Theory-Specific Proof Steps Witnessing Correctness of SMT Executions

Rodrigo Otoni*, Martin Blicha*†, Patrick Eugster*‡§, Antti E. J. Hyvärinen* and Natasha Sharygina*
* Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland
{rodrigo.benedito.otoni,martin.blicha,patrick.thomas.eugster,antti.hyvaerinen,natasha.sharygina}@usi.ch
† Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic
‡ Department of Computer Science, Technische Universität Darmstadt, Darmstadt, Germany
§ Department of Computer Science, Purdue University, West Lafayette, USA

***Abstract*—Ensuring hardware and software correctness increasingly relies on the use of symbolic logic solvers, in particular for satisfiability modulo theories (SMT). However, building efficient and correct SMT solvers is difficult: even state-of-the-art solvers disagree on instance satisfiability. This work presents a system for witnessing unsatisfiability of instances of NP problems, commonly appearing in verification, in a way that is natural to SMT solving. Our implementation of the system seems to often result in significantly smaller witnesses, lower solving overhead, and faster checking time in comparison to existing proof formats that can serve a similar purpose.**

***Index Terms*—proof of unsatisfiability, correctness certificate, model checking, satisfiability modulo theories**

I. Introduction

Automated reasoning engines for first-order logic are a core component of a vast range of different approaches for symbolic model checking [1]. The aim of these approaches is to leverage techniques from computational logic to efficiently traverse immense search spaces in order to determine whether hardware and software implementations conform to their specifications. Many model checkers ultimately reduce the verification problem to satisfiability queries in first-order logic expressed in a form suitable for satisfiability modulo theories (SMT) [2] solvers. The approach has recently gained significant traction. In addition to internal and less advertised uses in companies such as Microsoft, the Ethereum Foundation's SOLC smart contract compiler includes the capability of producing logic models suited for model checking [3].

As the approach becomes more common, the correctness of the reasoning algorithms becomes critical. Efficient reasoning algorithms are, however, non-trivial to implement and might themselves contain bugs. For example, in the 2020 edition of the SMT-COMP annual competition for SMT solvers (see smt-comp.github.io/2020), there were in total 149 instances with unknown status where at least two state-of-the-art solvers disagreed on the satisfiability. Some SMT solvers (e.g. [4]–[6]) offer the capability of producing proofs in a given proof system [7], that, among other uses, can partially respond to the need of increasing the trust in SMT solver. The idea is that the proofs can be replayed with an external proof checker (e.g. [8]–[12]), that may then accept or reject the proofs. In contrast to constructing proofs, we aim at making it simple for the interested parties to ensure the correctness of results[1]. The idea is to use the mathematical and logical foundations of the SMT algorithms to produce certificates that are simple enough to allow an auditor to write checkers for them with little effort, e.g., in a matter of hours or days.

Our emphasis is on quantifier-free fragments of first-order logic, and in particular showing correct the unsatisfiability of problems in NP, as this is more challenging than satisfiability in NP, assuming NP $\neq$ coNP. We further concentrate on the theories of linear real and integer arithmetics, and uninterpreted functions with equality, as these underly most other SMT theories. Our core contribution is in showing how to efficiently certify the executions of the theory-specific algorithms employed in SMT, and in joining them to produce checkable certificates starting from a directed acyclic graph (DAG) representation of the formula and ending, essentially, in an empty resolvent of a resolution refutation. The base of the certificates is the *deletion resolution asymmetric tautology (DRAT) format* [13], which we extend with real arithmetic certificates obtained based on Farkas' Lemma [14], natural formalization of equality of functions, lemmas for simple forms of Gomory cuts, and an axiomatization of the Tseitin transformation [15] and the De Morgan rules for converting logical circuits to conjunctive normal form (CNF).

We designed a draft format for the certificates, implemented the TSWC (Theory-Specific Witness Checker) tool for checking the correctness of proofs with respect to DAG representations of the corresponding input formulas, and instrumented the SMT solver OPENSMT [16] to produce the certificate format. Our experiments on the SMT-LIB benchmarks[2] show that the resulting system has a low overhead and that it performs well in the task of certifying unsatisfiability in comparison to the existing proof formats for SMT, in particular the logical framework with side conditions (LFSC) format [17] from the SMT solver CVC4 [18].

