaborator and the associated ALF/LFSC proof checker as the validator. Therefore, our approach is quite general and not limited to simple theories.

A limitation of *e*DRAT is that it requires problem instances to be in conjunctive normal form (CNF), while SMT problems can be arbitrary formulas. The preprocessing, simplification, and rewriting steps that SMT solvers perform to convert formulas to CNF are not expressible in *e*DRAT. Complementary proof techniques are required for checking that the conversion steps the SMT solver used were sound. We discuss possible approaches to bridge this gap.

II. BACKGROUND

SMT is the problem of deciding the satisfiability of formulas in some (typically first-order) logical theory [13], [6]. The mainstream method employed by SMT solvers is conflict-driven clause learning modulo theories (CDCL(T)). This combines the CDCL algorithm from SAT [25] with theory-specific reasoning implemented by theory solvers. Given a formula ϕ in a theory T, the SMT solver first creates a Boolean abstraction ϕ_{abs} of this formula. The abstraction process replaces atoms in the background theory T with Boolean variables. The CDCL(T) algorithm then alternates between Boolean search and theory reasoning. The CDCL solver enumerates (possibly partial) models σ of ϕ_{abs} that are interpreted as conjunctions of literals in theory T. The theory solver checks whether this conjunction is satisfiable in T. If it is, Boolean search can continue and try to extend the assignment to a full model of ϕ_{abs}. If the conjunction of literals is not satisfiable, the theory solver produces a *theory lemma* that is added to the sets of clauses in the CDCL solver. This clause must be inconsistent with σ and will cause the CDCL solver to backtrack.

Modern SMT solvers extend this basic scheme in many ways—for example, with the dynamic creation of new variables and atoms on the fly and with mechanisms such as theory propagation—but the general principle remains. In one direction, the CDCL solver sends candidate Boolean assignments to the theory solver. In the other direction, the theory solver sends new clauses—that is, theory lemmas—to the CDCL solver. As in SAT, an SMT formula is unsatisfiable if the empty clause is derived by this process.

Generating proofs for CDCL(T) solvers is an active area of research, and several proof formats have been proposed. Proofs may include reasoning steps used during preprocessing and simplification of the original formula, conversion to clauses, Boolean resolution in CDCL, and theory-specific reasoning for justifying theory lemmas. Notable proof formats include Alethe [30] (supported by VeriT [7] and CVC5 [3]) and LFSC [33] (supported by CVC5 and ts predecessors CVC4 and CVC3). Both Alethe and LFSC have dedicated checkers [1], [32]. Other proof-producing SMT solvers [12], [10], [23] use solver-specific proof formats [11], [22], [27] and do not use independent checkers.

Most of these formats represent proofs as terms in a proof calculus. Such terms describe a traditional proof tree (or DAG) with the empty clause at the root. Each node in the tree represents a step that derives a conclusion (stored in the node) from the child nodes using a rule of proof calculus. Leaves represent axioms or assumptions (e.g., assertions from the original formula). One can distinguish generic logical frameworks such as LFSC that encode a particular proof calculus, and formats such as Alethe that come with a fixed calculus for a fixed set of theories. Logical frameworks are more flexible, since proof rules can be added to cover new theories and reasoning steps, but not all rules employed by SMT solvers can be compactly encoded in a logical framework. Most solvers use a fixed proof calculus and a dedicated proof format, which is similar to what Alethe provides. The recently introduced AletheLF format (ALF) is a logical framework that relies on an SMT-like syntax (similar to Alethe) in which SMT constructs can be more easily represented. ALF is supported by CVC5-1.1.0 and newer releases.

Coverage and proof granularity vary across solvers. Some solvers, such as CVC5, can generate low-level, high-detail proofs for almost all of the theories they support.[1] Other solvers, such as Z3, support proofs for a subset of theories and use more coarse-grained proofs. Detailed proofs with small inference steps are easier to verify, but generating such proofs can be costly and can introduce significant overhead both in runtime and memory usage.

We propose a coarse proof format that records the clauses produced (and deleted) during execution of the CDCL(T) algorithm. This extends the DRAT format used by Boolean SAT solvers, which is known to have low overhead.

III. DRAT EXTENSIONS FOR SMT

Our new proof format, *e*DRAT records the reasoning steps performed during the execution of the CDCL(T) algorithm. We do not attempt to express preprocessing, simplification, or conversion of a formula to CNF. Instead, *e*DRAT focuses on capturing the theory reasoning (the theory lemmas) and Boolean reasoning steps (the resolution clauses) that the SMT solver generates. We extend DRAT with syntax for defining theory terms and atoms, describing how these atoms map to Boolean variables, and distinguishing between the different types of clauses involved in SMT. We distinguish between three types of clauses: *assertions*, which are clauses from the CNF representation of the original formula; *theory lemmas* generated by the theory solver; and *regular clauses* generated by the CDCL solver.

An *e*DRAT proof consists of four components:

- The definitions of literals and terms used in the problem.
- The input problem converted to CNF.
- The theory lemmas that were emitted by the theory solver during the course of solving.

[1] As far as we know, CVC5 can generate proofs for all theories that do not involve floating points.

- The DRAT proof of the unsatisfiability of the formula as produced by the underlying SAT solver.

A. Syntax of *e*DRAT

SMT-LIB [5] is the standard input format for SMT solvers. The *e*DRAT syntax for term and atom definition is similar to SMT-LIB with a few extensions. Sort and term declarations in *e*DRAT use the following SMT-LIB syntax:

```
(declare-sort <name> <arity>)
(declare-fun <name> ( <sort>* ) <sort> )
```

The *e*DRAT syntax for datatypes and terms is also the same as in SMT-LIB. We introduce two new commands to give names to terms and to map DIMACS variables to literals:

```
(define-let <term-name> <smt-term>)
(define-literal <varid> <atom-name>)
```

The `define-let` command assigns a name to a term. It is a variant of the SMT-LIB `define-fun` command that omits the type of the term. The `define-literal` command states that a Boolean variable is mapped to a given atom. As in DIMACS, boolean variables are represented by positive integers. To simplify processing, the atom is specified by its name, which must appear either as a declaration or in a previous `define-let` command.

As in DRAT, a clause is represented by a list of non-zero literals terminated by 0. A positive integer denotes a positive literal, and a negative integer denotes its negation. The syntax for clausal reasoning is as follows:

```
a <list-of-integers> 0          Input problem clause
t <list-of-integers> 0       Theory reasoning clause
<list-of-integers> 0        Boolean reasoning clause
d <list-of-integers> 0               Clause deletion
```

*e*DRAT adds then two new prefixes to the DRAT syntax: one for assertions, and one for theory lemmas.

