DRAT-based Bit-Vector Proofs in CVC4*

Alex Ozdemir, Aina Niemetz, Mathias Preiner,
Yoni Zohar, and Clark Barrett

Stanford University

Abstract. Many state-of-the-art Satisfiability Modulo Theories (SMT) solvers for the theory of fixed-size bit-vectors employ an approach called bit-blasting, where a given formula is translated into a Boolean satisfiability (SAT) problem and delegated to a SAT solver. Consequently, producing bit-vector proofs in an SMT solver requires incorporating SAT proofs into its proof infrastructure. In this paper, we describe three approaches for integrating DRAT proofs generated by an off-the-shelf SAT solver into the proof infrastructure of the SMT solver CVC4 and explore their strengths and weaknesses. We implemented all three approaches using CryptoMiniSat as the SAT back-end for its bit-blasting engine and evaluated performance in terms of proof-production and proof-checking.

1 Introduction

The majority of Satisfiability Modulo Theories (SMT) solvers for the theory of fixed-size bit-vectors employ an approach called bit-blasting. That is, an input formula is first simplified, and then eagerly translated into propositional logic and handed to a Boolean satisfiability (SAT) solver. Thus, when producing a proof of unsatisfiability for a given bit-vector input, it is crucial to obtain the unsatisfiability proof from the SAT solver back-end and incorporate it into a possibly larger SMT proof. The bit-blasting engine of the SMT solver CVC4 [1] currently supports several SAT solvers as back-ends. Producing proofs, however, is only supported with a modified version of MiniSat [6], which was extended to record resolution proofs that can be embedded into CVC4 proofs [9]. This custom MiniSat implementation requires extra maintenance and is less competitive than more recent off-the-shelf SAT solvers.

In recent years, the *Delete Resolution Asymmetric Tautologies* (DRAT) proof system [18], a generalization of *extended resolution* (ER) [17], has become the de facto standard for validating unsatisfiability in SAT solvers. Using a state-of-the-art SAT solver with support for DRAT inside CVC4 would allow CVC4 to use the latest, best SAT techniques while being able to produce bit-vector proofs without additional customization of the SAT solver code. However, in order to support this, CVC4 must be able to incorporate DRAT proofs into its proof infrastructure, which is based on LFSC, an extension of *Edinburgh's*

* This work was supported in part by DARPA (N66001-18-C-4012 and FA8650-18-2-7861), NSF (1814369) and the Stanford Center of Blockchain Research.

Logical Framework [10] (LF) with functional programs called *side conditions* (see [15] for more details on LFSC and [2] for a more general survey of proofs in SMT-solvers). In this paper, we examine three approaches for translating DRAT proofs to LFSC: (*i*) a direct translation from DRAT to LFSC proofs, (*ii*) an intermediate translation from DRAT to Linear RAT (LRAT) proofs [4], and (*iii*) an intermediate translation from DRAT to ER proofs [11], which are then translated to LFSC. The produced proofs can be independently checked by any proof checker for LFSC. We describe the implementation of these three approaches for generating bit-vector proofs in CVC4, discuss their strengths and weaknesses, and evaluate their performance in terms of proof production and proof checking.

2 From DRAT to LFSC

We briefly review the definitions relevant to the proof systems DRAT, LRAT, and ER. More details can be found in [4,11,18].

A *literal* is either a propositional variable or its negation. A *clause* is a disjunction of literals, sometimes interpreted as a set of literals. A clause is *unit* if it is a singleton. A *formula* in conjunctive normal form (CNF) is a conjunction of clauses, sometimes interpreted as a set of clauses.

A proof for formula F in CNF is a sequence $\pi = C_1, \ldots, C_m, I_{m+1}, \ldots, I_n$ with clauses $C_1, \ldots, C_m \in F$ and pairs I_i of the form $\langle \diamond, X \rangle$, where either $\diamond \in \{a, d\}$ and X is a clause, or $\diamond = e$ and X is a CNF formula. Letters a, d, and e indicate addition, deletion, and extension, respectively. Sequence π induces a sequence of CNF formulas $F_0, \ldots, F_n$ such that $F_i = \{C_1, \ldots, C_i\}$ for $1 \leq i \leq m$, and for $i > m$, $F_i = F_{i-1} \cup \{C\}$ if $I_i = \langle a, C \rangle$, $F_i = F_{i-1} \setminus \{C\}$ if $I_i = \langle d, C \rangle$, and $F_i = F_{i-1} \cup G$ if $I_i = \langle e, G \rangle$. It is a proof of unsatisfiability of F if $\emptyset \in F_n$.