This work was partially supported by the SNSF (grant 200021_197353), by the ERC (grant FP7-617805), and by the GAČR (project 20-07487S).

978-1-6654-3274-0/21/$31.00 © 2021 IEEE

[1] We refrain from calling our certificates proofs, but acknowledge that the study of their polynomial simulation relations, worst-case sizes, or other aspects as a proof system, are interesting, but out of the scope of this paper.

[2] See http://smtlib.cs.uiowa.edu

II. State of the Art

We address the concerns on the correctness of SMT solvers previously identified for solvers for the propositional satisfiability problem (SAT) [19]. The annual SAT competition adopted a proof format for ensuring the correctness of unsatisfiability results in 2013 [20] and since 2014 adopted the DRAT format [13], based on a property of SAT solvers that inserted clauses result in contradiction as a result of a limited, polynomial-time computation called *unit propagation* [21]. The format is designed to express most current SAT solving techniques compactly and is a generalisation of the *deletion reverse unit propagation* (DRUP) [22] and the *resolution asymmetric tautology* (RAT) [23] formats. A variant, linear RAT (LRAT) [24], allows proof-checking in strictly linear time by decorating the DRAT format with indices serving as hints for unit propagation, with the price of an additional logging overhead during the search. While there is no widely accepted format for SMT proofs [7], we use a subset of the DRAT format for the underlying SAT reasoning.

The approach closest to ours is that of the SMT solver CVC4 [18], which produces proofs in the LFSC metalogic [17] that can be checked by the automatic LFSC checker. Similar to us, the system uses DRAT to validate the steps of the back-end SAT solver. However, the DRAT proofs need to be translated into LFSC, creating a bottleneck both for proof production and for proof checking [25]. CVC4's proofs can further be reconstructed in Coq [12]. The veriT SMT solver [26] is specifically designed for producing proofs and its proof format [4] is being developed alongside the solver itself. Unlike in our system, there is no independent checker for veriT proofs, but instead proofs can be checked using interactive theorem provers. A common workflow for SMT is to tune and replay a solver-specific proof in an interactive theorem prover such as Coq [8] or Isabelle/HOL [9]. The SMT solver Z3 [27] produces its own format [5], which can be reconstructed in Isabelle/HOL [11]. Currently the Z3 proof production capability seems to be experimental[3] and to the best of our knowledge, no independent checker exists.

III. Background

Our formulas F are in quantifier-free, multi-sorted first-order logic and contain logical operators, predicates, and *theory atoms*, i.e., (in)equalities over arithmetics and uninterpreted functions. We treat a formula as a directed, acyclic graph F_{DAG} where nodes are labeled with *symbols*, i.e., functions, predicates, constants, or logical operations; and outgoing edges are ordered. A symbol is a tuple $\langle p, s, \mathbf{c}\rangle$ where p is the *symbol name* (e.g. $+, \wedge$, or 0), s the *return sort*, and $\mathbf{c} = s_1 \dots s_n$ are the *argument sorts*, where n is the *arity* of p. Hence if a node is labeled with $\langle p, s, \mathbf{c}\rangle$, it has the ordered outgoing edges to the nodes c_i having the return sorts s_i for $1 \leq i \leq n$. There is a unique source node, and it must be labelled with a symbol with the Boolean return sort. To avoid exponential blowup, if two subtrees rooted at the nodes r and r' are equal (in the natural sense), then $r = r'$.

Given the formula F, an SMT solver attempts to determine whether F_{DAG} is satisfiable [2]. The formula F_{DAG} is first converted to an equisatisfiable CNF formula F_{CNF}, i.e., a conjunction of clauses over a set of propositional atoms and their negations, while maintaining the first-order theory interpretation of the atoms. The SMT solver determines the satisfiability of the formula in a search space traversal done by a *conflict-driven clause-learning* SAT solver [28] operating on F_{CNF}. During the search the SAT solver adds clauses to F_{CNF} while maintaining as invariant that if F is satisfiable, F_{CNF} is propositionally satisfiable, i.e., satisfiable when the theory interpretations of its atoms are ignored. Therefore, if at some point F_{CNF} becomes unsatisfiable propositionally, F must be unsatisfiable as well.

