A General Approach for SMT Proof Skeletons

Joseph E. Reeves[1]✉, Haniel Barbosa[2], Andrew Reynolds[3,4], and Marijn J. H. Heule[1,4]

[1] Carnegie Mellon University, Pittsburgh, Pennsylvania, United States
{jereeves,mheule}@cs.cmu.edu
[2] Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
hbarbosa@dcc.ufmg.br
[3] The University of Iowa, Iowa City, USA
andrew-reynolds@uiowa.edu
[4] Amazon Web Services, Seattle, WA, USA

Abstract. SMT solvers increasingly produce proof certificates to meet the trust requirements of safety-critical applications. However, eagerly justifying learned theory lemmas during solving constitutes a major performance bottleneck. Recent work mitigates this cost by emitting proof skeletons that record only SAT reasoning and unannotated theory lemmas, though existing approaches depend on new proof formats and specialized theory-specific tooling. We present a general approach that restricts SMT proof skeletons to core SMT reasoning: preprocessing, clausification, and unannotated theory lemmas. We develop external tools for SAT reasoning and proof trimming that reduce the number of theory lemmas requiring justification. An experimental evaluation using the SMT solver CVC5 on SMT-LIB benchmarks across the UF, LIA and LRA theories, with and without quantifiers, demonstrates faster solving and competitive checking performance compared to eager proof production, particularly on quantifier-free problems.

Keywords: Satisfiability Modulo Theories · Proof Logging · Proof Skeletons.

1 Introduction

Automated reasoning tools, particularly satisfiability modulo theories (SMT) solvers [6], are widely used in applications requiring high levels of trust including cloud security [2] and hardware and software verification [15, 23–25, 28]. Because SMT solvers are large and complex pieces of software, some tools now produce proof certificates [3] that justify solver reasoning thereby increasing confidence for *unsatisfiable* results.

To derive unsatisfiability, an SMT solver performs: i) *preprocessing*—rewriting and simplification; ii) *clausification*—transforming the preprocessed input into clauses forming the Boolean abstraction; iii) *propositional satisfiability (SAT) reasoning* over the clauses; and iv) *theory reasoning*—using theory solvers to learn *theory lemmas* constructed as clauses that refine the Boolean abstraction; eventually deriving a contradiction when no solution exists. An SMT proof certificate contains the information necessary to verify reasoning underlying the SAT refutation, including both the SAT reasoning steps used to derive a contradiction and the preprocessing, clausification, and

theory reasoning proof steps used to justify the relevant clauses and theory lemmas that appear in the SAT reasoning. Each proof step is accompanied by a *justification* with sufficient details for efficient validity checking. While justifications for SAT reasoning and clausification are straightforward to log, those for preprocessing and theory lemmas can be substantially more complex, as they must capture the intricate reasoning of rewriting engines and theory solvers.

The standard *eager* approach has the preprocessing module and theory solvers produce justifications online during solving [3]. For example, each time a theory solver implementing congruence closure for the theory of uninterpreted functions (QF_UF) is called, the theory solver will record the series of inferences used to deduce a theory lemma [29]. Once the SMT solver derives a conflict, the solver traces its SAT reasoning and emits all relevant theory lemmas and their justifications to the proof certificate. Any justifications for theory lemmas that are ultimately unused represent wasted computation; this overhead can be substantial for certain theories [5, Sect. 6], and is exacerbated in high-throughput applications processing up to a billion queries a day [35].

An alternative *lazy* approach has the SMT solver produce a *proof skeleton*— a subset of the full reasoning steps that omits e.g. theory lemma justifications— thereby reducing proof-logging overhead during solving. Existing frameworks support skeletons containing clausification steps, unjustified theory lemmas, and SAT reasoning [14, 18, 31]. To check such a skeleton, the SAT refutation is extracted, optionally *trimmed* to remove redundant reasoning steps, and the theory lemmas retained in the trimmed refutation are then justified by a dedicated *elaborator* and verified by a *proof checker*.

We present a more general approach for SMT proof skeletons (Section 3) that logs only core SMT reasoning: preprocessing (pre_i), clausification (C_i) and theory lemmas (T_i). Such a skeleton is accessible to SMT solvers without prior proof support for theories or SAT reasoning. Our skeleton extends the Alethe proof format [37], which supports preprocessing and quantifier reasoning beyond the scope of existing skeleton formats. The skeleton contains justifications J_i^{type} for the preprocessing and clausification steps but omits theory lemma justifications:

$$(pre_1, J_1^{\text{pre}}), \ldots, (pre_m, J_m^{\text{pre}}), (C_1, J_1^{\text{C}}), \ldots, (C_r, J_r^{\text{C}}), (T_1, \_), \ldots, (T_k, \_)$$

Rather than recording the SMT solver's SAT refutation, our skeleton contains only the information needed to extract the unsatisfiable propositional formula $F = C_1 \wedge \cdots \wedge C_r \wedge T_1 \wedge \cdots \wedge T_k$. We developed separate tools for SAT reasoning and trimming that produce an unsatisfiable core $U = C_1 \wedge \cdots \wedge C_r \wedge T_1' \wedge \cdots \wedge T_j'$ with $U \subseteq F$. By performing SAT reasoning externally, we can apply trimming strategies specifically targeting theory lemma removal, reducing the number of theory lemmas in the core. Finally, we produce justifications for the remaining theory lemmas in the core: $(T_1', J_1^T), (T_2', J_2^T), \ldots, (T_j', J_j^t)$.

While other skeleton checking tools used hand-crafted elaborators and checkers for each supported theory to improve performance, we use a general elaborator, CVC5 [5], that can produce Alethe justifications [22] for lemmas from the quantified and quantifier-free theories of equality and uninterpreted functions (UF), linear integer and real arithmetic (LIA and LRA), and theory combinations therein (Section 4.2). We mitigate the overhead of using a general-purpose elaborator by i) organizing justification in *batches*,

meaning several theory lemmas can be justified and checked at once, and ii) parallelizing theory lemma justifications.

In summary, our contributions are:

- a generic framework for production and checking of SMT proof skeletons that relies on external SAT reasoning tools, based on the Alethe proof format, implemented in the SMT solver CVC5 and proof checker CARCARA;
- iterative proof-trimming techniques for minimizing theory lemma counts in cores, together with batched and parallelized lemma justification;
- a large-scale evaluation on SMT-LIB benchmarks that demonstrates the effectiveness of our lazy approach on the quantifier-free logics QF_UF, QF_LIA, QF_LRA, and QF_UFLIA and its limitations on the quantified logics UF and UFLIA.[5]

2 Background

2.1 Proof Production in SAT

We represent formulas for the Boolean satisfiability problem (SAT) in propositional logic using *conjunctive normal form* (CNF), i.e., conjunctions of *clauses* where each clause is a disjunction of *literals*. A literal ℓ is either a Boolean variable x or a negated variable $\overline{x}$. A *unit* clause contains a single literal. *Unit propagation* applies the following operation to formula F until a fixed point is reached: for all units α, remove clauses from F containing a literal in α and remove from the remaining clauses all literals negated in α. If unit propagation yields the empty clause ($\bot$) we say it derived a *conflict*. An *unsatisfiable core* (core) of a formula is a subset of the formula that is unsatisfiable, and is *minimal* if removing any clause from the core will make it satisfiable.

