Value Mutation Testing for SMT Solvers

DYLAN WOLFF*, ETH Zürich, Switzerland

We propose *Value Mutation Testing*, a novel approach for testing SMT solvers. The core idea behind this technique is to generate test cases by mutating the constant values within a given seed SMT formula. We further develop a strategy, *Boolean Expression Coercion* (BEC), for systematically choosing value mutations. BEC extracts value mutations from the model of a separate SMT solver call that leverages constraints derived from the original seed formula. We realized Value Mutation Testing via BEC in a prototype tool: ValFuzz. In a preliminary evaluation with ValFuzz, we have already uncovered and confirmed 11 new bugs in the comprehensively tested, state-of-the-art SMT solvers, Z3 and CVC4. Developers of these solvers greatly appreciated our bug reports. We further demonstrate with ValFuzz, using a set of known bugs, that BEC is an effective method for picking value mutations. In the future, we aim at a more thorough evaluation of Value Mutation Testing to understand its full potential.

1 INTRODUCTION

Satisfiability Modulo Theory (SMT) solvers evaluate the satisfiability of first-order logic formulas augmented with functions from various additional theories. They are key components in tools with critical security implications such as program verifiers [2, 8, 13]. Thus errors in the underlying solvers for commercial [1] and open source [12] projects dependent on these tools can have serious potential consequences. This is true especially for hard-to-find *soundness* bugs, where the SMT solver silently outputs an incorrect answer. Recent work has also demonstrated that SMT solvers are less reliable than previously thought [9, 15, 16]. Furthermore, because SMT formulas are a highly structured input format, it is difficult to test these solvers via classic automated testing techniques [5]. Consequently, finding and fixing bugs in SMT solvers is both a challenging open research question and of vital importance.

We observed that many SMT solver bugs are triggered only by a specific set of constant values in the triggering formula. For example, consider Figure 1 which shows a soundness bug in Z3:

```
(declare-funs a b () String)
(assert (and (= (str.++ (str.substr
      "1" 0 (str.len a)) "0") b)
    (< (str.to.int b) 11 )))
(check-sat)
```

```
(declare-funs a b () String)
(assert (and (= (str.++ (str.substr
      "1" 0 (str.len a)) "0") b))
    (< (str.to.int b) 0 )))
(check-sat)
```

Fig. 1. Left: An SMT-LIB formula. Right: Mutating the highlighted constant triggers a bug in Z3 (#4153).[1]

When we mutate a constant value in the correctly handled formula in the left of Figure 1 to produce the formula on the right, Z3 outputs "sat" for this new formula, rather than the correct answer "unsat", indicating a critical *soundness* bug. Thus we propose that this transformation, a *Value Mutation*, can be used to generate many syntactically valid test cases that may trigger bugs in SMT solvers from a corpus of non-buggy seed files. By performing only type preserving mutations, we circumvent the challenge posed by the structured nature of the input language.

In practice, however, finding a value mutation of the formula in the left of Figure 1 that triggers this bug from the search space of possible value mutations is non-trivial. Using randomized value mutations, we were unable to reproduce this bug in practice from the structure of the non-triggering seed. While utilizing branch or code coverage as feedback for choosing new inputs in automated testing has been an effective and popular approach [7, 11, 14], in the context of SMT solvers, Winterer

*Advised by Zhendong Su

[1]We extend the SMT-LIB syntax here with the **declare-funs** command to abbreviate multiple function declarations

et. al. uncovered many critical bugs in Z3 and CVC4 without any significant corresponding increase in code coverage [15]. This observation lead us to develop a novel approach for *choosing* value mutations: *Boolean Expression Coercion* (BEC), in which we use constraints derived from the original formula and an additional call to an SMT solver to pick value mutations that are more likely to trigger divergent, buggy behavior in the solver under test. Indeed, our refined implementation of value mutation via BEC, was able to reproduce the bug shown in Figure 1 in practice.