A proof π of unsatisfiability of F is a valid ER proof if every I_i is either: (i) $\langle a, C \cup D \rangle$, where $C \cup \{p\}$ and $D \cup \{\overline{p}\} \in F_{i-1}$ for some p; or (ii) $\langle e, G \rangle$, where G is the CNF translation of $x \leftrightarrow \varphi$ with x a fresh variable and φ some formula over variables occurring in F_{i-1}. Proof π is a valid DRAT proof if every I_i is either $\langle d, C \rangle$ or $\langle a, C \rangle$ and for the latter, one of the following holds:

- C is a *reverse unit propagation* (RUP) [8] in F_{i-1}, i.e., the empty clause is derivable from F_{i-1} and the negations of literals in C using unit propagation.
- C is a *resolution asymmetric tautology* (RAT) in F_{i-1}, i.e., there is some $p \in C$ such that for every $D \cup \{\overline{p}\} \in F_{i-1}$, $C \cup D$ is a RUP in F_{i-1}. If C is a RAT but not a RUP, we call it a *proper* RAT.

LRAT proofs are obtained from DRAT proofs by allowing a third element in each I_i that includes hints regarding the clauses and literals that are relevant for verifying the corresponding proof step.

2.1 Integration Methods

The *Logical Framework with Side Conditions* (LFSC) [15] is a statically and dependently typed Lisp-style meta language based on the Edinburgh Logical

```
(program is_specified_drat_proof ((f cnf) (proof DRATProof)) bool
  (match proof
     (DRATProofn (cnf_has_bottom f))
     ((DRATProofa c p) (
        match (is_rat f c) (tt (is_specified_drat_proof (cnfc c f) p)) (ff ff)))
     ((DRATProofd c p) (is_specified_drat_proof (cnf_remove_clause c f) p))))
```

Fig. 1. Side condition for checking a specified DRAT proof. The side conditions `cnf_has_bottom`, `is_rat`, and `cnf_remove_clause` are defined in the same signature. Type `cnfc` is a constructor for CNF formulas, and is defined in a separate signature.

Framework (LF) [10]. It can be used to define logical systems and check proofs written within them by way of the Curry-Howard correspondence. Like LF, LFSC is a framework in which axioms and derivation rules can be defined for multiple theories and their combination. LFSC additionally adds the notion of *side conditions* as functional programs, which can restrict the application of derivation rules. This is convenient for expressing proof-checking rules that are computational in nature. In order to use DRAT proofs in CVC4, the proofs need to be representable in LFSC. We consider the following three approaches for integrating DRAT proofs into LFSC.

Checking DRAT Proofs in LFSC. This approach directly translates DRAT proofs into LFSC. It requires creating a signature for DRAT in LFSC, which essentially is an LFSC implementation of a DRAT checker.

Checking LRAT Proofs in LFSC. LRAT proofs include hints to accelerate unit propagation while proof checking. We use the tool DRAT-trim [18] to translate DRAT proofs into the LRAT format and then check the resulting proof with an LRAT LFSC signature.

Checking ER Proofs in LFSC. This approach aims at further reducing computational overhead during proof checking by translating a DRAT proof into an ER proof with the tool drat2er [11]. The ER proof is then translated to LFSC and checked with an ER LFSC signature.

3 LFSC Signatures

In this section, we describe the main characteristics of the LFSC signatures[1] that we have defined for checking DRAT, LRAT, and ER proofs.

