Retrieving Unknown SMT Formulas via Structural Mutations

Shuo Ding[0000−0003−0843−0729] and Qirun Zhang[0000−0001−5367−9377]

Georgia Institute of Technology, Atlanta GA 30332, USA
{sding,qrzhang}@gatech.edu

Abstract. Satisfiability Modulo Theories (SMT) solvers are fundamental tools for program analysis and verification. The satisfiability problem for first-order logic is undecidable. In practice, SMT solvers typically employ various heuristics and are inherently *incomplete*. Solvers return unknown if they cannot solve a particular formula. The unknown results drastically hinder the usability of SMT solvers and directly affect client applications. The standard way to reduce unknown cases is to develop more powerful solvers, which requires significant algorithmic and engineering efforts.

This work-in-progress paper discusses a new perspective on improving SMT solving: instead of developing more powerful solvers for all formulas, we focus on mutating "hard" formulas (unknown formulas) to make them "easier" to solve. That gives us enormous flexibility to process unknown formulas without affecting normal formulas. Specifically, given an unknown formula and a solver, we propose to repeatedly modify the formula via structural mutations. Our key insights are (1) structural mutations make formulas smaller so that they are presumably easier to reason about, and (2) structural mutations approximate formulas so that we can reason about the original formulas indirectly. Then, we utilize the same solver to solve the mutated formulas to retrieve the sat/unsat results of the original unknown formulas.

1 Introduction

Satisfiability Modulo Theories (SMT) is a powerful formulation that can express many problems arising in symbolic execution [12,30], formal verification [7,23], program synthesis [19], etc. An SMT problem instance describes a first-order logic formula with respect to certain background theories. SMT solvers are software tools for deciding the satisfiability of SMT formulas. Z3 [25] and CVC4 [5] (now succeeded by CVC5 [3]) are two widely used SMT solvers. However, it is well-known that the satisfiability problem for first-order logic is undecidable. In addition to theoretical restrictions, modern SMT solvers also face practical issues, including incomplete implementations and resource limits. Therefore, practical SMT solvers return unknown results for formulas that they cannot solve. In the popular Satisfiability Modulo Theories Competition (SMT-COMP), in many tracks, a solver receives a zero "correctly solved score" if the `check-sat` command returns `unknown` [4]. In practice, SMT solvers strive to offer best-effort answers by solving as many formulas as possible.

The standard way to reduce `unknown` cases is by improving solvers, including developing new algorithms and engineering better solvers. That is challenging and time-consuming due to both the theoretical hardness of SMT solving and the implementation issues of such complex systems. For example, CVC4's bitvector rewriting rules contain more than 3.5K source lines of code [32]. From users' perspective, it is possible to try solvers' available options or tactics, or even different solvers to handle `unknown` cases, but there are usually a limited number of choices at a given time.

We consider a source-level approach for improving SMT solving. Rather than developing more powerful solvers for all formulas, we focus on "hard" formulas for which solvers return `unknown`. Working directly on `unknown` formulas enables unique opportunities for employing solver-agnostic source-to-source transformations to make "hard" formulas easier to solve. Specifically, given an `unknown` formula ϕ for a specific solver, we propose a technique called *structural mutations* to perform lightweight rewriting on ϕ and obtain a mutated formula ϕ'. Then we apply the same solver on ϕ' to reason about ϕ indirectly. There are two key observations that underlie structural mutations:

– *Small formulas are easier to solve.* In general, smaller cases have simpler structures and are presumably easier to reason about. A similar observation exists in compiler testing, where developers strongly encourage submitting small, reproducible test programs because it is easier to manually inspect small test cases [36]. Indeed, well-known production compilers such as GCC and LLVM always advocate test reduction [31] in bug reporting processes [17,24]. Following the same observation, our structural mutations produce smaller formulas that are generally "simpler" to solve.
– *Approximations enable indirect reasoning of formulas.* Approximations, which are used in many SMT solving techniques [8,11,20], enables indirect reasoning of formulas. By mutating the original formula, our technique can either over- or under-approximate the original `unknown` formula. For example, we can perform a structural mutation by deleting a top-level conjunct, which relaxes the original `unknown` formula and provides an over-approximation. If the over-approximated formula is `unsat`, the origin formula must be `unsat`. Figure 1 describes the rationale for retrieving `unknown` formulas via over- and under-approximations.