Example III.1. The following small example illustrates the *e*DRAT syntax.

```
1  (declare-sort T 0)
2  (declare-sort S 0)
3  (declare-fun f (T) S)
4  (declare-fun y () T)
5  (declare-fun x () T)
6  (define-let aux!312 (f y))
7  (define-let aux!338 (= (f x) (aux!312)))
8  (define-let aux!339 (= x y))
9  (define-literal 1 aux!338)
10 (define-literal 2 aux!339)
11 a 2 0
12 a -1 0
13 t 1 -2 0
14 0
```

This proof tells us that literals `1` and `2` are mapped to the atoms $f(x) = f(y)$ and $x = y$, respectively, where $f : T \to S$ is an uninterpreted function. Lines 11 and 12 state two assertions $x = y$ and $\neg f(x) = f(y)$, respectively, that come from the input problem. Line 13 is a theory lemma: $f(x) = f(y) \vee x \neq y$. The final step is the empty clause, which is derived by Boolean resolution of the three preceding clauses. This shows that the formula $x = y \wedge f(x) \neq f(y)$ is unsatisfiable.

B. Valido: A Toolchain for Checking eDRAT Proofs

*e*DRAT proof checking consists of two separate tasks: checking that the Boolean reasoning steps derive the empty clause, assuming that the theory lemmas are valid; and checking the validity of the theory lemmas. VALIDO is our toolchain for performing these two tasks.

To check the propositional part of the proof, VALIDO constructs a CNF formula ϕ with the input clauses (those with prefix `a`) and the theory lemmas (those with prefix `t`). It then extracts the DRAT proof π from the *e*DRAT file. This proof includes all the clauses corresponding to Boolean resolutions and all the clause deletions, keeping them in the same order as they occur in *e*DRAT. We then use a restricted version of the `DRAT-trim` tool to check that π is valid for ϕ. In this step, we treat all the theory lemmas as axioms and add them to the input clauses.

We restrict `DRAT-trim` to allow only clause additions that satisfy the reverse-unit-propagation (RUP) property and not the more general resolution-asymmetric-tautology (RAT) property, because accepting RAT clauses is not sound for SMT. It is possible for a formula ϕ to be satisfiable in the background theory T and for a clause C to be RAT with respect to ϕ, but for $\phi \wedge C$ to be unsatisfiable in T. This occurs because the addition of RAT clauses may eliminate some of the Boolean models for ϕ. RUP is sound for SMT as adding a RUP clause preserves all the satisfying models of the original formula.

Example III.2. Consider the formula $\phi := (\neg p \vee q) \wedge (p \vee r) \wedge (q \vee \neg r)$ where the propositional variables represent real arithmetic theory terms defined as follows: $p := x < 0$, $q := x \geq 1$, and $r := x \geq 2$. It follows from the definition that the unit clause p is RAT with respect to ϕ.

We can hence conclude that in propositional logic, ϕ is equisatisfiable to $\phi \wedge p$. However, in SMT, it can be readily checked that while the original formula ϕ is satisfiable with a model $x \to 2$, $\phi \wedge p$ is unsatisfiable.

When `DRAT-trim` successfully validates a DRAT proof, it also returns a set of clauses $U \subseteq \phi$ that forms the *unsat core* of the DRAT proof. To check the rest of the *e*DRAT proof, we only need to check the validity of the *core theory lemmas* that appear in the unsat core U.

One way of validating a theory lemma t is to employ existing SMT technology to generate a proof of the unsatisfiability of $\neg t$ in another proof format, and then to check it with a corresponding proof checker. This works, but we pursue a different approach to provide a higher degree of assurance: relying on purpose-built checkers that are provably sound.

In VALIDO, the theory lemmas are checked by two complementary tools that we call the *elaborator* and the *validator*. These tools are instantiated for each background theory T separately.

- An elaborator checks the unsatisfiability of a conjunctive formula in theory T and generates a *proof certificate*.
- A validator is a provably correct tool that takes a proof certificate and a theory lemma as input, and checks that the proof certificate validates the theory lemma.

Algorithm 1 gives an overview of this method. The architecture allows us to write an efficient but untrusted elaborator that generates a proof certificate for every theory lemma in the core. The validator is a simpler component that we develop and prove correct within the Lean 4 theorem prover [14].

Algorithm 1 General Method for Checking *e*DRAT Proofs

```
Input: An eDRAT Proof
Output: Result of eDRAT proof validation
(input,                                        ▷ Input problem in DIMACS format
t_lemmas,                                      ▷ Theory Lemmas in DIMACS format
drat_proof,                                    ▷ Boolean Reasoning as DRAT Proof
definitions) ← Decompose(eDRAT Proof)          ▷ Term and Literal Definitions
(e_res, unsat_core) ← DRAT-Trim(input, t_lemmas, drat_proof)
if e_res = Success then
    core_lemmas ← t_lemmas ∩ unsat_core
    proof_cert ← Elaborator(core_theory_lemmas, definitions)
    val_res ← Validator(proof_cert, core_lemmas, definitions)
    if val_res = Success then
        return Proof Validation Successful
    else
        return Theory Lemma Validation Failed
    end if
else
    return DRAT Proof Check Failed
end if
```

The key benefit of this approach is that it reduces the trusted code base. If the validator says that a proof certificate is valid, then we can trust that the corresponding lemma is also valid, independent of how the certificate was generated. In other words, we do not need to trust the elaborator, only the validator.

We have implemented elaborators and validators for two SMT-LIB theories: quantifier-free linear real arithmetic (QF_LRA) and quantifier-free uninterpreted functions with equality (QF_UF).

IV. Elaborator and Validator for QF_LRA

Let V be a set of variables. A theory lemma in QF_LRA is of the form $\psi := \bigvee_{i \in [n]} (F_i \bowtie_1 0)$, where each F_i is a linear expression over the variables V and $\bowtie_i \in \{<, \leq, =, >, \geq, \neq\}$. For example, the law of trichotomy $\psi_0 := x > 0 \vee x = 0 \vee x < 0$ is a theory lemma. Validating such a lemma is equivalent to proving that its negation—a conjunction of linear inequalities—is not satisfiable.

Example IV.1. The negation of ψ_0 is $\neg\psi := x \leq 0 \wedge x \neq 0 \wedge x \geq 0$, which can be rewritten as $\neg\psi := (-x \geq 0 \wedge x > 0 \wedge x \geq 0) \vee (-x \geq 0 \wedge -x > 0 \wedge x \geq 0)$, where the inequalities in each disjunct only have either $\geq$ or $>$ as the relational operator.

As shown in Example IV.1, our goal is to create a proof of unsatisfiability for a disjunction of conjunctions of linear inequalities that only involve $\geq$ or $>$ as the relational operators. For a single conjunctive QF_LRA formula, we use Farkas' Lemma to produce the unsatisfiability certificate.

Lemma 1. [16, Farkas' Lemma] A set S of linear inequalities of the form $F_i \; \{\geq, >\} \; 0$ is unsatisfiable if and only if there exists a non-negative linear combination of the inequalities in $S \cup \{1 > 0\}$ deriving either $-1 \geq 0$ or $0 > 0$.