The *LFSC DRAT signature* makes extensive use of side conditions to express processes such as unit propagation and the search for the resolvents of a proper RAT. Because of the divergence between operational and specified DRAT and the resulting ambiguity (see [13] for further details), our signature accepts both kinds of proofs. Figure 1 shows the main side condition that is used to check a DRAT proof. Though we do not explain the LFSC syntax in detail here due to lack of space, the general idea can be easily understood. Given a proof candidate

[1] `https://github.com/CVC4/CVC4/blob/master/proofs/signatures/`

```
(declare definition (! x var (! p lit (! ls lit_list type))))
(declare decl_definition
   (! p lit (! ls lit_list (! pf_continuation
          (! x var (! def (definition x p ls) (holds empty_clause)))
   (holds empty_clause)))))
```

Fig. 2. Derivation rules for checking that a clause constitutes an extension.

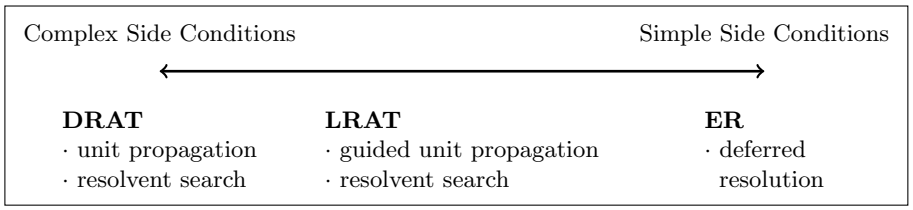

Fig. 3. Side conditions across our signatures.

`proof`, it covers three cases: (*i*) the proof is empty and the working formula includes a contradiction; (*ii*) the proof begins with an addition of a (proper or improper) RAT; or (*iii*) the proof begins with a deletion of some clause. In (*ii*) and (*iii*), the same side condition is recursively called on the rest of `proof`, with an updated working formula. In (*ii*), side condition `is_rat` checks whether the added clause is indeed a RAT via resolvent search and unit propagation.

The *LFSC LRAT signature* is similar in nature, and also makes extensive use of side conditions—albeit less computationally expensive ones. In particular, this signature uses hints provided in the LRAT proofs to accelerate unit propagation.

The *LFSC ER signature* is an extension of the LFSC signature for resolution proofs that is currently employed by CVC4. It implements *deferred resolution* to quickly check large resolution proofs using only a single side condition [15]. The signature extends resolution in order to check the ER proofs produced by the `drat2er` tool. These proofs feature extensions of the form $x \leftrightarrow (p \vee (l_1 \wedge l_2 \wedge \cdots \wedge l_k))$, where x is fresh and p and l_i are not. Our signature includes side-condition-free rules for introducing such extensions and translating them to CNFs of the form $\left\{\{x, \overline{p}\}, \left\{x, \overline{l_1}, \ldots, \overline{l_k}\right\}, \{\overline{x}, p, l_1\}, \ldots, \{\overline{x}, p, l_k\}\right\}$. The `decl_definition` rule in Figure 2 is used to introduce these extensions. Its first two arguments are literal p and the list of literals l_i (denoted as `ls` of type `lit_list`) from the definition. The third argument is a function that receives a fresh variable `x` and connects the introduced definition to the rest of the proof. Figure 3 illustrates the difference in side conditions between the three signatures.

4 Workflow: From CVC4 to LFSC

Figure 4 shows the general workflow for incorporating DRAT proofs in the LFSC proof infrastructure of CVC4 after bit-blasting. LFSC proofs for the bit-blasting step are described in [9]. A given bit-vector formula is bit-blasted to SAT, and the resulting CNF is then sent to the underlying SAT solver. We use DRAT-trim to trim the original formula, optimize the proof produced by the SAT

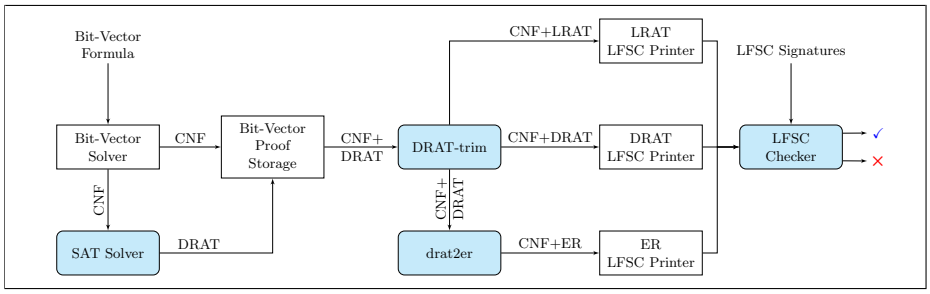

Fig. 4. Producing and checking LFSC proofs in DRAT, LRAT and ER proof systems in CVC4. White boxes are CVC4-internal components; blue boxes are external libraries.