2 Motivating Examples

This section gives two motivating examples of reasoning about `unknown` formulas via structural mutations.

Figure 2 gives a formula [2] in the LIA (Linear Integer Arithmetic) category of the `SMT-LIB` benchmarks. Z3 [1] reports `unknown` on the original formula in Figure 2

[1] We use commit `11477f1` (December 16, 2020) for Z3.

$$\begin{tikzcd}
\phi_{\mathsf{under}} & \phi & \phi_{\mathsf{over}} \\
\mathsf{sat} \arrow[r, "\text{implies}"] & \mathsf{sat} & \mathsf{sat} \\
\mathsf{unsat} & \mathsf{unsat} & \mathsf{unsat} \arrow[l, "\text{implies}"']
\end{tikzcd}$$

Fig. 1: Satisfiability relations between the original formula ϕ, the under-approximated version ϕ_{under}, and the over-approximated version ϕ_{over}. Arrows in the figure represent implication relations.

due to incomplete quantifiers. We mutate the original formula by deleting the third assertion (lines 15-20, inclusive). With fewer assertions, we clearly have obtained a "relaxed" version of the original formula. Z3 can successfully report `unsat` on our over-approximated formula. Therefore, we can conclude that the original formula is unsatisfiable because even the over-approximated formula is unsatisfiable (the "$\leftarrow$" direction in Figure 1).

Figure 3 gives a formula [1] in the AUFLIA (Arrays, Uninterpreted Functions, and Linear Integer Arithmetic) category of the popular `SMT-LIB` benchmarks. CVC4 [2] reports `unknown` on the original formula in Figure 3a because the solver is incomplete in this case. We mutate the original formula by instantiating the free variable `n` (line 3) to the constant `0`. Clearly, this is an under-approximation because it restricts the value of `n`. CVC4 can successfully solve the under-approximated formula and return `sat`. Because the under-approximated version is satisfiable, it implies that the original formula is satisfiable (the "$\rightarrow$" direction in Figure 1). Moreover, CVC4 can generate a model in Figure 3b for the under-approximated formula. The model assigns `false` and 0 to the uninterpreted functions `f` and `v`, respectively. It is straightforward that appending `n = 0` to the model gives us a model of the original formula, because if we evaluate the formula on this model, the assertion becomes "there does not exist an `x` such that $1 \leq x \leq 0$ and ...", which is clearly true.

3 Structural Mutations

SMT formulas are first-order logic formulas with respect to different background theories. A theory over a signature Σ could be defined as a set I of interpretations for Σ, and I is also called the models of T. Under a background theory T, we use $\phi(\vec{x})$ to represent a SMT formula with free variables $\vec{x}$ as a vector. $\phi(\vec{x})$ is satisfiable if and only if there exists a model of T in which $\phi(\vec{x})$ evaluates to `true`. Otherwise, the formula is unsatisfiable. In practice, a model M of $\phi(\vec{x})$ usually refers to a function that maps each free variable in $\vec{x}$ to a value of the corresponding sort, such that $\phi(\vec{x})$ evaluates to `true` under this assignment and the corresponding theory. We adopt this function-mapping view of models in