The dominant SAT algorithm, conflict-driven clause-learning (CDCL), produces a *clausal proof* for unsatisfiable problems. A clausal proof is a sequence of clause additions and deletions, where each subsequent clause addition step must meet the criteria of a chosen proof system, e.g., $F, C_1, C_2, \ldots, C_m$ is a clausal proof of C_m. The case of $C_m = \bot$ serves as a refutation for F. We will refer to clauses in the proof as *clausal lemmas* to avoid confusion with theory lemmas. CDCL solvers and proof checkers commonly use the resolution asymmetric tautology with deletions (DRAT) proof system [20], but when used within SMT solvers they are restricted to a subset of DRAT, the reverse unit propagation with deletions (DRUP) proof system, to prevent unsound reasoning. A clause C is RUP w.r.t. formula F if unit propagation on F and the units $\overline{\ell}$ for $\ell \in C$ derives a conflict.

A DRUP proof is transformed into a linear RUP (LRUP) proof by adding *hints* to proof steps, i.e., identifiers that denote clauses propagated in each RUP check, used to speed up checking with a formally verified tool [39]. The proof trimming tool DRAT-TRIM simultaneously trims unnecessary clausal lemmas from the proof and transforms RUP steps into LRUP steps through a process called *backward checking* (Section 4.1).

[5] Code and data available at https://github.com/jreeves3/SMT-Skeleton-Check

2.2 Proof Production in SMT

SMT solvers generally follow the CDCL($\mathcal{T}$) architecture [30], combining a *CDCL SAT solver* and *a set of specialized theory solvers*. The SAT solver searches for satisfying assignments for the Boolean abstraction of the input, in which the theory atoms are abstracted as propositional variables. The theory solvers determine whether these literals are also satisfiable modulo the combination of theories $\mathcal{T}$. If they are not, a $\mathcal{T}$-valid disjunction of literals, a *lemma*, is learned to capture the inconsistency, which will force the SAT solver to search for a new assignment, if any exists. The other key solving components include a *preprocessing module*, which simplifies the input formulas as much as possible; and a *clausifier*, which converts the preprocessed formulas into a conjunction of *clauses* over theory literals, whose abstraction is the initial clause set for the SAT solver. Given the differences in these components, and the varying ways solvers simplify formulas and reason about theories, SMT solvers can differ significantly in how they produce proofs [8, 19, 21, 27], and a standard proof system or format is yet to emerge in SMT. Here we focus on the proof production architecture of CVC5, the Alethe proof format [37] and its checker CARCARA [1], which we extend for this work, and the proof system relevant to the logical fragment we consider: formulas involving any of quantifiers, equality, uninterpreted functions, and linear integer or real arithmetic.[6]

Given an SMT input φ, CVC5 can produce a refutation in the Alethe proof system with the form $\pi : \varphi \to \bot$, i.e., deriving $\bot$ from the assumption φ. This proof is produced modularly, with each solving component producing individual proofs that are merged together into this refutation. The preprocessing module produces proofs $\pi_i^{\mathrm{pp}} : \varphi \to \phi_i$, for each ϕ_i generated during the simplification of φ. Different simplifications may require different sets of rules, such as constant folding in different theories or if-then-else elimination. The clausifier produces proofs $\pi^{\mathrm{cnf}} : \phi \to C_j$ for each clause C_j generated from the preprocessed formulas. The rules in each π^{cnf} are a combination of Boolean transformations and introductions of Boolean formulas representing the definition of Tseytin variables, used to ensure that the CNF conversion is polynomial. The combination of theory solvers produce proofs $\pi^{\mathrm{t}} : L$ for each theory lemma L derived. The rules in each π^{t} depend on the respective theory. For example, the UF theory reasoning is modeled with rules to describe congruence reasoning [17], whereas the LRA theory reasoning is modeled with a rule capturing Farkas' lemma [13, 36], which guarantees that there exists a linear combination of these inequalities equivalent to $\bot$, that can be checked polynomially. For LIA, rules for branching and integer bound tightening are used in addition to the Farkas' lemma. The SAT solver produces a resolution proof $\pi^{\mathrm{res}} : C_1 \wedge \cdots \wedge C_m \wedge T_1 \wedge \cdots \wedge T_l \to \bot$ for the refutation of the clause set corresponding to the subset of the clausified preprocessed input and the theory lemmas needed to derive $\bot$. Note that π^{res} is in terms of the first-order clauses, as are the derivation rules that conclude $\bot$ from them, since the inferences on the propositional abstraction are

[6] Internally CVC5 produces proofs in the Cooperating Proof Calculus (CPC) (https://cvc5.github.io/docs-ci/docs-main/proofs/output_cpc.html), which are then translated into Alethe. The Alethe format, together with a complete list of the proof rules, is described at https://verit.gitlabpages.uliege.be/alethe/specification.pdf. Extensions of Alethe to other theories, such as the theories of fixed-width bit-vectors and of strings, are under active development.

lifted to the original literals. By connecting the assumptions of each of these proofs, the final refutation of the input φ is generated. For more details see e.g. [5, 19].

Alethe proofs are a series of steps represented as an indexed list of `step` commands referencing their premises and justified by a given inference rule, concluding a clause, represented as a list of theory literals. The command `assume` is analogous but used only for introducing assumptions. The indexed steps induce a directed acyclic graph rooted on the step concluding $\bot$ and with $\varphi_1, \ldots, \varphi_n$ as the only assumption leaves (the input φ is generally given as multiple assertions $\varphi_1, \ldots, \varphi_n$). The format supports subproofs, which simulate the effect of the $\Rightarrow$-introduction rule of Natural Deduction, where local assumptions are put in context and the last step in a subproof represents its conclusion and the closing of its context. The overall proof may be seen as preprocessing and a ground first-order resolution refutation with theory lemmas justified by closed subproofs. Alethe proofs can be reconstructed in several proof assistants [11, 12, 22, 38] and it has a dedicated high-performance checker and elaborator, CARCARA.

Example 1. This QF_UF query requiring simple congruence reasoning is used throughout the paper, labelled CONGEX. The query can be found in the Github repository.

```
(assert (= a b))
(assert (= b d))
(assert (and p1 true))
(assert (or (not p1) (and p2 p3)))
(assert (or (not p3) (not (= (f a b) (f b d)))))
(assert (or (not p3) (not (= a d))))
```

3 The Alethe SMT Proof Skeleton Framework

Our SMT proof skeleton framework, shown in Figure 1, is divided into solving, checking, and a *theory oracle* for the new proof rule `prop_unsat`. The oracle computes and trims a SAT refutation, then justifies and checks the core theory lemmas.