Example IV.2. The formula $\neg\psi$ from Example IV.1 is unsatisfiable if both the disjuncts are unsatisfiable. The expression $1 \cdot (-x \geq 0) + 1 \cdot (x > 0) + 0 \cdot (x \geq 0) \equiv 0 > 0$ is a witness to the unsatisfiability of $-x \geq 0 \wedge x > 0 \wedge x \geq 0$, and $0 \cdot (-x \geq 0) + 1 \cdot (x > 0) + 1 \cdot (x \geq 0) \equiv 0 > 0$ witnesses the unsatisfiability of $-x \geq 0 \wedge -x > 0 \wedge x \geq 0$.

The set of non-negative multipliers for the linear inequalities that derive a trivially false inequality such as $-1 \geq 0$ or $0 > 0$ is called the *Farkas certificate* of unsatisfiability. We reduce the problem of finding the Farkas certificate to solving a linear program. For this purpose, we rely on the following variant of Farkas' Lemma:

Theorem 2. *A conjunction of linear inequalities of the form* $\psi := \bigwedge_{i=1}^{n} F_i \geq 0 \wedge \bigwedge_{j=1}^{m} G_j > 0$ *is unsatisfiable if and only if there exist non-negative constants* $\lambda_1, \ldots, \lambda_n$ *and* $\mu_0, \mu_1, \mu_2, \ldots, \mu_m$ *such that* $\mu_0 + \sum_{i=1}^{n} \lambda_i F_i + \sum_{j=0}^{m} \mu_j G_j \equiv 0$ *with* $\sum_{j=0}^{m} \mu_j = 1$ *(where* $\equiv$ *means that the expressions on both sides are identical).*

Proof. Farkas' Lemma guarantees that ψ is unsatisfiable if and only if one can derive either $-1 \geq 0$ or $0 > 0$ as non-negative linear combination of inequalities in $\psi \cup \{1 > 0\}$.

Let the non-negative linear combination be $D :\equiv \mu_0(1 > 0) + \sum_{i=1}^{n} \lambda_i (F_i \geq 0) + \sum_{j=0}^{m} \mu_j (G_j > 0)$. WLOG, we assume that $D \equiv 0 > 0$ because if $D \equiv -1 \geq 0$, then we set $\mu_0 \leftarrow \mu_0 + 1$ to derive $0 > 0$. Finally, we scale all λ_is and μ_js by a factor of $1/(\sum_{j=0}^{m} \mu_j)$ to ensure that $\sum_{j=0}^{m} \mu_j = 1$. □

The Valido elaborator for QF_LRA produces the Farkas certificate for each core theory lemma by searching for coefficients λ_i and μ_j that satisfy the conditions of Theorem 2. This amounts to solving a system of linear inequalities, which we do using the Simplex algorithm. The generated certificates are stored in a single file for all the core theory lemmas.

Example IV.3. Consider the following *e*DRAT proof fragment.

```
1  (declare-fun x () Real)
2  (define-let aux!0 (* x 1/2))
3  (define-let aux!1 (>= aux!0 0))
4  (define-let aux!2 (< x 0))
5  (define-let aux!3 (> x 0))
6  (define-literal 1 aux!1)
7  (define-literal 2 aux!2)
8  (define-literal 3 aux!3)
9  t 1 2 0
10 t 2 3 0
```

The theory lemma at line 9 is $\psi_6 := x/2 \geq 0 \vee -x > 0$ (which is valid), and that at line 10 is $\psi_7 = x > 0 \vee x < 0$

(which is not valid). On this example input, the elaborator will produce the following output:

```
LINE 9, (0, 1>0), (2, 1), (1, 2)
LINE 10, INVALID LEMMA
```

The first line is the Farkas certificate for ψ_6 (which is at line 9 in the original *e*DRAT proof). The certificate is a list of pairs (`farkas_coefficient,literal id`) with an optional term of the form (`farkas_coefficient,1>0`) for the Farkas coefficient of $1 > 0$. Thus, the certificate for ψ_6 is $0 \cdot (1 > 0) + 2 \cdot (x/2 \geq 0) + 1 \cdot (-x > 0) \equiv 0 > 0$.

The second line states that the lemma at line 10 of the *e*DRAT proof is invalid.

We have implemented a QF_LRA validator in Lean 4, in around 1300 lines of code. A few important data structures and functions are as follows:

1) A `LinearExpression` is a map `lexpr : Variable` $\rightarrow$ `Rat` that maps a variable to its rational coefficient in the expression. A `LinearConstraint` is a pair that consists of a `LinearExpression` and a relational operator, which is either $\geq$ or $>$.
2) A `Model` is a mapping from variables to rationals.
3) Function `evaluate (lexpr : LinearExpression) (m : Model)` computes the value of a `LinearExpression` in a `Model`
4) We define a proposition `isUnsat` as

```
def isUnsat (lemma : List LinearConstraint) (
    m: Model): Prop :=
  match lemma with
  | [] => False
  | (cnstr, lemma') => (evaluate cnstr m) → (
      isUnsat lemma' m)
```

   Given a negated lemma $S = \{C_1, \ldots, C_n\}$ as a set of linear constraints, `isUnsat` S is equivalent to

   ∀`(m: Model)`, `evaluate` C_1 `m` $\rightarrow \ldots \rightarrow$ `evaluate` C_n `m` $\rightarrow$ `False`

   This proposition says that for every `(m:Model)` at least one of `evaluate` C_i `m` must evaluate to *false*.
5) Given a negated lemma and its Farkas certificate of unsatisfiability, the following function checks whether the certificate is valid.

```
def check_farkas_certificate
    (farkas_coefficients: List Rat)
    (negated_lemma: List LinearConstraint) :
        Bool := ...
```

6) Finally, we proved the following theorem, which shows that function `check_farkas_certificate` is sound:

```
theorem check_farkas_cert_implies_isUnsat (
    check_farkas_certificate
    farkas_coefficients negated_lemma) = true
     → isUnsat negated_lemma := ...
```

The validator first parses the original *e*DRAT proof to collect the definition of each literal and theory lemma. It then parses the certificate file produced by the elaborator and checks every theory Farkas certificate with the function `check_farkas_certificate`. The check is successful if all theory lemmas in the certificate are valid.

V. Elaborator and Validator for QF_UF

QF_UF is one of the simplest theories defined in SMT-LIB. Formulas in QF_UF can include uninterpreted functions, predicates, and equality. A theory lemma in QF_UF is a disjunction of equalities and inequalities between uninterpreted terms. For example, $\psi := x \neq f(y) \vee y \neq g(z) \vee f(x) = f(f(g(z)))$ is a valid theory lemma in QF_UF.