[2] We use commit `80e0246` (December 16, 2020) for CVC4.

```
 1 (set-logic LIA)
 2 (declare-fun ~a29~0 () Int)
 3 (assert (not (exists ((v_prenex_81 Int))
 4     (let ((.cse0 (* 4 (div v_prenex_81 5))))
 5         (and (<= 0 (+ .cse0 4)) (<= 0 .cse0)
 6              (<= ~a29~0 (+ (mod .cse0 299978) 300021))
 7              (= 0 (mod v_prenex_81 5)))))))
 8 (assert (not (exists ((v_~a29~0_1039 Int))
 9     (let ((.cse1 (* 4 (div v_~a29~0_1039 5))))
10         (let ((.cse0 (mod .cse1 299978)))
11             (and (= 0 (mod v_~a29~0_1039 5))
12                  (not (= 0 .cse0)) (< .cse1 0)
13                  (<= ~a29~0 (+ .cse0 43))
14                  (= (mod (+ .cse1 4) 299978) 0)))))))
15 (assert (not (exists ((v_prenex_81 Int))
16     (let ((.cse2 (* 4 (div v_prenex_81 5))))
17         (let ((.cse1 (+ .cse2 4))(.cse0 (mod .cse2 299978)))
18             (and (<= ~a29~0 (+ .cse0 300021))
19                  (< .cse1 0) (not (= (mod .cse1 299978) 0))
20                  (= 0 .cse0) (= 0 (mod v_prenex_81 5))))))))
21 (assert (exists ((v_prenex_81 Int))
22     (let ((.cse0 (* 4 (div v_prenex_81 5))))
23         (and (<= 0 (+ .cse0 4)) (<= 0 v_prenex_81) (<= 0 .cse0)
24              (<= ~a29~0 (+ (mod .cse0 299978) 300021))))))
25 (check-sat)
26 (exit)
```

Fig. 2: An over-approximation example for Z3, where the over-approximation is realized by removing the third assertion (line 15-20, inclusive).

later sections. Moreover, we assume a fixed background theory T over a signature Σ. Let F_Σ be the set of formulas over Σ.

Definition 1 (Structural Mutations). *A* structural mutation $\mathcal{M}$ *is a function from* F_Σ *to* F_Σ *such that for each* $\phi \in F_\Sigma$, $\mathcal{M}(\phi)$ *could be obtained by replacing* n *(*$n > 0$ *and* n *may depend on* ϕ*) non-overlapping subterms* $f_1, f_2, ..., f_n$ *in* ϕ *with* n *new terms* $g_1, g_2, ..., g_n$ *simultaneously, where for each* $i \in \{1, 2, ..., n\}$, f_i *and* g_i *are of the same sort.*

The essence of structural mutations is approximating unknown formulas. The usefulness of retrieved satisfiability results is strongly correlated to the approximation directions. Based on Figure 1, if solvers return sat for over-approximated formulas ϕ_{over}, the result is uninformative. Similarly, the unsat result from under-approximated formulas ϕ_{under} is uninformative as well. Straightforward and unguided approximations can easily lead to uninformative results. In the ideal case, approximations achieved by structural mutations need to be *effective* (i.e., they could make unknown formulas solvable) and *admissible* (i.e., they should not lead to uninformative results).

Unfortunately, there is a tension between effectiveness and admissibleness, and finding a sweet spot of approximations is challenging. To tackle the challenge, we devise *fine-grained* mutations to strike a balance between these two competing needs. Specifically, by repeatedly applying small structural mutations to the original formula, we get a directed acyclic graph (DAG) of mutated formulas whose nodes are formulas and edges are approximation steps. The graph

```
1 (set-logic AUFLIA)
2 (declare-fun f (Int Int) Bool)
3 (declare-fun n () Int)
4 (declare-fun v () Int)
5 (assert (! (not (exists ((x Int))
6     (and (<= 1 x) (<= x n) (f x v))))
7                         :named goal))
8 (check-sat)
9 (exit)
```

(a) Original formula.

```
(
 (define-fun f
  ((BOUND_VARIABLE_327 Int)
    (BOUND_VARIABLE_328 Int))
    Bool false)
 (define-fun v () Int 0)
)
```

(b) A model for our under-approximated formula.