In CVC5, the existing eager approach [5] computes and stores all preprocessing and theory lemma justifications during solving, then after solving trims the SAT refutation and prints the relevant proof steps.[7] We instrumented CVC5 to produce an Alethe proof skeleton by logging inferences as they were derived. The skeleton includes fully justified preprocessing and clausification steps transforming the preprocessed input query into clauses $C_1, \ldots, C_r$, plus two new step types: unjustified theory lemma additions (`hole`) producing the set $T_1, \ldots, T_k$, and a claim of propositional unsatisfiability (`prop_unsat`). An example `hole` step from CONGEX derived via transitivity is:

```
step t7 (cl (or (= a d) (not (= a b)) (not (= b d)))) :rule hole :args ("LEMMA")
```

Unlike the default eager approach, proof logging within theory solvers is disabled when producing theory lemma holes, speeding up solving. Furthermore, the skeleton

[7] Hitarth et al. [18] instrumented CVC5 to log SAT clauses and theory lemmas as they are derived, but their tool is not open source. Moreover it did not support preprocessing nor other formats than their *e*DRAT proof format, whose validation toolchain VALIDO is also not open source.

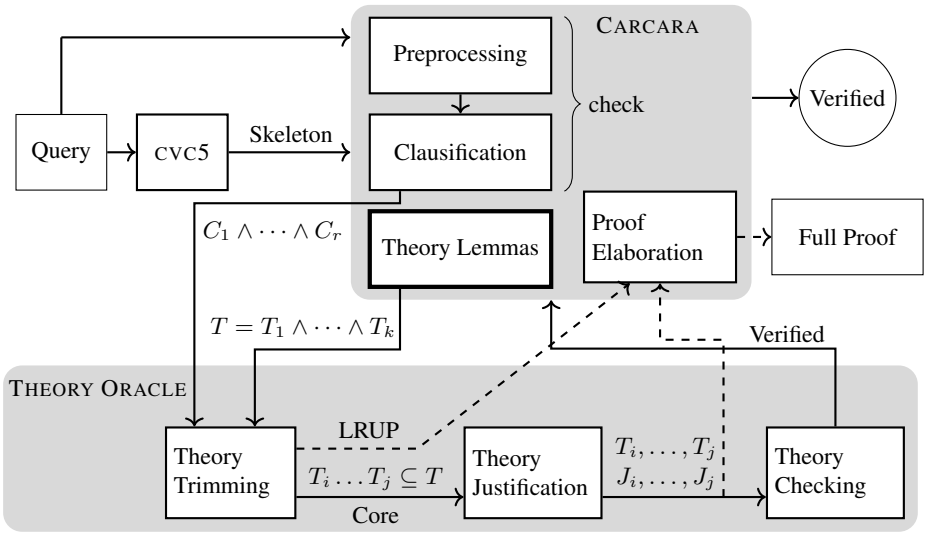

Fig. 1: SMT solving and proof checking. CARCARA checks preprocessing and clausification steps normally. Then, the theory oracle produces, trims, and checks SAT reasoning and justifies and checks core theory lemmas. LRUP proof production and theory justifications can optionally be used for proof elaboration (dashed lines).

omits SAT reasoning and instead concludes with the `prop_unsat` step containing an unsatisfiable set of premises $C_1, \ldots, C_r, T_1, \ldots, T_k$. A `prop_unsat` step for CONGEX would, e.g., be

```
(step t9 (cl) :rule prop_unsat :premises (C1 C2 C3 C4 C5 C6 C7 C8 C9 T1 T2))
```

which uses 9 justified premises and 2 unjustified theory lemmas (`T1` and `T2`). Their conjunction (each interpreted in their clausified format) forms an unsatisfiable propositional formula.

The checker, CARCARA, accepts the resulting skeleton and verifies the preprocessing and clausification steps normally, skipping the theory lemma hole steps. Upon reaching the `prop_unsat` step, it calls the external theory oracle with a propositional representation of the premises (F) and an SMT-LIB representation of the theory lemmas. The oracle verifies F's unsatisfiability by computing a refutation, then justifies and checks the relevant theory lemmas. An optional elaborator replaces the `prop_unsat` step by a fine-grained proof using theory lemma justifications and the LRUP proof from the oracle, enabling independent checking outside our framework. The elaborator's detailed description and evaluation can be found in the tool's Github repository.

The oracle operates in two phases: (1) *theory trimming*, using a SAT solver and proof trimmer to verify SAT reasoning and reduce the number of theory lemmas in the core; and (2) *theory lemma justification and checking*, using CVC5 to compute justifications for core lemmas and default CARCARA to verify them. Theory lemmas are processed in batches (multiple at a time) and the process can be parallelized. If the the-

ory oracle verifies the SAT reasoning and verifies the justifications of core lemmas it returns a verified result for the `prop_unsat` step and otherwise returns a not verified result. Note, the oracle could in principle use any tool to justify theory lemmas and check those justifications. We chose to use CVC5 and CARCARA because they already support many theories. Furthermore, all justifications produced by CVC5 within the external oracle are verified by CARCARA without nested calls to the external oracle, so the reliance on CVC5 as skeleton producer and theory lemma elaborator introduces no unsoundness.

4 Theory Oracle for the `prop_unsat` Proof Step

4.1 Theory Trimming with SAT Reasoning

The theory trimming module receives an unsatisfiable propositional formula F containing the clausified preprocessed input $C_1, \ldots, C_r$ and the clausified unjustified theory lemmas $T_1, \ldots, T_k$, together with a mapping from theory lemma clauses to their SMT-LIB representation. It first computes a clausal proof $(P_1, \ldots, P_n)$ of F using the SAT solver CADICAL [7], then applies our *theory-last* proof trimmer to both verify the SAT reasoning and produce a core $U \subseteq F$. Only the theory lemmas in U are subsequently justified and checked.

Theory-Last Proof Trimming Theory-last trimming modifies the *backward trimming* algorithm of DRAT-TRIM [40]. Standard backward trimming processes the clausal proof in reverse, performing RUP checks on active steps and using conflict analysis to mark (activate) the clauses (reasons) that contributed to each conflict (see Example 2); all marked input clauses then constitute the core. Specifically, a RUP check on step P_i will add all clauses in F and clauses $P_1, \ldots, P_{i-1}$ to the propagation engine, then propagate the units $\overline{P}_i$ to derive a conflict. Conflict analysis marks reasons for the conflict, where a *reason* is the clause that propagated a specific unit. Given a conflicting clause C, conflict analysis starts from the list of negated unit literals $\{\ell_1, \ldots, \ell_k\} = C$, then for each ℓ_i in the list the reason clause $R(\ell_i)$ will be marked and the literals $R(\ell_i) \setminus \ell_i$ will be added to the unit list. The procedure continues until only unit literals without reasons (unit clauses from the original formula or unit clauses from $\overline{P}_i$) remain.

Example 2. Given a formula $F = C_1 : (\overline{x}_1 \vee x_4 \vee x_6) \wedge C_2 : (\overline{x}_2 \vee \overline{x}_3) \wedge C_3 : (x_1 \vee x_3 \vee x_4) \wedge C_4 : (\overline{x}_2 \vee x_7) \wedge C_5 : (\overline{x}_2 \vee \overline{x}_4)$, to show that a clause $D = (\overline{x}_2 \vee x_6)$ is RUP w.r.t. F, we propagate $F \wedge (x_2 \wedge \overline{x}_6)$.