New clauses can be derived in two ways: either as *learned clauses*, through resolution on previous clauses, or as *theory clauses*, when a theory solver reports that a set of theory (in)equalities is unsatisfiable. One way to obtain a certificate for the correctness for an SMT execution determining unsatisfiability is thus to produce partial certificates connecting an input formula F_{DAG} to the step in a propositional proof system deriving *false*. This includes proving the derivation of the learned clauses, the theory clauses, and the transformation into F_{CNF}. In the next section we cover each step in detail, including how to connect the individual certificates.

IV. SMT Certification

The formula F_{DAG} is constructed by the *SMT parser* from an input *SMT2 file*, or directly by the *model checker*. The instrumentation added to an SMT solver produces the *certificates* and a *DAG file*, a serialization of F_{DAG}, that in the case of unsatisfiability can be checked by the *certificate checker* using different *checker modules*, and a *DAG checker* that can be used to ensure the correctness of the construction of F_{DAG} from F with a straight-forward traversal. Figure 1 shows the components of our system.

The core of our system is in SAT solving certificates, where clauses derived from both CNF conversion and theory solving are provided to the DRAT proof checker. It thus suffices to produce and check certificates for the clauses. In the following we discuss certificate production and checking for CNF conversion, theories for arithmetics for linear reals and integers, as well as uninterpreted functions with equality.

Input. The original formula F is transformed to its simplified version F_{DAG}. To validate the conversion, it suffices to traverse the representation of F and compare it to F_{DAG}.

CNF conversion. Certifying the CNF conversion consists of reproducing the standard way in which modern SMT solvers perform CNF conversion by applying Tseitin transformation [15] and the De Morgan rules. The certificate consists of a sequence of rule applications that map the nodes N_B of F_{DAG} having the Boolean return sort to clauses in F_{CNF}; and a bijection L between N_B and a set of literals. The checker verifies that L is a bijection, traverses F_{DAG} and,

[3] See the discussion in https://github.com/Z3Prover/z3/issues/4226

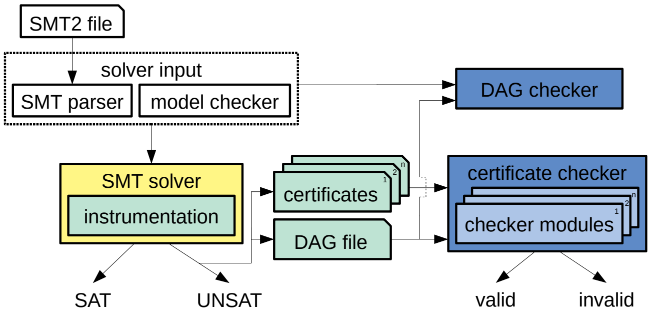


Fig. 1: Overview of our system. The SMT solver is shown yellow, with the artefacts needed for certificate production being in green, and those for checking being in blue.

at each node in N_B, applies the appropriate rule based on the node's symbol's name and verifies that the correct set of clauses appears in F_{CNF}. This is done by applying L^{-1} to each literal in the candidate clauses and checking that they match correctly the node and its children. Identifiers can be used to point to clauses in the certificate to avoid the worst-case quadratic blow-up in the size of the CNF formula in cases where many nodes produce a shared clause.

SAT solving. To validate the computations done by the back-end SAT solver we use the DRAT proof format, which is based on the concept of clause redundancy. Having a CNF formula as input, a DRAT proof sequentially adds and deletes clauses from the formula while preserving unsatisfiability, with the last addition of a valid proof being that of the empty clause, interpreted as $false$.

Linear real arithmetic solving. To validate the theory clauses in linear real arithmetic contained in the CNF formula we recreate the conflicts that originated them. To do so, the conflict's explanations, i.e., the conjunction of literals that led to the conflict, are logged in the linear real arithmetic certificate, with the literals representing inequalities derived from the original formula F. Using Farkas' lemma, a linear combination of the inequalities must lead to an inconsistency of form $1 \leq 0$ in a correct witness. The corresponding conjunction of inequalities is then negated and matched to the theory clause.

Linear integer arithmetic solving. The theory clauses in linear integer arithmetic can be validated by two different methods. The first method involves recreating the conflicts that happen in the real domain, using the same approach as done for real arithmetic, since an unsatisfiable result in the real domain implies the same result in the integer domain. The second method validates the tautological theory clauses of the form of integer bounds, given when a non-integer assignment is found, in order to restrict the search space. Since tautological clauses do not interfere with the satisfiability of the formula, we only check if they are well-formed bounds of the form $x \leq n \vee x \geq n+1$, where $n \in \mathbb{Z}$ and x is a variable in F_{DAG}.