A set of literals F in QF_UF must contain at least one inequality to be inconsistent. The traditional approach to show the inconsistency of F is based on congruence closure, as shown in Algorithm 2. This algorithm builds the smallest congruence relation Eq over the terms of F that includes all input equalities, and then checks whether a negated equality of F is inconsistent with Eq.

Algorithm 2 Congruence Closure Algorithm

```
1:  Input: $E$: a finite set of equalities, $D$: a finite set of inequalities
2:  Output: Unsat if $E \wedge D$ is not satisfiable, Sat otherwise
3:  $T \leftarrow$ All terms occuring in $E \cup D$ (including all the sub-terms)
    ▷ Initialization
4:  $Eq \leftarrow$ Each $t \in T$ in a singleton class
5:  for Each $t = u$ in $E$ with $Eq(t, u) =$ False do        ▷ Process
    input equalities
6:      $Eq \leftarrow$ Merge classes of $t$ and $u$ in $Eq$
7:  end for
8:  while $\exists\, C_1, C_2 \in Eq, f(t_1, \ldots, t_n) \in C_1, f(u_1, \ldots, u_n) \in C_2$
    such that $C_1 \neq C_2$ and $Eq(t_1, u_1) \wedge \ldots \wedge Eq(t_n, u_n)$ do
9:      $Eq \leftarrow$ Merge classes $C_1$ and $C_2$ in $Eq$
10: end while
11: for each inequality $t \neq u$ in $D$ do        ▷ Check for inconsistency
12:     if $Eq(t, u)$ holds then
13:         return Unsat
14:     end if
15: end for
16: return Sat
```

To check the results of Algorithm 2, it is sufficient to prove that we start with the right initial Eq and that every *Merge Class* operation is sound: that is, when we merge C_1 and C_2 at line 9 of Algorithm 2, the terms in those classes are congruent with respect to the current equivalence relation Eq.

The QF_UF elaborator in Valido generates unsatisfiability certificate based on this idea. Each certificate contains a description of the set of terms T, the initial equalities E, a series of equalities derived from E through congruence, and the inequality from D that led to unsatisfiability. The certificate format is kept simple to simplify parsing. An example is shown in Figure 1.

The certificate consists of the following three parts.

1) Definitions: The certificate starts with the definition of seven terms: four atomic terms including the two Boolean constants `true` and `false` and two uninterpreted constants c_4 and c_0, and three terms built by the application of function f. Each term is identified by its index in this

```
LINE: 16648, CERT
true
false
c_4
c_0
f 3
f 2
f 5
E(3, 5)
E(2, 6)
C(6, 4)
D(2, 4)
```

Fig. 1: Example QF_UF certificate

list. For example, the line `f 3` defines a term of index 4 obtained by applying the uninterpreted function f to the term of index 3. In other words, the term of index 4 is $f(c_0)$.

2) Equalities: After the term definitions, we list equalities from E. Each input equality is written as a line $E(i, j)$ where i and j are two term indices. For example, the line `E(3, 5)` is the equality $c_0 = f(c_4)$. An equality derived by congruence is written similarly but with the letter `C`. In the example `C(6,4)` represents the equality $f(f(c_4)) = f(c_0)$.

3) Inequality: Finally, the last line of the certificate is an inequality, indicated with the letter `D`, between the terms at indices 2 and 4, that is, $c_4 \neq f(c_0)$

The Boolean constants are predefined and included in all certificates (as the first two terms). This enables us to treat uninterpreted predicates as functions from some domain type to the Boolean. For example, a literal of the form $P(x)$ occurring in a theory lemma is treated as $P(x) = \mathit{true}$ in our certificates, and if $\neg P(x)$ occurs, it is converted to $P(x) = \mathit{false}$. This simple trick allows an unmodified congruence closure algorithm to handle uninterpreted predicates (provided we add the built-in inequality $\mathit{true} \neq \mathit{false}$).

The QF_UF validator parses the certificates produced by Valido and checks that they are valid. The central part in the validation process is a union-find data structure implemented in Lean 4 that is used to check that all equalities of the form $C(i, j)$ listed in a certificate are correct, that is, that the two indices i and j denote existing terms and that these two terms are congruent. The validator also checks a similar property for the inequality $D(i, j)$: the two indices i and j must represent existing terms, and the certificate is valid if i and j are in the same equivalence class in the union-find data structure. These checks are implemented in a function `check_certificate`, and the main correctness result follows:

```
def true_certificate (m: Model α β)
       (c: Certificate α β): Prop :=
  m.list_eq_holds c.wft c.base →
     m.diseq_holds c.wft c.conflict

theorem check_certificate_is_sound {α β: Type} [BEq β]
    [LawfulBEq β}] (c: Certificate α β})
      (h: Checker.check_certificate c = .ok ()):
        ∀ m, true_certificate m c := by
        ...
```

This states that the function `check_certificate` is sound. If this function succeeds (i.e., it returns `.ok ()`) then the certificate is true in any model `m`. In this theorem, a certificate is parameterized by two types α and β that represent the constants and function symbols in QF_UF terms. The certificate data structure includes a term table, a list of base equalities, a list of derived equalities, and a conflict of the form $D(i, j)$. A model is defined by three components: a domain τ (which is an arbitrary Lean type), a mapping from α to τ that defines the interpretation of constants, and a mapping from β to functions on τ that defines the interpretation of function symbols.

VI. Experiments

We have instrumented CVC5-1.1.1 to produce *e*DRAT proofs. The modifications consist of a new module that prints the *e*DRAT proof and changes to several existing CVC5 modules involved in the creation of input and theory clauses. Most changes were in the CDCL solver employed by CVC5, which is a heavily modified variant of MiniSat.

We have compared the *e*DRAT and Valido toolchain with two other proof formats currently supported by CVC5-1.1.1 on the QF_UF and QF_LRA benchmarks of the SMT-LIB repository [31]. All the experiments were run on a server with 384 GB RAM and 96 cores (48 Intel Xeon Platinum 8259CL CPUs), with a 2.50 GHz CPU frequency. The server runs Amazon Linux 2.

We ran CVC5 with a timeout of 300 seconds with four different proof-generation options: no proofs, proofs in the Alethe-LF (ALF) format, proofs in the LFSC format, and proofs in the *e*DRAT format. Some older versions of CVC5 also support the Alethe format, but this does not appear to be supported anymore in CVC5-1.1.1 and did not work on our benchmarks.

A summary of our experimental results is shown in Table I. The table includes the number of solved problems, the number of proofs successfully produced, and the average runtime on the satisfiable and unsatisfiable problems. A more detailed view of the experimental results is given in Tables II and III.