Propagation derives the corresponding units and corresponding reason clauses: $(\overline{x}_3, C_2), (x_7, C_4), (\overline{x}_4, C_5), (\overline{x}_1, C_1)$, then C_3 is in conflict.

Analysis of the conflict on C_3 yields reasons C_2, C_5, C_1. Note, C_4 is absent because the unit x_7 is not used to derive the conflict on C_3.

The proof is verified once all marked proof steps have been checked. During each RUP check, *core-first* propagation can be used to minimize the core by partitioning propagation into two levels: $Level_1$ (marked core clauses) and $Level_2$ (unmarked clauses). Initially, $Level_1$ is empty while $Level_2$ contains input clauses and proof steps. During

Algorithm 1 Theory-Last Trimming

```
1. Level_1 ← Level_2 ← ∅                          // initialize propagation levels
2. Level_1 ← {C_1, ..., C_r}                      // add input clauses
3. Level_2 ← {T_1, ..., T_k}, {P_1, ..., P_n}     // add theory lemmas and proof steps
4. for i = n down to 1:                           // iterate proof steps backwards
5.    Delete(P_i, Level_1); Delete(P_i, Level_2)  // remove from future propagation
6.    if marked(P_i):                             // skip unmarked proof steps
7.       C ← clause(P_i)                          // clause representing proof step
8.       Units ← {¬ℓ | ℓ ∈ C}                     // negate clause
9.       while propagate(Level_1, Units) ≠ ⊥:     // propagate in level 1
10.         propagatePartial(Level_2, Units)     // propagate in level 2
11.         if no new unit literals:             // nothing new derived
12.            return UNVERIFIED                 // proof check failed
13.      Reasons ← analyze(conflict)              // analyze conflict
14.      for each D ∈ Reasons:
15.         mark(D)                               // mark reason clause
16.         if D ∈ Level_2: move D to Level_1     // promote clause
17. return filter(marked, {T_1, ..., T_k})        // return marked theory lemmas
```

a RUP check, the propagation engine alternates between $Level_1$ and $Level_2$ as follows. First, propagation is performed on $Level_1$. If a conflict is derived, conflict analysis proceeds immediately. Otherwise, $Level_2$ is *partially propagated* until the first new unit is derived, at which point control returns to $Level_1$ to propagate that learned unit. During conflict analysis, any reason clauses originating from $Level_2$ are marked and promoted to $Level_1$ for subsequent RUP checks. This approach attempts to minimize the number of newly marked clauses in each RUP check, thereby reducing the size of the core.

Theory-last trimming (Algorithm 1) modifies core-first trimming by initializing $Level_1$ with all input clauses (line 2) and $Level_2$ with theory lemmas and proof steps (line 3). Consequently, the initial propagations during a RUP check prioritize input clauses (line 9). Only when propagation on $Level_1$ fails to derive a conflict does propagation proceed on $Level_2$; thus, theory lemmas are propagated last. The function $propagatePartial$ propagates until a new unit literal is derived, after which control returns to $Level_1$. Lines $13-16$ mark reason clauses, and the marked (core) theory lemmas are returned on line 17. Example 3 illustrates theory-last propagation.

The resulting core may be larger because it always contains all input clauses, but it often contains fewer theory lemmas. Theory-last trimming always produces a valid core for a valid RUP proof because the propagation engine still has access to all relevant clauses across $Level_1$ and $Level_2$. In the worst case, backward trimming is quadratic in the size of the proof; however, in practice, its runtime is often comparable to the time required to generate the proof.

Example 3. Given F from Example 2, let C_4 and C_5 be theory lemmas. To show D is RUP, theory-last propagation would proceed as follows:

1. Propagate $Level_1$: (C_1, C_2, C_3) with units $x_2, \overline{x}_6$, giving $(\overline{x}_3, C_2)$.
2. No conflict was derived, so partially propagate $Level_2$: (C_4, C_5) with units $x_2, \overline{x}_6, \overline{x}_3$ giving (x_7, C_4).

3. Return to $Level_1$ propagating the new unit x_7.
4. Again, no conflict is derived so partially propagate $Level_2$, giving $(\overline{x}_4, C_5)$.
5. Return to $Level_1$ propagating $\overline{x}_4$ giving $(\overline{x}_1, C_1)$ then a conflict.
6. Analysis gives reasons C_2, C_5, C_1, so theory lemma C_5 would be marked and promoted from $Level_2$ to $Level_1$, with C_4 remaining in $Level_2$.

Iterative Theory Trimming The theory trimming algorithm does not guarantee optimality; algorithms for finding a minimal core require many calls to a SAT solver, in the worst case removing only one clause from the core at a time. Small reductions to the size of the core provide diminishing returns for theories where theory lemmas are easy to justify. *Iterative trimming* feeds the output core back into the trimming pipeline as the new input formula, repeating until the number of core theory lemmas stabilizes. Subsequent iterations benefit from fresh SAT proofs and can remove additional theory lemmas, though the first iteration typically achieves the largest reduction. This approach achieves moderate reductions with fewer calls to a SAT solver and trimming tool.

4.2 Theory Lemma Justification and Checking

Each core theory lemma $l_1 \vee \cdots \vee l_n$ is justified by proving the unsatisfiability of $\neg l_1 \wedge \cdots \wedge \neg l_n$ via CVC5. The justification is checked with CARCARA and optionally returned for proof elaboration. The input to CVC5 is an SMT-LIB query using the original problem's preamble, which contains the declaration of sorts and functions symbols, along with the assertion that negates the lemma. The query below, for example, would be used to justify theory lemma t_7 from CONGEX.

```
<Preamble>
(assert (not (or (= a d) (not (= a b)) (not (= b d)))) )
(check-sat)
```

Rather than justifying lemmas individually, we can process them in batches, similar to the work on propositional proof skeletons [33]. Given a batch $T_1, \ldots, T_k$, their joint validity is established by proving the unsatisfiability of $\overline{T}_1 \vee \cdots \vee \overline{T}_k$. This is constructed as a single SMT-LIB query with the original problem's preamble and an assertion forming a disjunction of the negation of the set of theory lemmas, for example:

```
(assert ( or ( not (t_1)) (not (t_2)) ... (not (t_k)))))
```

A single CARCARA call checks the validity of the entire batch. The batch quality—the difficulty of solving and checking the batch—depends on the number of lemmas in the batch, the relationship between lemmas in the batch, and the underlying theory. Smaller batch sizes can lead to significant overhead when executing CVC5 and CARCARA on thousands of trivial queries, but larger batches can grow increasingly more difficult to prove. Through experimentation, a batch size of 50 was found to balance solver overhead against per-batch difficulty. Lemmas are batched in the order they were learned by the SMT solver, keeping related theory lemmas together and enabling the SMT solver to reuse intermediate results.

Batches may be distributed across multiple cores to parallelize justification. However, the effectiveness of parallelization will still depend on the batch quality, as much harder batches can create a bottleneck. While a similar issue is recurring in parallel SMT solving [41], we did not experience this much.