Fig. 3: An under-approximation example for CVC4, where the under-approximation is realized by instantiating the free variable `n` (line 3) to `0`.

is acyclic because our mutations strictly reduce formulas. Then, based on the satisfiability results of running solvers on mutated formulas, we can perform a backtracking search on this DAG to refine the approximated formulas ϕ'. Consequently, the feedback-based iteration guides structural mutations toward the useful directions (depicted as "$\xrightarrow{implies}$" and "$\xleftarrow{implies}$") in Figure 1. Our fine-grained mutation process resembles abstraction refinements. However, common abstraction refinement techniques for SMT solvers (e.g. the mixed abstraction technique [8]) are not directly applicable because they (1) do not explicitly handle the unknown cases and (2) finally, always resort to the most precise abstraction (the original formula) but in our case, the original formula is unknown.

We propose four concrete structural mutations. The mutations are both reducers and approximations (i.e., they can both reduce and approximate the original formulas). Moreover, they are all theory-independent, meaning that they could be applied to all background theories. In our mutations, a top-level disjunct/conjunct denotes a disjunct/conjunct whose corresponding disjunction/-conjunction is at the root of the formula's abstract syntax tree (AST). For example, in $P \vee Q$, P is a top-level disjunct. A non-trivial disjunct/conjunct is a disjunct/conjunct that is not the literal false/true. A non-trivial subterm is a subterm that is not a single free variable.

– *Removing Top-Level Disjuncts ($\mathcal{U}_\vee$):* Replacing the first top-level non-trivial disjunct (if it exists) with false is a structural mutation. It is a reducer with respect to the number of top-level non-trivial disjuncts. It is also a domain-preserving under-approximation. Note that changing $P \vee Q$ to $\mathsf{false} \vee Q$ could still be regarded as domain-preserving, because $\mathsf{false} \vee Q$ could be regarded as a formula with free variables $\{P, Q\}$ while P is not used.
– *Instantiating Free Variables ($\mathcal{U}_{\mathsf{in}}$):* Replacing all occurrences of the first occurred free variable (if it exists) with one value in its sort is a structural mutation. It is a reducer with respect to the number of free variables. It is also a domain-adjusting under-approximation.
– *Removing Top-Level Conjuncts ($\mathcal{O}_\wedge$):* Replacing the first top-level non-trivial conjunct (if it exists) with true is a structural mutation. It is a reducer with

respect to the number of top-level non-trivial conjuncts. It is also a domain-preserving over-approximation.
- *Abstracting Subterms ($\mathcal{O}_{\mathsf{term}}$):* Replacing the first non-trivial subterm that does not contain variables bound by quantifiers (if it exists) with a new free variable of the same sort is a structural mutation. It is a reducer with respect to the number of non-trivial subterms. It is also a domain-adjusting over-approximation.

Note that some mutation steps could be interpreted as several different approximations. For example, replacing P in $P \wedge Q$ with true could be regarded as a domain-adjusting under-approximation $\mathcal{U}_{\mathsf{in}}$ or a domain-preserving over-approximation $\mathcal{O}_{\wedge}$. The actual effect is preserving the satisfiability because both the original formula $P \wedge Q$ and the modified formula $\mathsf{true} \wedge Q$ are satisfiable.

The definitions in Section 3 impose constraints such as "replacing *the first* top-level disjunct" when there are multiple top-level disjuncts, so each application of mutation produces only one transformed formula. It is possible to remove those constraints and get multiple mutated formulas in each step. Therefore, we can get more mutated formulas to use in practice. If we regard the mutated formulas as nodes and approximation steps as directed edges, we form a directed acyclic graph (DAG). It is a DAG because there is no cycle due to the reducer property. We call the graph "under-approximation DAG" or "over-approximation DAG".