A. Proof Production Cost

As the table shows, generating proofs in the *e*DRAT format has low overhead. The difference in runtime between baseline CVC5 and CVC5-*e*DRAT on the QF_LRA problems is about 1%. On the QF_UF benchmarks, the average overhead of *e*DRAT proofs is about 16% on satisfiable instances and 27% on unsatisfiable instances. However, the QF_UF benchmark contains many easy problems that are solved in milliseconds (63% of the problems are solved by CVC5 in less than 0.1 s). If we remove these easy problems, the runtime difference between CVC5 and CVC5-*e*DRAT is less than 10%. In total, CVC5 and CVC5-*e*DRAT solve the same number of problems in all categories, apart from the class of satisfiable QF_LRA

TABLE I: Summary of Experiments

	QF_LRA						QF_UF					
	Solved Problems				Avg. Runtime (s)		Solved Problems				Avg. Runtime (s)	
Proof Mode	Unsat	Proofs	Sat	Unsolved	Sat	Unsat	Unsat	Proofs	Sat	Unsolved	Sat	Unsat
None	639		902	212	22.046	31.890	4353		3142	8	0.245	0.723
*e*DRAT	639	639	899	215	22.267	31.481	4353	4353	3142	8	0.286	0.924
ALF	621	591	876	256	32.185	56.267	4345	4335	3142	16	0.361	5.543
LFSC	623	503	876	254	32.412	85.825	4344	4283	3142	17	0.353	12.786

TABLE II: Experiment results on QF_LRA benchmark. All sizes are in MBs and times are in seconds. The averages are taken over the benchmark where the corresponding proof was successfully checked. The column ✓ represents the number of proofs that were successfully checked.

		cvc5 + Proof Generation Time				LFSC Proof			ALF Proof			*e*DRAT Proof		
Family	#	No Proof	LFSC	ALF	EDRAT	✓	Size	Time	✓	Size	Time	✓	Size	Time
Heizmann	29	87.725	196.195	159.793	85.504	15	61.251	55.099	19	40.978	105.052	29	14.491	10.066
LassoRanker	91	77.518	178.089	126.096	77.762	61	72.289	29.124	87	20.665	68.470	91	10.216	4.437
sc	35	44.889	194.576	81.503	42.856	20	225.166	104.933	32	23.178	129.505	35	5.735	1.339
uart	34	32.989	224.543	126.002	31.094	11	206.111	87.419	26	44.383	558.786	34	6.753	3.508
clock	36	20.454	81.828	28.276	19.925	29	70.890	20.705	36	6.158	25.426	36	0.938	1.027
latendresse	1	18.494	36.977	29.516	17.963	0	NA	NA	1	3.236	1.702	1	0.460	28.339
miplib	11	14.988	111.516	59.597	14.979	7	72.550	31.241	10	25.839	175.199	11	10.351	78.223
tta_startup	45	8.353	60.505	33.536	8.206	34	29.170	15.290	43	12.041	158.288	45	3.314	0.990
blending	9	3.886	300.212	114.993	4.301	0	NA	NA	9	129.184	119.071	9	15.932	49.526
TM	1	0.525	2.305	1.827	0.923	0	NA	NA	1	1.147	1.665	1	0.680	0.274
sal	96	0.094	1.731	0.655	0.120	96	2.222	1.354	96	0.680	0.941	96	0.162	0.178
spider	42	0.061	1.351	0.357	0.085	42	2.876	0.818	42	0.295	0.110	42	0.082	0.148
robotics	12	0.011	0.037	0.041	0.010	12	0.065	0.047	12	0.048	0.013	12	0.000	0.127
check	1	0.009	0.108	0.055	0.011	1	0.144	0.038	1	0.109	0.045	1	0.009	0.132
meti-tarski	150	0.007	0.013	0.011	0.007	150	0.008	0.011	150	0.007	0.011	150	0.001	0.128
keymaera	21	0.006	0.010	0.009	0.006	21	0.003	0.010	21	0.003	0.010	21	0.000	0.128

TABLE III: Experiment results on QF_UF benchmark. All sizes are in MBs and times are in seconds. The averages are taken over the benchmark where the corresponding proof was successfully checked. The column ✓ represents the number proofs that were successfully checked.

		cvc5 + Proof Generation Time				LFSC Proof			ALF Proof			*e*DRAT Proof		
Family	#	No Proof	LFSC	ALF	EDRAT	✓	Size	Time	✓	Size	Time	✓	Size	Time
Rodin	20	0.006	0.008	0.007	0.006	20	0.002	0.010	20	0.002	0.010	20	$\leq$ 0.001	0.079
Goel	229	0.209	9.406	8.549	0.232	217	0.413	1.660	226	0.311	0.220	229	0.092	0.091
CLEARSY	11	0.013	0.135	0.084	0.015	11	0.101	0.064	11	0.086	0.028	11	0.008	0.078
eq_diamond	100	0.022	0.342	0.340	0.043	100	0.051	0.067	100	0.044	0.031	100	0.055	0.086
NEQ	45	3.589	141.322	27.222	3.934	24	83.305	26.559	45	39.562	139.976	45	4.217	1.022
PEQ	38	7.394	171.559	61.164	8.418	20	136.427	46.852	34	51.902	129.771	38	20.327	3.919
SEQ	39	1.150	85.259	12.251	1.392	34	135.283	72.992	39	20.125	65.002	39	3.398	0.729
QG-class	3859	0.315	8.956	3.993	0.390	3854	10.486	23.253	3857	6.349	31.858	3859	0.495	0.224
TypeSafe	3	0.006	0.009	0.008	0.006	3	0.001	0.011	3	0.002	0.010	3	$\leq$ 0.001	0.078

problems. In this class, four problems are solved by cvc5 but not by cvc5-*e*DRAT and one problem is solved by cvc5-*e*DRAT but not by cvc5. This difference is most likely due to random variation caused by the operating system as all four take a runtime close to the timeout, rather than caused by the *e*DRAT proof generation.

The LFSC and ALF formats are more expensive, and the overhead depends on the theory. On QF_LRA, cvc5 fails to solve about 40 problems when using either format. On QF_UF producing either LFSC or ALF proofs doubles the number of timeouts. The runtime overhead is around 45% for both LFSC and ALF on satisfiable problems (on both QF_LRA and QF_UF). For the unsatisfiable problems, the overhead varies depending on proof-format and theory: on QF_LRA, LFSC incurs an overhead of 2.7x, and ALF is close to 2x slower than baseline cvc5. On QF_UF, the overhead is 7x for ALF and 17x for LFSC. The larger overhead on QF_UF is due to the fact that LFSC and ALF are not compatible with a symmetry-breaking procedure that baseline cvc5 employs [15]. Symmetry breaking is effective on the QF_UF benchmarks, but