In our evaluation the majority of theory lemmas were justified quickly, motivating the batch size of 50. However, initial experimentation for other theories, including the theory of bitvectors which require more complex justifications, suggest that smaller batch sizes may lead to faster verification.

4.3 Lemmas Containing Fresh Symbols

Preprocessing in quantified logics may introduce fresh symbols via Skolemization, and these symbols can appear in theory lemmas. Therefore when applying the process above to check lemmas with such symbols, the problem's preamble has to be extended to declare them. This however is not sufficient because the fresh symbol is not unconstrained. For example, in the case of Skolemization, the introduced symbol is a witness for a particular Skolemized quantifier. So these constraints may be required in the query to justify the lemma. This is a general issue that limits e.g. validating Skolemization inferences via external solvers [32, Sec. 1.2].

We can circumvent the problem in our approach because Skolemization in Alethe is justified in an equivalence-preserving way via Hilbert's choice operator $\epsilon x.\, \varphi$, which is characterized via the axiom $\exists x.\varphi[x] \rightarrow \varphi[\epsilon x.\, \varphi]$, i.e., the operator yields a term that makes its bound formula hold [4]. Therefore, CARCARA can use the formulas in the choice operators to constrain the fresh symbols appearing in lemmas. So the validity of a lemma $L[\epsilon x.\, \varphi]$ in the Alethe skeleton is justified by proving the unsatisfiability of the SMT formula $(\exists x.\varphi[x] \rightarrow \varphi[k]) \wedge \neg L[k]$, where k is a fresh symbol of the same type of the choice term.

In full generality symbols introduced via Skolemization may depend on other fresh symbols. For example, when Skolemizing $\exists x_1 x_2.\varphi$, first $\exists x_2.\, \varphi[\epsilon x_1.\, \exists x_2.\, \varphi]$ is derived, and the Skolemization of x_2 will be defined by a formula containing the choice term for x_1. Checking the validity of lemmas containing such choice terms, i.e., $L[k]$ where k is $\epsilon x.\, \varphi[\epsilon_1, \ldots, \epsilon_n]$, with $\epsilon_1, \ldots, \epsilon_n$ being choice terms, requires the constraint to validate L to also characterize k accounting for all possible values that $\epsilon_1, \ldots, \epsilon_n$ may assume. Thus the formula that we generate for such lemmas is $(\forall y_1 \ldots y_n.\, \exists x.\varphi[y_1, \ldots, y_n] \rightarrow \varphi[k]) \wedge \neg L[k]$, requiring quantifier reasoning by the backend solver.

5 Experimental Evaluation

All experiments were performed in the Pittsburgh Supercomputing Center on nodes with 128 cores and 256 GB RAM [9]. We used a $5{,}000$ second timeout for solving and a $5{,}000$ second timeout for proof checking, and 64 experiments were run in parallel per node so each process held approximately 4GB of memory. We report the *penalized average runtime* (PAR2). PAR2 is the average runtime with a two times penalty for timeouts, e.g., a timeout on $5{,}000$ seconds is counted as a $10{,}000$ second runtime.

We considered benchmarks from the 2023 release of the SMT-LIB logics QF_UF, QF_LIA, QF_LRA, and QF_UFLIA , UF, and UFLIA which are representative of the fragment supported by CVC5's Alethe proof production. We exclude benchmarks from the `nec-smt` family in QF_LIA, for which CVC5 introduces fresh terms during preprocessing in a way not yet supported by the Alethe translation. We first present the results for the quantifier-free logics, and then for quantified logics in Section 5.4.

For the evaluation, we first solved each formula using CVC5 without proof production. Unsatisfiable formulas without a timeout were used to compare solving times in Table 1. Next, we further filtered away formulas for which the proof did not include a `prop_unsat` step, i.e., CVC5 solved them purely via preprocessing. These problems for which trimming can occur were used to compare checking times in Table 2.

While we only consider the UF, LIA, and LRA theories, our approach is theory agnostic and can be readily extended to other theories once an Alethe-producing SMT solver is available for it. Extensions of CVC5 to produce Alethe proofs for example for bitvectors and strings are ongoing work, and we expect the impact of our approach to be significant in these theories, where theory reasoning is more expensive.

We used the following configurations in our main experiments, with all proofs checked by CARCARA as described in Section 3:

- CVC5 - Default CVC5 without proof production.
- CVC5-PROOF - Default eager proof production in CVC5.
- CVC5-THEORY-TRIM5 - Proof skeleton framework with 5 iterations of theory trimming and lemma justification performed in batches of 50. Summarized as T-5. Optionally, we could use a single trimming iteration (T-1), use a single lemma per batch (T-5-S), or use standard DRAT-TRIM instead of our theory-trim tool (D-5).

5.1 Solving Time

Solving results (Table 1) show the expected ordering: CVC5-THEORY-TRIM5 is faster on average than CVC5-PROOF but slower than uninstrumented CVC5. Scatter plots (Figure 2) show results for a set of hard problems, with the benchmark selection criteria described below in the parallel checking section. The solving benefit is most pronounced on a group of hard problems in QF_LIA where CVC5-PROOF times out at 5,000 seconds but CVC5-THEORY-TRIM5 solves them in one to two hundred seconds.

Table 1: Solving time comparison on unsatisfiable formulas that CVC5 solves within the timeout (number of formulas shown beneath each theory).

Theory	Config	Timeouts	PAR2	Theory	Config	Timeouts	PAR2
QF_UF	CVC5	–	5	QF_LIA	CVC5	–	42
4,356	CVC5-PR	14	47	2,757	CVC5-PR	127	483
	T-5	1	12		T-5	47	221
QF_LRA	CVC5	–	118	QF_UFLIA	CVC5	–	10
696	CVC5-PR	3	205	190	CVC5-PR	3	161
	T-5	1	141		T-5	2	108

5.2 Sequential Checking

Table 2 compares solving times, checking times, and core theory lemma counts for configurations with proof support. For QF_UF and QF_LIA, the combined solving and checking time under CVC5-THEORY-TRIM5 is often lower than under CVC5-PROOF, demonstrating that iterative trimming and batched justification can rival eager proof production while also providing faster solving. Iterative trimming (CVC5-THEORY-TRIM5) often produces cores with fewer theory lemmas than non-iterative (T-1) or stan-

Table 2: Solving and checking comparison on formulas with a `prop_unsat` step. Solving timeouts count towards the checking PAR2. Average lemma count reports core theory lemmas from solved formulas. Best Core is the number of formulas for which a unique configuration among CVC5-PR, D-5, and T-5 found a core with the fewest theory lemmas, with no values reported for T-1 which has at least as many lemmas as T-5 and T-5-S which has exactly as many lemmas as T-5. *CVC5-PR times out on harder QF_LIA instances with many lemmas, making the average incomparable because the number of theory lemmas for unsolved problems are not included. Excluding the 75 CVC5-PR timeouts, T-5 has an average of 599.