Uninterpreted functions solving. The uninterpreted functions theory clauses have a specific form, which corresponds to a conflict of a single disequality $t_1 \neq t_2$ and a set of equalities P from which an equality $t_1 = t_2$ can be derived. Moreover, this equality can be derived using the basic properties of logical equality: *symmetry*, *transitivity* and *congruence*. The uninterpreted functions certificate records the applications of the transitivity and the congruence rule in the derivation of the equality $t_1 = t_2$. It is possible to record the whole derivation in the run of the *Explain* procedure of the state-of-the-art congruence-closure algorithm of [29] with minimal changes. To validate the certificate it is sufficient to (recursively) check that the equality $t_1 = t_2$ is *well-derived*, where an equality is well-derived if it or its symmetrical counterpart *i)* is an element of P, or *ii)* has been derived using congruence or transitivity from well-derived equalities.

V. Implementation

We implemented both the certificate production and the certificate checking parts of our approach, (the *instrumentation* and the *certificate checker* in Fig. 1). The instrumentation was done on the freely available, MIT licensed SMT solver OpenSMT [16]. OpenSMT won the quantifier-free linear real arithmetic (QF_LRA) track in SMT-COMP 2020, and is competitive in the instances from the logics quantifier-free linear integer arithmetic (QF_LIA) and quantifier-free uninterpreted functions (QF_UF). The instrumentation size (as reported by git-diff) is 1795 lines, excluding regression and unit tests. The certificate checking was implemented as the independent automatic checker TSWC; both our tools are available online[4].

Certificate production. We developed an instrumented version of OpenSMT, called OpenSMT-C, that accepts instances in the SMT-LIB2 standard and produces correctness certificates for each individual checker module, i.e., CNF conversion, SAT solving, and theory solving for QF_LRA, QF_LIA, and QF_UF, that can then be forwarded to a checker. A key point to avoid the introduction of a memory overhead due to certificate production is that all information relevant to the certificate is written to the disk immediately, instead of being stored in an internal data structure of the solver. In the current implementation we omit certain simple rewriting steps that OpenSMT performs while transforming F to F_{DAG}. These steps are thus currently left for the DAG checker. We took great care that all the simplifications that we do not validate can indeed be performed by the DAG checker in the sense that checking them does not require knowledge of the SMT solver's data structures.

Certificate checking. Our TSWC tool consists of nine independent components, one of them being the DRAT-trim proof checker [13] for certifying the results of the SAT solver, and the other eight being Python scripts with less than 300 lines of code each. We believe that the compact and modular checker design makes the code base easier to inspect both manually and, in the future, automatically. An overview of TSWC can be seen in Fig. 2. Starting with the CNF conversion, its checker receives the DAG, the CNF conversion

[4] Available at: http://verify.inf.usi.ch/certificate-producing-opensmt2

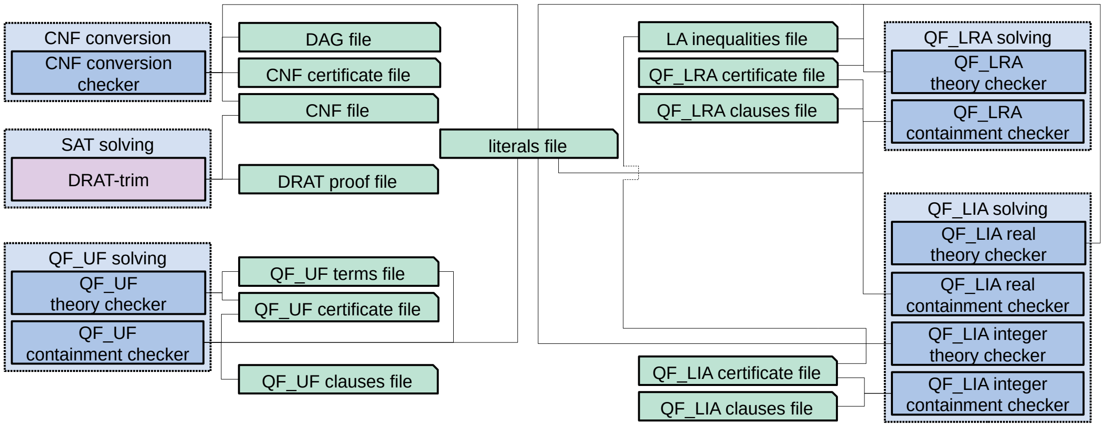


Fig. 2: TSWC architecture. The nine components are represented as solid rectangles, and are grouped by checker modules they belong to; the components in blue were developed by us, and the one in purple is off-the-shelf. All the files used by TSWC are also shown, in green, with the lines indicating which files are used by each component.