solver, and optionally produce an LRAT proof that is forwarded to the LRAT LFSC pipeline. In case of DRAT LFSC proofs, we can directly use the optimized proof and formula emitted by DRAT-trim. For ER LFSC proofs, we first use drat2er to translate the optimized DRAT proof into an ER proof, which is then sent to the ER LFSC pipeline. The result of each pipeline is an LFSC proof in the corresponding proof system, which can be checked with the corresponding signature (see Section 3) using the LFSC proof checker. Note that prior to bit-blasting, the input is usually simplified via rewriting and other preprocessing techniques, for which CVC4 currently does not produce proofs. The addition of such proofs is left as future work and orthogonal to incorporating DRAT proofs from the SAT solver back-end, which is the focus of this paper.

5 Experiments

We implemented the three approaches described in Section 2.1 in CVC4 using CryptoMiniSat 5.6 [14] as the SAT back-end. We compared them against the resolution-based proof machinery currently employed in CVC4 and evaluated our techniques on all 21125 benchmarks from the quantifier-free bit-vector logic QF_BV of SMT-LIB [3] with status *unsat* or *unknown*. All experiments were performed on a cluster with Intel Xeon E5-2620v4 CPUs with 2.1GHz and 128GB of memory. We used a time limit of 600 seconds (CPU time) and a memory limit of 32GB for each solver/benchmark pair. For each error or memory-out, we added a penalty of 600 seconds.

Proof System	**solve**		**log**		**log,prod**		**log,prod,check**	
	#	[s]	#	[s]	#	[s]	#	[s]
Resolution	20480	464k	20396	524k	**14217**	**4400k**	13510	4973k
DRAT	**20767**	**283k**	**20736**	**319k**	14098	4492k	12563	5616k
LRAT					14088	4500k	12877	5370k
ER					14035	4565k	**13782**	**4886k**

Table 1. Impact of proof logging, production, and checking on # solved problems.

Table 1 shows the results for the Resolution approach with MiniSat, and the DRAT, LRAT and ER approaches with CryptoMiniSat. For each, we ran the following four configurations: proofs disabled (*solve*), proof logging enabled (*log*), proof production enabled (*prod*), and proof checking enabled (*check*). Proof logging records proof-related information but does not produce the actual proof, e.g., when producing DRAT proofs, proof logging stores the DRAT proof from the SAT-solver, which is only translated to LFSC during proof production. In the *solve* configuration, the DRAT-based approaches (using CryptoMiniSat) solve 287 more problems than the Resolution approach (which uses CVC4's custom version of MiniSat). This indicates that the custom version of MiniSat was a bottleneck for solving. In the *log* configuration, the DRAT-based approaches solve 31 fewer problems than in the *solve* configuration; and in the *prod* configuration the DRAT-based approaches produce proofs for ∼6600 fewer problems. This indicates that the bottleneck in the DRAT-based approaches is the translation of DRAT to LFSC. For all approaches, about 30% of the solved problems require more than 8GB of memory to produce a proof, showing that proof production can in general be very memory-intensive. Finally, with proof checking enabled, the ER-based approach outperforms all other approaches. Note that in ∼270 cases, CryptoMiniSat produced a DRAT proof that was rejected by DRAT-trim, which we counted as error. Further, for each *check* configuration, our LFSC checker reported ∼200 errors, which are not related to our new approach. Both issues need further investigation.

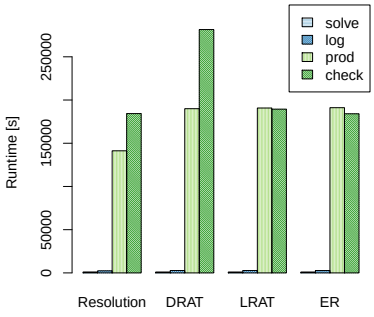

Fig. 5. Runtime distribution on 12539 commonly proved problems.