Theory Formulas	Config	# Timeouts		PAR2		Avg. Lemmas	# Best Core
		Solve	Check	Solve	Check		
QF_UF	CVC5	–	–	4	–	–	–
4,154	CVC5-PR	13	0	46	33	2,510	17
	T-1	–	9	10	44	1,662	–
	D-5	–	10	10	45	1,562	604
	T-5-S	–	78	10	543	1,549	–
	T-5	–	10	10	43	1,549	3,011
QF_LRA	CVC5	–	–	129	–	–	–
639	CVC5-PR	2	0	207	38	4,057	10
	T-1	–	5	138	209	2,212	–
	D-5	–	5	138	194	1,698	99
	T-5-S	–	27	138	871	1,686	–
	T-5	–	5	138	191	1,686	239
QF_LIA	CVC5	–	–	27	–	–	–
1,843	CVC5-PR	75	0	440	414	*672	6
	T-1	–	31	69	250	4,903	–
	D-5	–	32	69	258	4,895	94
	T-5-S	–	94	69	573	4,894	–
	T-5	–	32	69	256	4,894	162
QF_UFLIA	CVC5	–	–	6	–	–	–
106	CVC5-PR	0	0	6	0	1,105	0
	T-1	–	0	6	6	589	–
	D-5	–	0	6	6	470	15
	T-5-S	–	0	6	236	462	–
	T-5	–	0	6	5	462	29

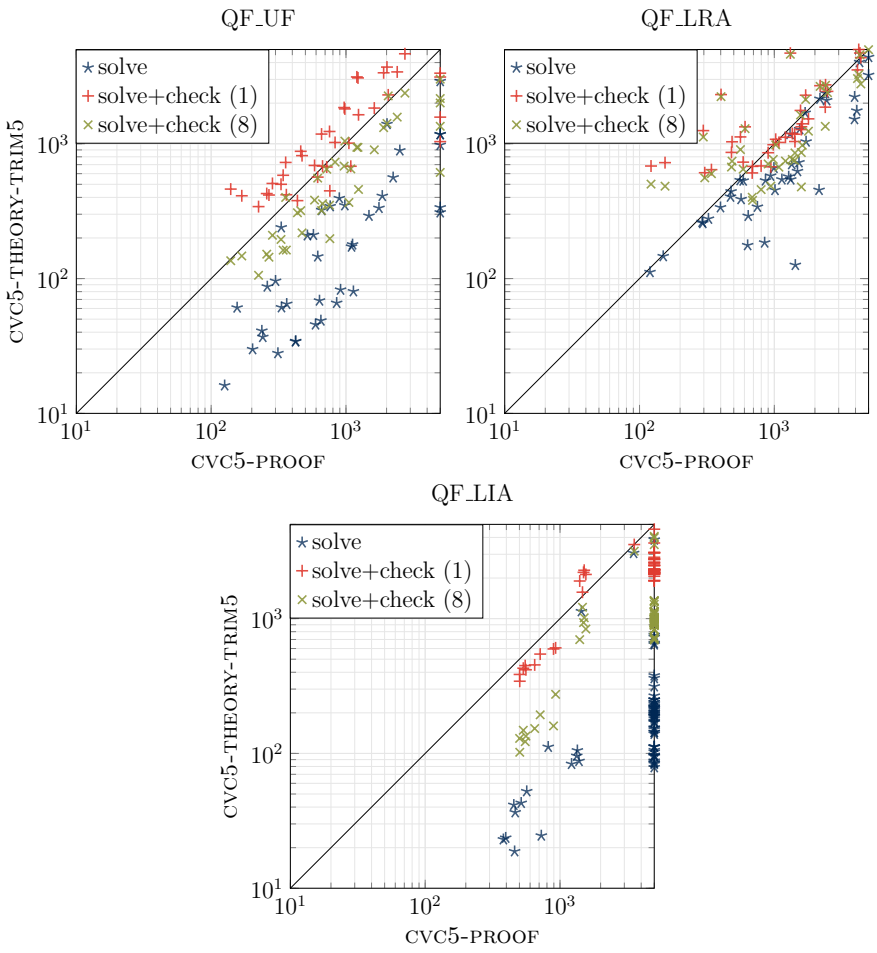

Fig. 2: Scatter plots showing the solving times and the full solving + checking times comparing CVC5-PROOF with a single core and CVC5-THEORY-TRIM5 with (1) or (8) cores. Benchmarks are the filtered set described in the parallel checking evaluation.

dard DRAT-TRIM (D-5), with CVC5-PROOF cores almost never containing fewer theory lemmas than our configurations. Mean lemma reduction rates with CVC5-THEORY-TRIM5 are 54% (QF_UF), 18% (QF_LRA), 21% (QF_LIA), 39% (QF_UFLIA). On average, between $30 \sim 45\%$ of the propositional formula prior to trimming consists of theory lemmas.

The benefit of iterative trimming varies by theory. In QF_LIA, iterative trimming produces smaller cores yet higher checking times, suggesting the trimming cost exceeds the savings from fewer justifications. In QF_UF, QF_LRA, or QF_UFLIA iterative trimming reduces total checking time. Sequential single lemma elaboration (T-5-S) performs poorly across all theories, underscoring the importance of batching.

Table 3: Number of formulas successfully checked for CVC5-PR on a single core, CVC5-THEORY-TRIM5 on a single core (1-core) and CVC5-THEORY-TRIM5 on 8 cores (8-core), and PAR2 scores for checking time and the speedup on CVC5-THEORY-TRIM5.

		# Checked			Checking PAR2			
Theory	Formulas	CVC5-PR	1-core	8-core	CVC5-PR	1-core	8-core	Speedup
QF_UF	40	34	38	39	1,565	1,596	722	2.2
QF_LRA	40	38	39	40	560	1,067	584	1.8
QF_LIA	70	15	62	65	7,883	2,819	1,487	1.9

5.3 Parallel Checking

We applied parallel batch justification across 8 cores to formulas with at least a 300 seconds checking time and at least 400 core theory lemmas when using the sequential version of CVC5-THEORY-TRIM5 . We chose the value of 400 so that each core can process at least one batch of size 50. No formula in QF_UFLIA met this criterion.

Table 3 shows for both QF_UF and QF_LIA, CVC5-THEORY-TRIM5 with 8 cores achieves faster solving-plus-checking than CVC5-PROOF, in addition to faster sequential solving. For QF_LRA, solving-time gains are moderate, resulting in a combined solving-plus-checking time that is often slightly worse than CVC5-PROOF. Note, there is no implementation for parallelizing CARCARA without our proof skeleton framework, so the CVC5-PROOF configuration can only be run on a single core. These results highlight the benefit of skeleton checking on difficult problems containing many theory lemmas for which proof logging is computationally expensive.

5.4 Theories with Quantifiers

Results for UF and UFLIA are summarized in Table 4. The large difference in checking PAR2 is mostly due to timeouts. As discussed in Section 4.2, in quantified logics, justifying some lemmas may require solving quantified problems, and SMT solvers are well known to be incomplete and unstable in quantified logics [42]. So unfortunately timeouts in lemma checking with CVC5 can be expected.