certificate, and the CNF formula used by DRAT-TRIM, and it applies the rules listed by the certificate in order to validate the clauses on the CNF formula that are derived from the DAG. The remaining clauses in the CNF formula are theory clauses, with the purpose of the checkers for each theory being to validate them. Each theory has both a theory checker, that validates the theory certificate, and a containment checker, that checks if all theory clauses added to the CNF formula are certified; for QF_LIA we use both the checkers for real and integer arithmetics, since it has solving procedures from both domains. With the clauses derived from both the CNF conversion and the theory solving validated, DRAT-TRIM can then ensure unsatisfiability at the SAT level. All auxiliary data is stored in the *literals*, *inequalities*, and *terms files*. The literals file contains a mapping between literals and nodes of F_{DAG}; the inequalities file contains the linear arithmetic literals, and the terms file contains the uninterpreted functions and their arguments.

VI. EVALUATION

In our evaluation we used the non-incremental benchmark sets of each theory supported by OPENSMT-C available in the SMT-LIB benchmark repository[2]. For certificate production we compared our implementation against three proof producing solvers, namely CVC4 1.8, VERIT 09a24ff-rmx, and Z3 4.8.9; from now on we refer to *witnesses* as either certificates or proofs, depending on the tool being referenced. We measure the number of witnesses produced and their sizes, as well as the witness production time and overhead, in terms of runtime and memory use. For witness checking we compared TSWC against the LFSC checker, which can automatically check CVC4's proofs, and is the only tool comparable to TSWC, in terms of runtime and memory use. Our experiments were done with a 60 seconds timeout and a 10 gigabytes memory limit; all results are available online[5].

Witness production. We ran all solvers in both standard and witness producing modes. Our results are compiled in Table I. For the number of witnesses produced, OPENSMT-C had the best result for QF_LRA and QF_UF, also having the smallest overhead in number of witnesses, while Z3 had by far the best results for QF_LIA. Regarding runtime, OPENSMT-C and VERIT had the best performances for QF_LRA and QF_UF, respectively, while for QF_LIA CVC4 had the shortest runtime in witness producing mode. CVC4's average runtime for QF_LIA has, however, to be taken with a grain of salt, since its apparent increase in performance when witness production is enabled may be due to the sharp increase in the number of timeouts, which are not part of the computed average. When comparing memory use, OPENSMT-C had the best results in witness producing mode for QF_LRA and QF_UF, while VERIT had the smallest use for QF_LIA. In terms of witnesses sizes, OPENSMT-C had the best results for QF_LIA and QF_UF, while also having a competitive result for QF_LRA, with an average witness size close to that of Z3, which had the best result. One important point to make about the Z3 proofs is that they are known to be unstable[3], and that it has been over a decade since the last documented update on their format [5], [11], unlike CVC4's [6], [17], [25] and VERIT's [4], [10]. A runtime, memory use, and witness size comparison of the two solvers with most witnesses produced, for each theory, can be seen in Fig. 3; all pair-wise comparisons are available online[5].

Certificate checking. For all proofs produced by OPENSMT-C and CVC4 we ran their respective checkers, with our results being compiled in Table II; no witness was rejected by either tool, nor did we register any memouts. We can see that TSWC was able to verify more witnesses for every theory. For runtime and memory use, LFSC had better results for two of the three theories, but this can be mainly attributed to the high number of errors it had, which are not part of the computed average.

[5] Available at: https://scm.ti-edu.ch/repogit/verify-witness-evaluation.git

TABLE I: Witness production comparison. We report, for each theory, the number of unsatisfiable instances, timeouts, memouts, and errors, given by the solvers, as well as their average runtime, in seconds, and memory use and witness size, in kilobytes; an error refers to an exception being thrown during execution. Each cell contains the result for standard mode on the left and for witness producing mode on the right, with the exception of those for % change, which report the variation on the number of unsatisfiable instances given, and for average witness size, whose results are only relevant in witness producing mode.