Our $\mathsf{unknown}$ formula retrieval algorithms repeatedly apply and revert mutations on the original formula. Thus, it forms a process of *adjusting* the approximations (making more or fewer approximations) along the corresponding under- or over-approximation DAG. Recall that approximations can be uninformative (e.g. over-approximating a formula ϕ to a sat formula ϕ' is uninformative since it provides no information about ϕ). To avoid encountering too many uninformative cases, our structural mutation framework prunes the adjusting process based on over- or under-approximation DAG: if we over-approximate the formula ϕ to a satisfiable formula, we don't need to continue the current branch of over-approximation since further over-approximations can only produce uninformative approximations. Similarly, if we under-approximate the formula ϕ to an unsatisfiable formula, we can stop the current branch of under-approximation.

4 Related Work

Formula simplification techniques have been developed to simplify formulas for SMT solvers [14,33,34]. These techniques, however, produce equivalent or equisatisfiable formulas, and thus often need to do sophisticated reasoning about Boolean logic and underlying theories. Our structural mutations relax the requirement from equivalence or equisatisfiability to approximations, and thus produce transformations that are easier to reason about.

Approximations have also been widely used to solve SMT formulas. The DPLL(T) framework [15,28,29], which forms the basis of many modern SMT solvers, leverages the Boolean abstraction of the original formula and then refines the abstraction using information provided by theory-specific solvers. De Moura and Rueß [27] have proposed lemmas on demand, which is also an abstraction refinement process. Approximations can also be done in the theory/first-order layer [10,26]. Bauer et al. have proposed a technique that can ignore parts of the Boolean abstraction that do not affect the overall truth value [6]. Explicit approximations have been introduced to SMT solvers to model bit-vector operations [20,37] and bit-vector values [9]. SMT solvers can also alternate between over-approximations and under-approximations [11,21,22], as well as mixing them altogether [8]. Approximations also help to simplify formulas [33], to change the decidability of certain formulas [16], etc. In the refinement aspect, techniques similar to counter-example guided abstraction refinement [13] are well-developed in SMT solvers. Approximating formulas can also happen outside solvers. For example, concolic testing [18,35] simplifies formulas by instantiating variables before using solvers to solve them. Compared with those existing techniques, our structural mutations are solver/theory-independent, are not part of any solver or automated reasoning tools, and can be applied to almost all types of formulas.

5 Conclusion

This paper has discussed a source-level approach to improve SMT solving: instead of improving solvers for all possible input formulas, we focus on mutating (approximating) formulas that are already `unknown` to solvers. As the next step, we plan to conduct an extensive study to validate the idea on real-world SMT constraints.

Acknowledgements. We thank the anonymous reviewers for their feedback. This work was supported, in part, by the United States National Science Foundation (NSF) under grants No. 2114627 and No. 2237440; and by the Defense Advanced Research Projects Agency (DARPA) under grant N66001-21-C-4024. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above sponsoring entities.

References

1. A Test Case of AUFLIA (Arrays, Uninterpreted Functions, andLinear Integer Arithmetic) Logic. `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/AUFLIA/-/blob/master/20170829-Rodin/smt4391808662368180273.smt2`, accessed: 2021-01
2. A Test Case of LIA (Linear Integer Arithmetic) Logic. `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/LIA/-/blob/master/20190429-UltimateAutomizerSvcomp2019/Problem15_label00_false-unreach-call.c_5.smt2`, accessed: 2021-01
3. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
4. Barbosa, H., Hoenicke, J., Hyvarinen, A.: 15th International Satisfiability Modulo Theories Competition (SMT-COMP 2020): Rules and Procedures. `https://smt-comp.github.io/2020/rules20.pdf`, accessed: 2021-02
5. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011). pp. 171–177 (2011)
6. Bauer, A., Leucker, M., Schallhart, C., Tautschnig, M.: Don't care in SMT: building flexible yet efficient abstraction/refinement solvers. Int. J. Softw. Tools Technol. Transf. **12**(1), 23–37 (2010)
7. Beyer, D., Dangl, M., Wendler, P.: A unifying view on smt-based software verification. J. Autom. Reason. **60**(3), 299–335 (2018)
8. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2009). pp. 69–76 (2009)
9. Brummayer, R., Biere, A.: Effective bit-width and under-approximation. In: Proceedings of the 12th International Conference on Computer Aided Systems Theory (EUROCAST 2009). pp. 304–311 (2009)
10. Brummayer, R., Biere, A.: Lemmas on demand for the extensional theory of arrays. J. Satisf. Boolean Model. Comput. **6**(1-3), 165–201 (2009)
11. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.A.: Deciding bit-vector arithmetic with abstraction. In: Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007). pp. 358–372 (2007)
12. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008). pp. 209–224 (2008)
13. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000). pp. 154–169 (2000)