Furthermore, there is less gain in the solving time, or no gain. This is explained by theory reasoning for quantified lemmas in SMT being done mostly via instantiation [34], for which the overhead of proof production is minimal (how the instance was found is irrelevant; only that it is a sound instantiation needs to be justified). Moreover, SMT solvers, and CVC5 is no different, generate instances mostly in a non-goal oriented way, i.e., they are not derived to refine the search directly, but rather on the expectation that they *may* be useful. As a result, solvers generate very large numbers of useless instances, which in our approach are all logged in the skeleton, and this overhead may be one reason our solving is often slightly slower than CVC5-PR. Finally, we observe that problems in UF/UFLIA mostly do not have as deep a ground structure as problems in the quantifier-free logics, and with less ground reasoning less overhead is added by producing proofs, which we would have saved.

Table 4: Timeouts, PAR2 scores, and number of core lemmas for quantified theories UF and UFLIA comparing CVC5, CVC5-PR, and T-5. Description of PAR2, Avg. Lemmas, and Best Core found in caption of Table 2.

Theory	Config	# Timeouts		PAR2		Avg. Lemmas	# Best Core
Formulas		Solve	Check	Solve	Check		
UF	CVC5	–	–	57	–	–	–
679	CVC5-PR	0	0	53	0	38	15
	T-5	–	14	53	210	33	659
UFLIA	CVC5	–	–	14	–	–	–
1,411	CVC5-PR	1	0	21	7	91	224
	T-5	–	12	14	88	82	1,133

6 Related Work

Several works have used clausal proofs for SMT with varying success [14,31]. The work on the *extended* DRAT (*e*DRAT) proof system [18] most closely relates to our work. They modified CVC5 to produce *e*DRAT proofs for QF_UF and QF_LRA consisting of the SAT proof steps as well as unjustified theory lemma steps. Then, after trimming the SAT proof with DRAT-TRIM, their VALIDO toolchain elaborated proof steps and checked them with theory-specific RUST and Lean [26] tools respectively. One key difference is that we leverage existing tools, avoiding the cost of developing theory-specific elaborators and checkers. This also allows us to support preprocessing proofs, a crucial component missing in *e*DRAT. Next, we focus on iterative proof trimming using our modified DRAT-TRIM with theory-last propagation. Finally, our relaxed approach with batch justification facilitates parallelization in checking. The VALIDO toolchain is not open source so we cannot compare our tools directly.

Proof skeleton frameworks have also been explored in other domains including constraint programming [16], with the focus of reducing proof logging overhead during solving.

Core extraction has also been explored in the context of SMT, with early ideas stemming from work on lemma-lifting [10]. Notably, these approaches use off-the-shelf core extractors, whereas our approach attempts to minimize theory lemmas in the core.

7 Conclusion and Future Work

We presented a general approach for producing and checking SMT proof skeletons. External SAT reasoning enables improved theory trimming, and batched parallel theory justification improves runtimes on quantifier-free logics. This framework requires only that an SMT solver logs preprocessing and clausification proofs together with theory lemma holes, making it accessible to solvers beyond CVC5 that lack full proof support.

Future work will explore harder theories including bit-vectors and mechanisms for incorporating preprocessing steps into the trimming and skeleton checking procedures. In addition, the skeleton format may be extended to support partial or fully justified theory lemmas when including such information does not compromise solver runtime.

Acknowledgments. This work was partially supported by a gift from Amazon Web Services, the National Science Foundation (NSF) under grant CCF-2415773, and by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-24-9-1000 and FA8750-24-2-1001. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of DARPA. Joseph E. Reeves was supported by a fellowship award under contract FA9550-21-F-0003 through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program, sponsored by the Air Force Research Laboratory (AFRL), the Office of Naval Research (ONR) and the Army Research Office (ARO).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Andreotti, B., Lachnitt, H., Barbosa, H.: Carcara: An efficient proof checker and elaborator for SMT proofs in the alethe format. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for Construction and Analysis of Systems (TACAS), Part I. Lecture Notes in Computer Science, vol. 13993, pp. 367–386. Springer (2023). https://doi.org/10.1007/978-3-031-30823-9_19
2. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: Bjørner, N., Gurfinkel, A. (eds.) Formal Methods In Computer-Aided Design (FMCAD). pp. 1–9. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8602994
3. Barbosa, H., Barrett, C.W., Cook, B., Dutertre, B., Kremer, G., Lachnitt, H., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Tinelli, C., Zohar, Y.: Generating and exploiting automated reasoning proof certificates. Commun. ACM **66**(10), 86–95 (2023). https://doi.org/10.1145/3587692
4. Barbosa, H., Blanchette, J.C., Fleury, M., Fontaine, P.: Scalable fine-grained proofs for formula processing. Journal of Automated Reasoning **64**(3), 485–510 (2020). https://doi.org/10.1007/s10817-018-09502-y
5. Barbosa, H., Reynolds, A., Kremer, G., Lachnitt, H., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Viswanathan, A., Viteri, S., Zohar, Y., Tinelli, C., Barrett, C.W.: Flexible proof production in an industrial-strength SMT solver. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) International Joint Conference on Automated Reasoning (IJCAR). Lecture Notes in Computer Science, vol. 13385, pp. 15–35. Springer (2022). https://doi.org/10.1007/978-3-031-10769-6_3
6. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 1267–1329. IOS Press (2021). https://doi.org/10.3233/FAIA201017
7. Biere, A., Faller, T., Fazekas, K., Fleury, M., Froleyks, N., Pollitt, F.: Cadical 2.0. In: Computer Aided Verification (CAV). pp. 133–152. Springer (2024), https://doi.org/10.1007/978-3-031-65627-9_7
8. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An Open, Trustable and Efficient SMT-Solver. In: Schmidt, R.A. (ed.) Proc. Conference on Automated Deduction (CADE). vol. 5663, pp. 151–156. Springer (2009). https://doi.org/10.1007/978-3-642-02959-2_12
9. Brown, S.T., Buitrago, P., Hanna, E., Sanielevici, S., Scibek, R., Nystrom, N.A.: Bridges-2: A Platform for Rapidly-Evolving and Data Intensive Research, pp. 1–4. Association for Computing Machinery, New York, NY, USA (2021), https://doi.org/10.1145/3437359.3465593