		UNSAT	% change	Timeout	Memout	Error	Avg. runtime	Avg. mem. use	Avg. witness size
QF_LRA (1648 instances)	OpenSMT-C	**636/633**	**99.5%**	**151/156**	0/0	0/0	**3.91/4.31**	30532/**26603**	3956
	CVC4	583/570	97.7%	274/289	0/0	0/0	5.73/6.61	35877/76953	13684
	veriT	580/561	96.7%	249/268	0/0	10/10	4.12/5.22	**19227**/44358	69438
	Z3	570/558	97.8%	253/264	0/0	0/0	5.17/5.30	36472/76929	**3546**
QF_LIA (6947 instances)	OpenSMT-C	2023/1519	75.0%	3341/3892	0/0	0/0	10.07/9.99	113622/51443	**13018**
	CVC4	1398/928	66.3%	3147/3620	0/0	0/0	6.30/**1.30**	63127/37728	18883
	veriT	1002/953	95.1%	4198/4247	0/0	1/1	**1.82**/4.05	**12028/20277**	165914
	Z3	**2238/2340**	**104.5%**	**1786/1685**	1/1	0/0	5.17/6.62	58325/276318	24876
QF_UF (7457 instances)	OpenSMT-C	4326/**4311**	**99.6%**	24/**39**	0/0	0/0	0.95/1.14	9416/**9563**	**6730**
	CVC4	4321/4220	97.6%	34/135	0/0	0/0	0.39/1.83	17517/28316	6854
	veriT	**4347**/4176	96.0%	**3**/166	0/0	0/8	**0.10/0.79**	**5931**/23940	20102
	Z3	4341/4236	97.5%	9/114	0/0	0/0	0.22/1.26	15474/49355	12601

TABLE II: Witness checking comparison. We report, for each theory, the number of verified witnesses, timeouts, and errors, given by the checkers, as well as their average runtime, in seconds, and memory use, in kilobytes; an error refers to an exception being thrown during execution. The instances used are those for which both OpenSMT-C and CVC4 produced a witness.

		Verified	Timeout	Error	Avg. runtime	Avg. mem. use
QF_LRA (567 instances)	TSWC	**564**	**3**	0	3.15	**67372**
	LFSC	471	8	88	**2.85**	102861
QF_LIA (913 instances)	TSWC	**903**	10	0	1.37	65039
	LFSC	128	**0**	785	**0.12**	**26609**
QF_UF (4218 instances)	TSWC	**4217**	**1**	0	**1.44**	69507
	LFSC	4157	50	11	4.01	**28454**

VII. Conclusion

SMT solvers are being increasingly used in studying correctness of safety-critical software, making the correctness of the solvers themselves critical. A central part of SMT solvers' work is in answering satisfiability queries of instances of problems in NP. While the correctness of a satisfiable result can be checked in time polynomial in the query size, certificates for unsatisfiability are believed to be much harder.

We showed in this paper that when a SMT solver claims that a query on an instance is unsatisfiable, the correctness of that claim can be checked using the data structures that the core solving algorithms of SMT are already maintaining, by connecting them to their underlying mathematical and logical foundations. In addition, it is easy to write checkers for these certificates, and producing them has a comparable or smaller overhead when contrasted with more traditional proofs.

We identify several directions for further work. First, we believe that the community would benefit from a common format for the certificates. Second, we would like to extend the approach to SMT theory combinations and more applied theories such as arrays and data structures. Third, the produced certificates could be integrated into tools for more complex logics such as those with quantifiers, or systems of Horn clauses. Finally, we believe that connecting the practical certificates to a formal proof system will provide interesting insight in the complexity of the SMT algorithms.

References

[1] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach," in *Principles of Programming Languages – POPL 1983*, 1983.

[2] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, *Satisfiability modulo theories*, 2009.

[3] M. Marescotti, R. Otoni, L. Alt, P. Eugster, A. E. J. Hyvärinen, and N. Sharygina, "Accurate smart contract verification through direct modelling," in *International Symposium on Leveraging Applications of Formal Methods – ISoLA 2020*, 2020.