14. Dillig, I., Dillig, T., Aiken, A.: Small formulas for large programs: On-line constraint simplification in scalable static analysis. In: Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6337, pp. 236–252. Springer (2010)
15. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): fast decision procedures. In: Proceedings of the 16th International Conference Computer Aided Verification (CAV 2004). pp. 175–188 (2004)
16. Gao, S., Avigad, J., Clarke, E.M.: Delta-decidability over the reals. In: Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science (LICS 2012). pp. 305–314 (2012)
17. GCC: A Guide to Testcase Reduction. `https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction`, accessed: 2021-01
18. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005. pp. 213–223. ACM (2005)
19. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010). pp. 215–224 (2010)
20. Jonás, M., Strejcek, J.: Abstraction of bit-vector operations for bdd-based SMT solvers. In: Proceedings of the 15th International Colloquium on Theoretical Aspects of Computing (ICTAC 2018). pp. 273–291 (2018)
21. Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O.: Abstraction-based satisfiability solving of presburger arithmetic. In: Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004). pp. 308–320 (2004)
22. Lahiri, S.K., Mehra, K.K.: Interpolant based decision procedure for quantifier-free presburger arithmetic. J. Satisf. Boolean Model. Comput. **1**(3-4), 187–207 (2007)
23. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16). pp. 348–370 (2010)
24. LLVM: How to submit an LLVM bug report. `https://llvm.org/docs/HowToSubmitABug.html`, accessed: 2021-01
25. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). pp. 337–340 (2008)
26. de Moura, L., Jovanovic, D.: A model-constructing satisfiability calculus. In: Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2013). pp. 1–12 (2013)
27. de Moura, L., Rueß, H.: Lemmas on demand for satisfiability solvers. In: Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT 2002). pp. 244–251 (2002)
28. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with exhaustive theory propagation and its application to difference logic. In: Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005). pp. 321–334 (2005)
29. 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 (2006)
30. Palikareva, H., Cadar, C.: Multi-solver support in symbolic execution. In: Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013). pp. 53–68 (2013)

31. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012). pp. 335–346 (2012)
32. Reynolds, A., Barbosa, H., Tinelli, C., Niemetz, A., Noetzli, A., Preiner, M., Barrett, C.: Rewrites for SMT Solvers Using Syntax-Guided Enumeration. `http://homepage.divms.uiowa.edu/~ajreynol/pres-smt2018.pdf`, accessed: 2021-02
33. Reynolds, A., Nötzli, A., Barrett, C., Tinelli, C.: High-level abstractions for simplifying extended string constraints in SMT. In: Proceedings of the 31st International Conference on Computer Aided Verification (CAV 2019). pp. 23–42 (2019)
34. Reynolds, A., Woo, M., Barrett, C.W., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10427, pp. 453–474. Springer (2017)
35. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005. pp. 263–272. ACM (2005)
36. Sun, C., Li, Y., Zhang, Q., Gu, T., Su, Z.: Perses: syntax-guided program reduction. In: Proceedings of the 40th International Conference on Software Engineering (ICSE 2018). pp. 361–371 (2018)
37. Teuber, S., Büning, M.K., Sinz, C.: An incremental abstraction scheme for solving hard smt-instances over bit-vectors. CoRR **abs/2008.10061** (2020), `https://arxiv.org/abs/2008.10061`