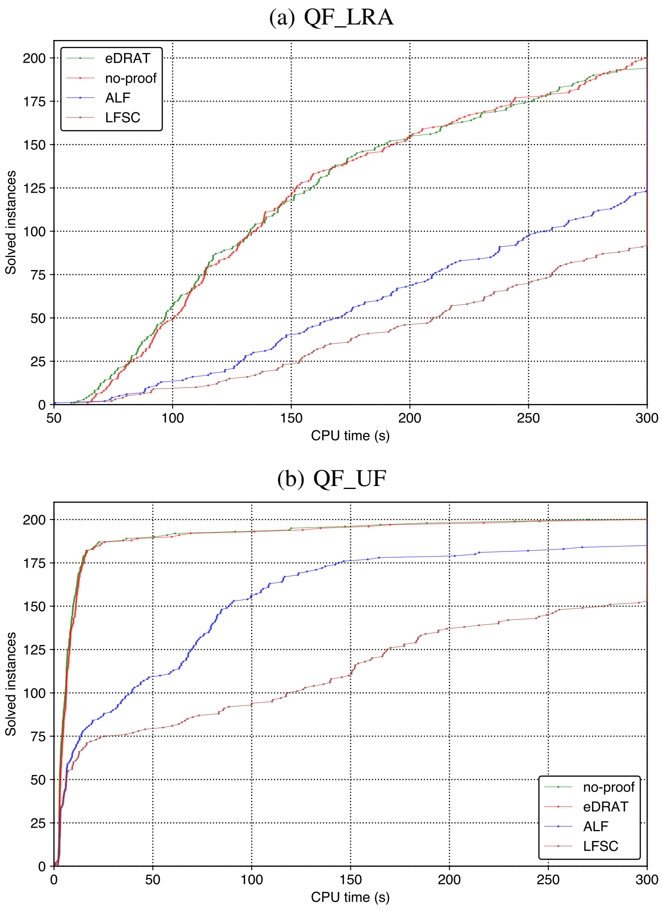

Fig. 2: Runtime on hardest problems

it must be disabled when cvc5 produces LFSC or ALF proofs. The *e*DRAT format is compatible with symmetry breaking and does not suffer from this disadvantage. We also see that both cvc5-LFSC and cvc5-ALF can solve a problem (i.e., print "unsat") but fail to generate a proof within the timeout. This happens because LFSC and ALF proofs are generated after cvc5 finds a problem to be unsat. After the problem is solved, cvc5 performs backward dependency analysis to construct a proof and export it to the LFSC or ALF format [4]. Both backward analysis and conversion to the external format can be expensive and cause a timeout.

Both the QF_UF and QF_LRA benchmarks contain a large number of easy problems that are solved in milliseconds. Figure 2 compares the runtime of our four cvc5 variants on the 200 problems that take the longest for baseline cvc5 to solve. The plots show that cvc5 and cvc5-DRAT have similar performance. cvc5-ALF and cvc5-LFSC are slower and timeout on several problems, but cvc5-ALF is more efficient than cvc5-LFSC.

B. Proof Size and Proof Checking Time

Figure 3 compares the proof sizes for different problem families in the QF_LRA and QF_UF benchmarks. The differences between the three formats vary with the theory and the problem family. Overall, *e*DRAT is more compact, except for a few problems. In QF_UF, ALF proofs are 2x larger than *e*DRAT proofs, and LFSC proofs are 11x larger than ALF proofs on average. In QF_LRA, ALF proofs are 4x larger than *e*DRAT proofs, and LFSC proofs are 11x larger than ALF proofs. Some of the size difference is due to the fact that ALF and LFSC include preprocessing steps, but this is significant mostly on easy problems. On hard problems, preprocessing represents a small part of the solver work, and proof steps related to resolution and theory lemmas dominate.

We validated the proofs with the appropriate checker. For LFSC, we used LFSCC[2]; for ALF, we use `alfc`[3]; and for *e*DRAT, we used Valido. All proofs were valid. Figure 4 shows the average proof checking time per benchmark family. For *e*DRAT, the graphs include the runtime of Valido (in Rust) and the certificate checkers (in Lean). On a few QF_LRA proofs, Valido is slower than the ALF checker (e.g., in the Latendresse family). This happens when theory lemmas are large (several hundreds of atoms per lemma) and our Simplex implementation is slow at computing Farkas certificates. Most proofs do not have such large lemmas. The ALF and LFSC checkers are also faster than Valido in some families of QF_UF problems, but the proofs in these families are small, and all checkers validate them in less than 0.1 s. On such small proofs, the cost of a call to `DRAT-trim` is a limiting factor for Valido. But overall, *e*DRAT proof checking is 3x and 15x faster than LFSC and ALF proof checking, respectively, in QF_LRA benchmarks, and 80x and 120x faster than LFSC and ALF, respectively, in QF_UF benchmarks. As one would expect, checking unsatisfiability certificates is cheaper than constructing unsat cores and certificates in the first place. The runtime of the certificate checker in Lean is smaller than the cost of the elaborator and `DRAT-trim` in all problem families. We also note that only a small fraction of all the theory lemmas included in the proof are part of the unsat core. Figure 5 shows the number of theory lemmas in the core compared with the total number of theory lemmas in the *e*DRAT file. Only lemmas in the core must be checked by Valido. On average, 1/8 of the QF_LRA theory lemmas and 1/2 of the QF_UF theory lemmas are in the core.

VII. Related and Future Work

Our results show that the DRAT proof format can be extended from SAT to SMT while preserving its efficiency. Compared with other proof formats currently supported by cvc5, *e*DRAT is the cheapest to generate. Although the *e*DRAT proofs are not detailed, it is still possible to efficiently check them by combining unsat core construction and specialized checkers for theory lemmas.

Otoni, et al. [27] present a proof system for OpenSMT that also combines DRAT with theory-specific checkers. A difference with our approach is that OpenSMT is modified to produce unsatisfiability certificates for each theory lemma,