[4] F. Besson, P. Fontaine, and L. Thry, "A flexible proof format for SMT: a proposal," First Workshop on Proof eXchange for Theorem Proving – PxTP 2011, 2011.

[5] L. de Moura and N. Bjrner, "Proofs and refutations, and Z3," *CEUR Workshop*, vol. 418, 2008.

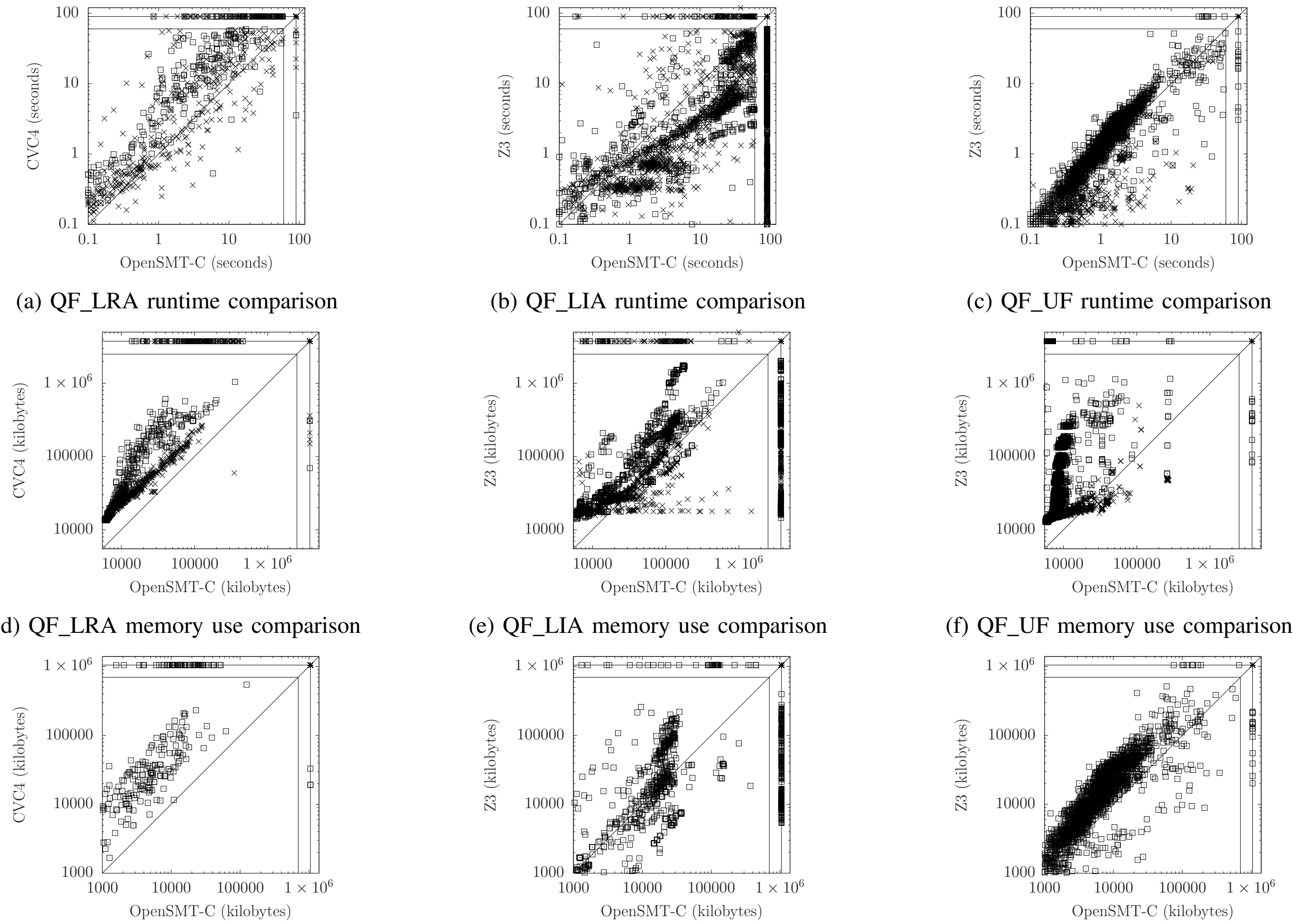
[6] G. Katz, C. Barrett, C. Tinelli, A. Reynolds, and L. Hadarean, "Lazy proofs for DPLL(T)-based SMT solvers," in *Formal Methods in Computer-Aided Design – FMCAD 2016*, 2016.