Figure 5 shows the runtime distribution for all approaches and configurations over the commonly proved problems (12539 in total). The runtime overhead of proof production for the DRAT-based approaches is 1.35 times higher compared to resolution. This is due to the fact that we post-process the DRAT-proof prior to translating it to LFSC, which involves writing temporary files and calling external libraries. The proof checking time correlates with the complexity of the side conditions (see Figure 3), where ER clearly outperforms DRAT.

6 Conclusion

We have described three approaches for integrating DRAT proofs in LFSC, which enable us to use off-the-shelf SAT solvers as the SAT back-end for the bit-blasting engine of CVC4 while supporting bit-vector proofs. For future work, we plan to reduce the complexity of the side conditions in the DRAT and LRAT signatures and the proof production overhead in the translation workflows. We also plan to add support for the new signatures in SMTCoq [7], a tool that increases automation in Coq [16] using proofs generated by CVC4. In a more applicative

direction, we plan to explore the potential DRAT proofs in SMT-solvers may have in the proof-carrying code paradigm [12], as well as its recent variant in blockchains, namely proof-carrying smart contracts [5].

References

1. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proceedings of the 23rd International Conference on Computer Aided Verification. pp. 171–177. CAV'11, Springer-Verlag (2011), http://dl.acm.org/citation.cfm?id=2032305.2032319
2. Barrett, C., de Moura, L., Fontaine, P.: Proofs in satisfiability modulo theories. In: Delahaye, D., Woltzenlogel Paleo, B. (eds.) All about Proofs, Proofs for All, Mathematical Logic and Foundations, vol. 55, pp. 23–44. College Publications, London, UK (2015)
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
4. Cruz-Filipe, L., Heule, M.J., Hunt, W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: International Conference on Automated Deduction. pp. 220–236. Springer (2017)
5. Dickerson, T.D., Gazzillo, P., Herlihy, M., Saraph, V., Koskinen, E.: Proof-carrying smart contracts. In: Financial Cryptography Workshops. Lecture Notes in Computer Science, vol. 10958, pp. 325–338. Springer (2018)
6. Eén, N., Sörensson, N.: An extensible sat-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing. pp. 502–518. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
7. 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: CAV (2). Lecture Notes in Computer Science, vol. 10427, pp. 126–133. Springer (2017)
8. Gelder, A.V.: Verifying RUP proofs of propositional unsatisfiability. In: International Symposium on Artificial Intelligence and Mathematics (ISAIM). Springer (2008)
9. Hadarean, L., Barrett, C.W., Reynolds, A., Tinelli, C., Deters, M.: Fine grained SMT proofs for the theory of fixed-width bit-vectors. In: LPAR. Lecture Notes in Computer Science, vol. 9450, pp. 340–355. Springer (2015)
10. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM **40**(1), 143–184 (1993)
11. Kiesl, B., Rebola-Pardo, A., Heule, M.J.: Extended Resolution Simulates DRAT. In: International Joint Conference on Automated Reasoning. pp. 516–531. Springer (2018)
12. Necula, G.C.: Proof-carrying code. In: POPL. pp. 106–119. ACM Press (1997)
13. Pardo, A.R., Biere, A.: Two flavors of drat. In: Berre, D.L., Järvisalo, M. (eds.) Proceedings of Pragmatics of SAT 2015 and 2018. EPiC Series in Computing, vol. 59, pp. 94–110. EasyChair (2019)
14. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: SAT. Lecture Notes in Computer Science, vol. 5584, pp. 244–257. Springer (2009)
15. Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT proof checking using a logical framework. Form. Methods Syst. Des. **42**(1), 91–118 (2013)

16. development team, T.C.: The coq proof assistant reference manual version 8.9 (2019), https://coq.inria.fr/distrib/current/refman/
17. Tseitin, G.S.: On the Complexity of Derivation in Propositional Calculus, pp. 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg (1983)
18. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing – SAT 2014. pp. 422–429. Springer International Publishing, Cham (2014)