[2] https://github.com/cvc5/LFSC

[3] https://github.com/cvc5/alfc

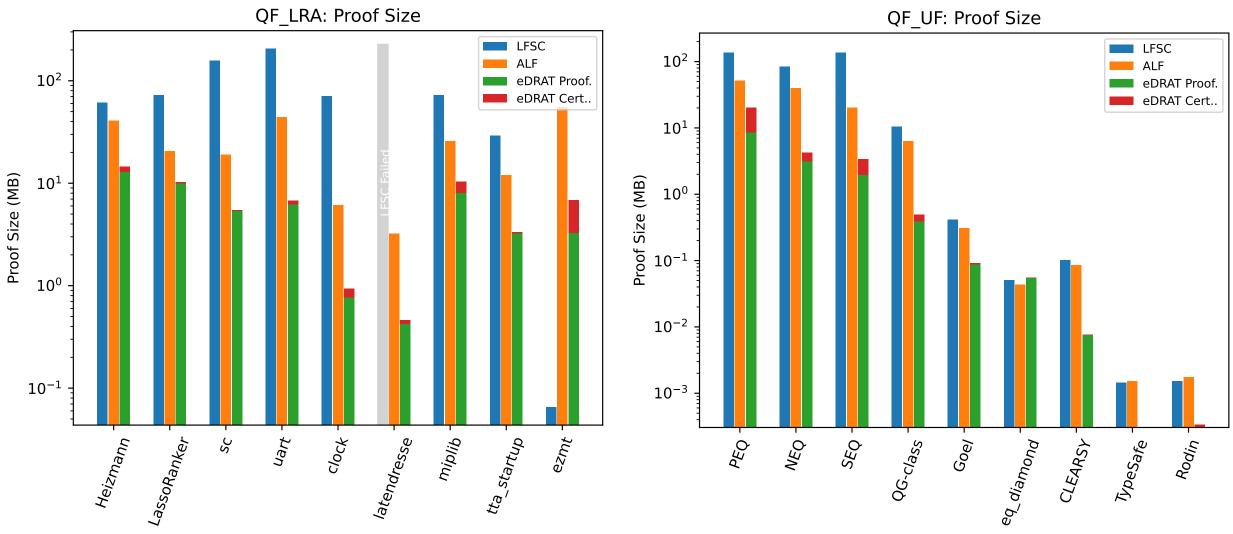

Fig. 3: Proof Sizes

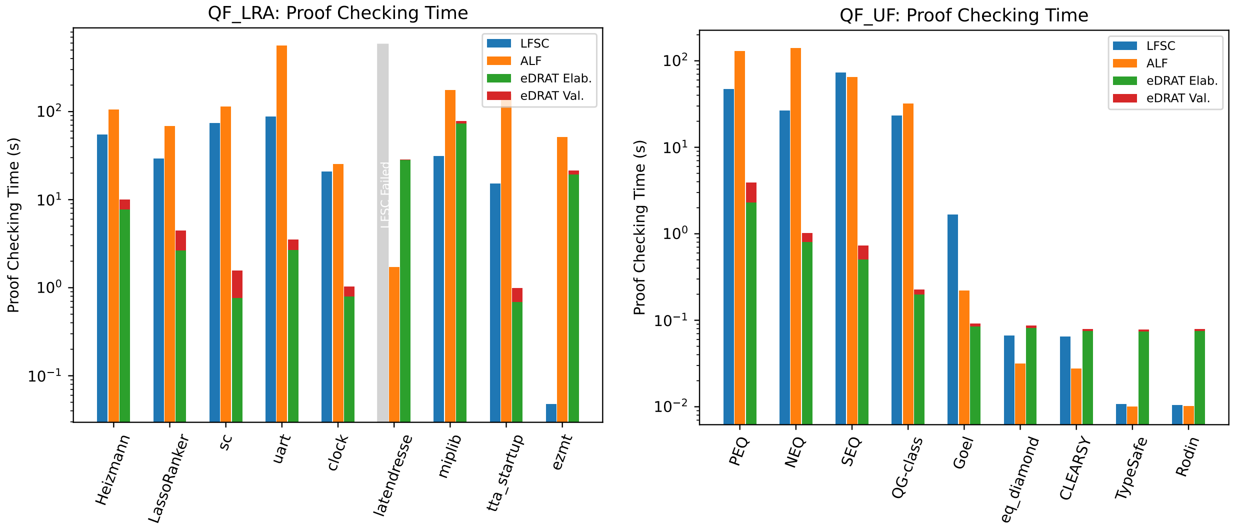

Fig. 4: Proof Checking Time

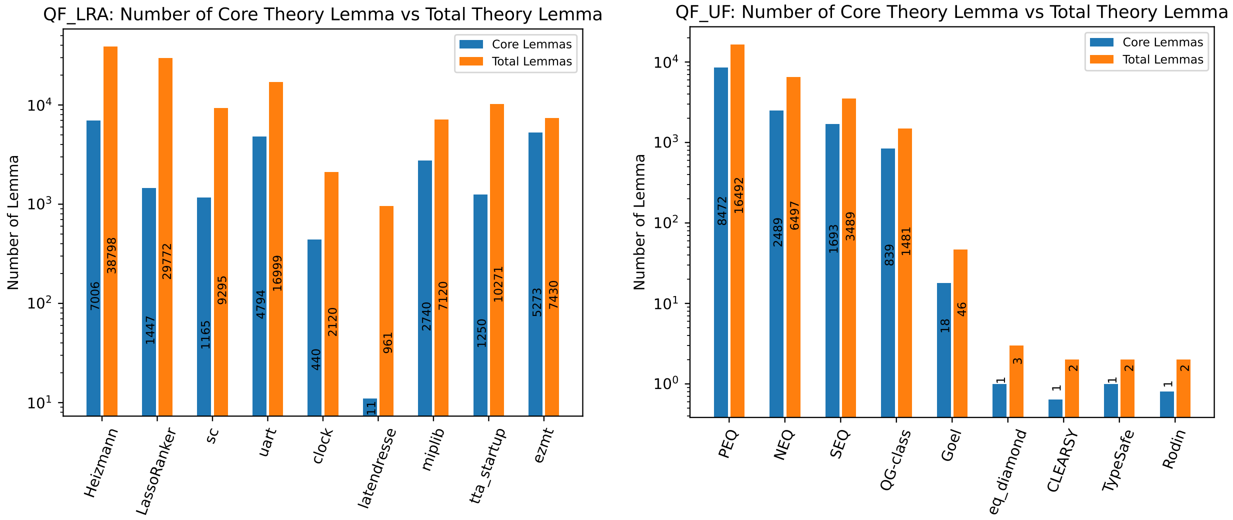

Fig. 5: Core Theory Lemmas

whereas we use an external elaborator to construct these certificates. Because we do not modify the CVC5 theory reasoning engines, we can efficiently produce *e*DRAT proofs for any theory supported by CVC5 (even though we cannot yet validate all of them). Another difference is that Otoni, et al. can the check conversion from SMT to CNF using a two-phase algorithm. The first phase checks a conversion from SMT to a DAG format (not defined in the paper), and the second phase checks Tseitin-style CNF conversion. This is more than what we can do with eDRAT, but it does not seem to be sufficient for the rewriting steps employed by CVC5. It is not clear from [27] how the simplifications that cvc5 heavily uses (such as the elimination of if-then-else, variable elimination, normalization of terms, and many other rewriting steps) could be handled. Finally, [27, Table II] shows that the overhead of their proof-production method is significant (e.g., 25% fewer solved instances in QF_LRA), while the main goal of eDRAT is to make proof generation as cheap as possible.

Another DRAT extension to SMT is presented by Feng, et al. [17]. This approach is specialized for satisfiability modulo monotonic theories. In this setting, predicates are monotonic relations over Boolean variables, and Feng, et al. use this property to build propositional DRAT proofs of theory lemmas. Like VALIDO, these extensions of DRAT for SMT offer proofs at low cost. The numbers reported in Otoni, et al. and Feng, et al. show that their proof generation techniques are efficient.