10. Cimatti, A., Griggio, A., Sebastiani, R.: Computing small unsatisfiable cores in satisfiability modulo theories. J. Artif. Int. Res. **40**(1), 701–728 (Jan 2011), https://dl.acm.org/doi/abs/10.5555/2016945.2016964
11. Coltellacci, A., Merz, S., Dowek, G.: Reconstruction of SMT proofs with lambdapi. In: Reger, G., Zohar, Y. (eds.) Proceedings of the 22nd International Workshop on Satisfiability Modulo Theories co-located with the Computer Aided Verification (CAV), Montreal, Canada, July, 22-23, 2024. CEUR Workshop Proceedings, vol. 3725, pp. 13–23. CEUR-WS.org (2024), https://ceur-ws.org/Vol-3725/paper8.pdf
12. Ekici, B., Mebsout, A., Tinelli, C., Keller, C., Katz, G., Reynolds, A., Barrett, C.W.: Smtcoq: A plug-in for integrating SMT solvers into coq. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification (CAV). vol. 10427, pp. 126–133. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_7
13. Farkas, G.: A Fourier-féle mechanikai elv alkamazásai. Mathematikaiés Természettudományi Értesítö **12**, 457–472 (1894), reference from Schrijver's Combinatorial Optimization textbook (Hungarian)
14. Feng, N., Hu, A.J., Bayless, S., Iqbal, S.M., Trentin, P., Whalen, M., Pike, L., Backes, J.: DRAT proofs of unsatisfiability for SAT modulo monotonic theories. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 3–23. Springer Nature Switzerland, Cham (2024), https://doi.org/10.1007/978-3-031-57246-3_1
15. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. pp. 125–128. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), https://doi.org/10.1007/978-3-642-37036-6_8
16. Flippo, M., Sidorov, K., Marijnissen, I., Smits, J., Demirović, E.: A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers. In: Shaw, P. (ed.) Constraint Programming (CP). Leibniz International Proceedings in Informatics (LIPIcs), vol. 307, pp. 11:1–11:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2024), https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2024.11
17. Fontaine, P., Marion, J., Merz, S., Nieto, L.P., Tiu, A.F.: Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. In: Hermanns, H., Palsberg, J. (eds.) Tools and Algorithms for Construction and Analysis of Systems (TACAS). vol. 3920, pp. 167–181. Springer (2006), https://doi.org/10.1007/11691372_11
18. Hitarth, S., Codel, C., Lachnitt, H., Dutertre, B.: Extending DRAT to SMT (2024), https://www.amazon.science/publications/extending-drat-to-smt
19. Hoenicke, J., Schindler, T.: A simple proof format for SMT. In: Déharbe, D., Hyvärinen, A.E.J. (eds.) International Workshop on Satisfiability Modulo Theories (SMT). CEUR Workshop Proceedings, vol. 3185, pp. 54–70. CEUR-WS.org (2022), http://ceur-ws.org/Vol-3185/paper9527.pdf
20. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: International Joint Conference on Automated Reasoning (IJCAR). LNCS, vol. 7364, pp. 355–370. Springer (2012), https://doi.org/10.1007/978-3-642-31365-3_28
21. Katz, G., Barrett, C.W., Tinelli, C., Reynolds, A., Hadarean, L.: Lazy proofs for DPLL(T)-based SMT solvers. In: Piskac, R., Talupur, M. (eds.) Formal Methods In Computer-Aided Design (FMCAD). pp. 93–100. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886666
22. Lachnitt, H., Fleury, M., Barbosa, H., Jakpor, J., Andreotti, B., Reynolds, A., Schurr, H., Barrett, C.W., Tinelli, C.: Improving the SMT proof reconstruction pipeline in isabelle/hol. In: Forster, Y., Keller, C. (eds.) Interactive Theorem Proving (ITP). LIPIcs, vol. 352, pp. 26:1–26:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2025). https://doi.org/10.4230/LIPICS.ITP.2025.26

23. Lattuada, A., Hance, T., Bosamiya, J., Brun, M., Cho, C., LeBlanc, H., Srinivasan, P., Achermann, R., Chajed, T., Hawblitzel, C., Howell, J., Lorch, J.R., Padon, O., Parno, B.: Verus: A practical foundation for systems verification. In: Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles. p. 438–454. SOSP '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3694715.3695952
24. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). pp. 348–370. Springer Berlin Heidelberg, Berlin, Heidelberg (2010), https://doi.org/10.1007/978-3-642-17511-4_20
25. Mann, M., Irfan, A., Lonsing, F., Yang, Y., Zhang, H., Brown, K., Gupta, A., Barrett, C.: Pono: a flexible and extensible SMT-based model checker. In: Computer Aided Verification (CAV). pp. 461–474. Springer (2021), https://doi.org/10.1007/978-3-030-81688-9_22
26. de Moura, L., Ullrich, S.: The lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) Proc. Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 12699, pp. 625–635. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_37
27. de Moura, L.M., Bjørner, N.: Proofs and refutations, and Z3. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) Workshops. CEUR Workshop Proceedings, vol. 418. CEUR-WS.org (2008)
28. Mukherjee, R., Kroening, D., Melham, T.: Hardware verification using software analyzers. In: 2015 IEEE Computer Society Annual Symposium on VLSI. pp. 7–12 (2015). https://doi.org/10.1109/ISVLSI.2015.107
29. Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: Giesl, J. (ed.) Rewriting Techniques and Applications (RTA). vol. 3467, pp. 453–468. Springer (2005), https://doi.org/10.1007/978-3-540-32033-3_33
30. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). J. ACM **53**(6), 937–977 (Nov 2006). https://doi.org/10.1145/1217856.1217859
31. Otoni, R., Blicha, M., Eugster, P., Hyvärinen, A.E.J., Sharygina, N.: Theory-specific proof steps witnessing correctness of smt executions. In: ACM/IEEE Design Automation Conference (DAC). pp. 541–546 (2021). https://doi.org/10.1109/DAC18074.2021.9586272
32. Rawson, M., Voronkov, A., Schoisswohl, J., Komel, A.P.: Ground truth: Checking vampire proofs via satisfiability modulo theories. In: Barrett, C.W., Waldmann, U. (eds.) Proc. Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 15943, pp. 136–149. Springer (2025). https://doi.org/10.1007/978-3-031-99984-0_8
33. Reeves, J.E., Kiesl-Reiter, B., Heule, M.J.H.: Propositional proof skeletons. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 329–347. Springer (2023), https://doi.org/10.1007/978-3-031-30823-9_17
34. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for Construction and Analysis of Systems (TACAS), Part II. vol. 10806, pp. 112–131. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_7
35. Rungta, N.: A billion SMT queries a day (invited paper). In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification (CAV), Part I. Lecture Notes in Computer Science, vol. 13371, pp. 3–18. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_1
36. Schrijver, A.: Theory of linear and integer programming. John Wiley & Sons (1998)
37. Schurr, H., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: Towards a generic SMT proof format (extended abstract). CoRR **abs/2107.02354** (2021), https://arxiv.org/abs/2107.02354

38. Schurr, H., Fleury, M., Desharnais, M.: Reliable reconstruction of fine-grained proofs in a proof assistant. In: Platzer, A., Sutcliffe, G. (eds.) Proc. Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 12699, pp. 450–467. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_26
39. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: cake_lpr: Verified propagation redundancy checking in CakeML. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part II. LNCS, vol. 12652, pp. 223–241 (2021). https://doi.org/https://doi.org/10.1007/s10009-022-00690-y
40. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 8561, pp. 422–429 (2014), https://doi.org/10.1007/978-3-319-09284-3_31
41. Wilson, A., Nötzli, A., Reynolds, A., Cook, B., Tinelli, C., Barrett, C.W.: Partitioning strategies for distributed SMT solving. In: Nadel, A., Rozier, K.Y. (eds.) Formal Methods In Computer-Aided Design (FMCAD). pp. 199–208. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_28
42. Zhou, Y., Bosamiya, J., Takashima, Y., Li, J., Heule, M., Parno, B.: Mariposa: Measuring SMT instability in automated program verification. In: Nadel, A., Rozier, K.Y. (eds.) Formal Methods In Computer-Aided Design (FMCAD). pp. 178–188. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_26