[7] C. Barrett, L. de Moura, and P. Fontaine, "Proofs in satisfiability modulo theories," 2014, available at: theory.stanford.edu/ barrett/pubs/BdMF15.pdf.

[8] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner, "A modular integration of SAT/SMT solvers to Coq through proof witnesses," in *Certified Programs and Proofs – CPP 2011*, 2011.

[9] J. C. Blanchette, S. Bhme, M. Fleury, S. J. Smolka, and A. Steckermeier, "Semi-intelligible Isar proofs from machine-generated proofs," *Journal of Automated Reasoning*, vol. 56, no. 2, 2016.

[10] H. Barbosa, J. C. Blanchette, and P. Fontaine, "Scalable fine-grained proofs for formula processing," in *Automated Deduction – CADE 26*, 2017.



(a) QF_LRA runtime comparison

(b) QF_LIA runtime comparison

(c) QF_UF runtime comparison

(d) QF_LRA memory use comparison

(e) QF_LIA memory use comparison

(f) QF_UF memory use comparison

(g) QF_LRA witness size comparison

(h) QF_LIA witness size comparison

(i) QF_UF witness size comparison

Fig. 3: Comparison between the two solvers with most witnesses produced, for each theory, both in witness producing mode; squares and crosses stand for UNSAT and SAT results, the top and right lines represent value limit, timeout, and memout.

[11] S. Böhme and T. Weber, "Fast LCF-style proof reconstruction for Z3," in *Interactive Theorem Proving – ITP 2010*, 2010.

[12] B. Ekici, A. Mebsout, C. Tinelli, C. Keller, G. Katz, A. Reynolds, and C. Barrett, "SMTCoq: A plug-in for integrating SMT solvers into Coq," in *Computer Aided Verification – CAV 2017*, 2017.

[13] N. Wetzler, M. J. H. Heule, and W. A. Hunt, "DRAT-trim: Efficient checking and trimming using expressive clausal proofs," in *Theory and Applications of Satisfiability Testing – SAT 2014*, 2014.

[14] J. Farkas, "Theorie der einfachen ungleichungen." *Journal fr die reine und angewandte Mathematik*, vol. 1902, no. 124, 1902.

[15] G. S. Tseitin, *On the Complexity of Derivation in Propositional Calculus*, 1983.

[16] A. E. J. Hyvärinen, M. Marescotti, L. Alt, and N. Sharygina, "OpenSMT2: An SMT solver for multi-core and cloud computing," in *Theory and Applications of Satisfiability Testing – SAT 2016*, 2016.

[17] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli, "SMT proof checking using a logical framework," *Formal Methods in System Design*, vol. 42, no. 1, 2013.

[18] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification – CAV 2011*, 2011.

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

[20] A. Balint, A. Belov, M. J. H. Heule, and M. Järvisalo, "Proceedings of SAT competition 2013: Solver and benchmark descriptions," Tech. Rep. B-2013-1, 2013.

[21] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM*, vol. 5, pp. 201 – 215, 1960.

[22] M. J. H. Heule, W. A. Hunt, and N. Wetzler, "Trimming while checking clausal proofs," in *Formal Methods in Computer-Aided Design – FMCAD 2013*, 2013.

[23] M. J. H. Heule, W. A. Hunt, and N. Wetzler, "Verifying refutations with extended resolution," in *Automated Deduction – CADE 24*, 2013.

[24] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt, M. Kaufmann, and P. Schneider-Kamp, "Efficient certified RAT verification," in *Automated Deduction – CADE 26*, 2017.

[25] A. Ozdemir, A. Niemetz, M. Preiner, Y. Zohar, and C. Barrett, "DRAT-based bit-vector proofs in CVC4," in *Theory and Applications of Satisfiability Testing – SAT 2019*, 2019.

[26] T. Bouton, D. Caminha B. de Oliveira, D. Déharbe, and P. Fontaine, "veriT: An open, trustable and efficient SMT-solver," in *Automated Deduction – CADE 22*, 2009.

[27] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2008*, 2008.

[28] J. P. M. Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, 1999.

[29] R. Nieuwenhuis and A. Oliveras, "Proof-producing congruence closure," in *Rewriting Techniques and Applications – RTA 2005*, 2005.