Currently, the main limitation of our approach is that it starts from a CNF formula. *e*DRAT is not adequate for representing proofs of preprocessing and conversion of formulas to clauses. We are considering three options to bridge this gap:

- Modify CVC5 to produce proofs of only its preprocessing steps in, say, the ALF format. This is probably the easiest approach but it has limitations. For example, as discussed in Sec. VI, some useful preprocessing steps must be disabled, and scalability remains to be evaluated.
- Use translation validation [28]. One can see preprocessing and conversion to CNF as a compilation process. Correctness amounts to showing that this compilation preserves satisfiability, and translation validation can be adapted to this problem. An issue is that this may require the solver to produce hints to enable this approach.
- Implement a provably correct preprocessor, say, in Lean. This may require the most effort, but it could provide the most benefit. One issue with this option is the cost of maintaining and updating the preprocessor as new theories and possibly new simplification techniques are discovered.

VIII. Conclusion

*e*DRAT extends the well-known DRAT format of SAT to SMT. Our experiments show that *e*DRAT proofs can be produced efficiently and can be efficiently validated, which makes routine use of proof-producing SMT solvers more practical. In future work, we will extend the VALIDO tool chain to cover more theories, and we will extend the approach to include proofs of preprocessing.

Acknowledgment

S Hitarth was partially supported by the Madrid Regional Government (César Nombela grant 2023-T1/COM-29001), MCIN/AEI/10.13039/501100011033/FEDER, and EU (grant PID2022-138072OB-I00).

References

[1] B. Andreotti, H. Lachnitt, and H. Barbosa. Carcara: An Efficient Proof Checker and Elaborator for SMT Proofs in the Alethe Format. In S. Sankaranarayanan and N. Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 13993, pages 367–386. Springer Nature Switzerland, 2023.

[2] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. Luckow, N. Rungta, O. Tkachuk, and C. Varming. Semantic-based automated reasoning for AWS access policies using SMT. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9, 2018.

[3] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. CVC5: A versatile and industrial-strength SMT solver. In D. Fisman and G. Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[4] H. Barbosa, A. Reynolds, G. Kremer, H. Lachnitt, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Viswanathan, S. Viteri, Y. Zohar, C. Tinelli, and C. Barrett. Flexible proof production in an industrial-strength SMT solver. In J. Blanchette, L. Kovács, and D. Pattinson, editors, *Automated Reasoning*, pages 15–35, Cham, 2022. Springer International Publishing.

[5] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at https://smt-lib.org.

[6] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Chapter 33. Satifiability Modulo Theories. In A. Biere, M. Heule, H. Van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*. IOS Press, 2021.

[7] T. Bouton, D. Caminha B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: An open, trustable and efficient SMT solver. In R. A. Schmidt, editor, *Automated Deduction – CADE-22*, pages 151–156, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[8] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv symbolic model checker. In A. Biere and R. Bloem, editors, *Computer Aided Verification*, pages 334–342, Cham, 2014. Springer International Publishing.

[9] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. The Kind 2 model checker. In *Computer Aided Verification*, pages 510–517, Cham, 2016. Springer International Publishing.

[10] J. Christ, J. Hoenicke, and A. Nutz. SMTInterpol: An interpolating SMT solver. In A. Donaldson and D. Parker, editors, *Model Checking Software*, pages 248–254, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[11] L. de Moura and N. Bjørner. Proofs and refutations, and z3. In *The LPAR 2008 Workshops: KEAPPA and IWIL 2008*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, November 2008.

[12] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] L. De Moura and N. Bjørner. Satisfiability Modulo Theories: An Appetizer. In M. V. M. Oliveira and J. Woodcock, editors, *Formal Methods: Foundations and Applications*, volume 5902, pages 23–36. Springer Berlin Heidelberg, 2009.

[14] L. de Moura and S. Ullrich. The Lean 4 theorem prover and programming language. In A. Platzer and G. Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

[15] D. Déharbe, P. Fontaine, S. Merz, and B. Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, pages 222–236, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[16] J. Farkas. Theory of simple inequalities. *Journal for pure and applied mathematics (Crelles Journal)*, 1902(124):1–27, 1902.

[17] N. Feng, A. J. Hu, S. Bayless, S. M. Iqbal, P. Trentin, M. Whalen, L. Pike, and J. Backes. Drat proofs of unsatisfiability for sat modulo monotonic theories. In B. Finkbeiner and L. Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 3–23, Cham, 2024. Springer Nature Switzerland.

[18] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.

[19] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[20] A. Goel and K. Sakallah. AVR: Abstractly verifying reachability. In A. Biere and D. Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 413–422, Cham, 2020. Springer International Publishing.

[21] M. J. H. Heule. The DRAT format and DRAT-trim checker.

[22] J. Hoenicke and T. Schindler. A simple proof format for SMT. In D. Déharbe and A. E. J. Hyvärinen, editors, *Satisfiability Modulo Theories, 2022*, volume 3185 of *CEUR Workshop Proceedings*, pages 54–70. CEUR-WS.org, August 2022.

[23] A. E. J. Hyvärinen, M. Marescotti, L. Alt, and N. Sharygina. OpenSMT2: An SMT solver for multi-core and cloud computing. In N. Creignou and D. Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 547–553, Cham, 2016. Springer International Publishing.

[24] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[25] J. Marsuqes-Sliva. Chapter 4. Conflict-Driven Clause Learning. In A. Biere, M. Heule, H. Van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*. IOS Press, 2021.

[26] K. L. McMillan and O. Padon. Ivy: A multi-modal verification tool for distributed algorithms. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification*, pages 190–202, Cham, 2020. Springer International Publishing.

[27] R. Otoni, M. Blicha, P. Eugster, A. E. J. Hyvärinen, and N. Sharygina. Theory-specific proof steps witnessing correctness of SMT executions. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 541–546, 2021.

[28] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[29] N. Rungta. A billion SMT queries a day (invited paper). In S. Shoham and Y. Vizel, editors, *Computer Aided Verification*, pages 3–18, Cham, 2022. Springer International Publishing.

[30] H. Schurr, M. Fleury, H. Barbosa, and P. Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). In C. Keller and M. Fleury, editors, *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021*, volume 336 of *EPTCS*, pages 49–54, 2021.

[31] SMT-LIB. The Satisfiability Modulo Theories Library. https://smtlib.cs.uiowa.edu/. Accessed on March 15, 2024.

[32] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.

[33] A. Stump, A. Reynolds, C. Tinelli, A. Laugesen, H. E. III, C. Oliver, and R. Zhang. LFSC for SMT proofs: Work in progress. In D. Pichardie and T. Weber, editors, *Proceedings of the Second International Workshop on Proof Exchange for Theorem Proving, PxTP 2012, Manchester, UK, June 30, 2012*, volume 878 of *CEUR Workshop Proceedings*, pages 21–27. CEUR-WS.org, 2012.

[34] N. Wetzler, M. J. H. Heule, and W. A. Hunt. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014*, volume 8561, pages 422–429. Springer International Publishing, 2014.