To evaluate our approach of Value Mutation Testing using Boolean Expression Coercion, we implemented ValFuzz, a differential value mutation fuzzer for SMT-LIB formulas and their solvers [10].[2] Using ValFuzz, we identified a new soundness issue in Z3 and 11 other bugs between Z3 and CVC4 in three months of preliminary testing [3, 6].

2 APPROACH

Our approach consists of Value Mutation Testing via Boolean Expression Coercion.

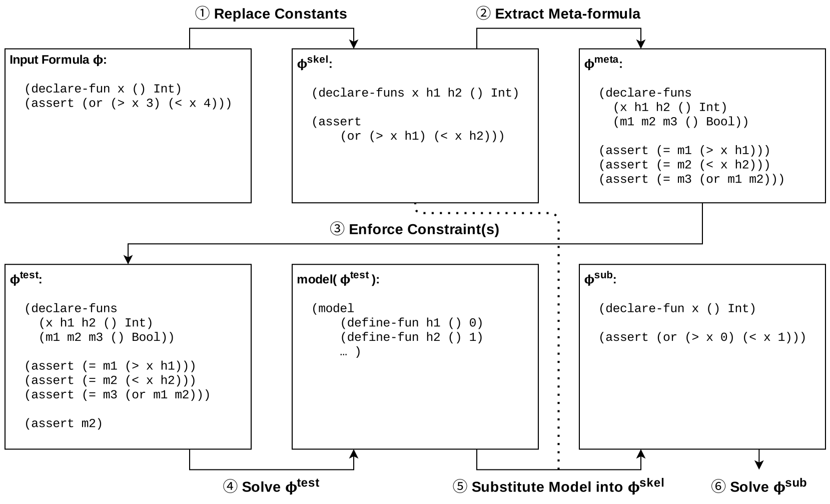
Definition 2.1 describes our core test-case generation technique, *Value Mutation*, more precisely:

Definition 2.1. *We define a* ***Value Mutation*** *for a given seed formula ϕ with n constant values $c_1 \ldots c_n$ as a transformation replacing each constant c_k in ϕ with a constant value c'_k of the same sort such that $\phi \neq \phi[c'_1 \ldots c'_n / c_1 \ldots c_n]$.*

In Value Mutation Testing, we find bugs by performing value mutations on a corpus of seed files and then differentially testing SMT Solvers using the resulting mutant formulas as inputs.

2.1 Picking Value Mutations With Boolean Expression Coercion

For Boolean Expression Coercion, we leverage constraints derived from the boolean sub-expressions of the seed formula, along with an additional call to an SMT solver to obtain our value mutations. By coercing the individual sub-expressions to be *true* or *false*, and allowing the model from this intermediate SMT solver call to dictate which constant values will be used for a given formula, we explore values that we will later show are empirically more likely trigger buggy behavior in the solvers under test.



Input Formula ϕ:

```
(declare-fun x () Int)
(assert (or (> x 3) (< x 4)))
```

ϕ^{skel}:

```
(declare-funs x h1 h2 () Int)

(assert
    (or (> x h1) (< x h2)))
```

ϕ^{meta}:

```
(declare-funs
  (x h1 h2 () Int)
  (m1 m2 m3 () Bool))

(assert (= m1 (> x h1)))
(assert (= m2 (< x h2)))
(assert (= m3 (or m1 m2)))
```

ϕ^{test}:

```
(declare-funs
  (x h1 h2 () Int)
  (m1 m2 m3 () Bool))

(assert (= m1 (> x h1)))
(assert (= m2 (< x h2)))
(assert (= m3 (or m1 m2)))

(assert m2)
```

model(ϕ^{test}):

```
(model
    (define-fun h1 () 0)
    (define-fun h2 () 1)
    … )
```

ϕ^{sub}:

```
(declare-fun x () Int)

(assert (or (> x 0) (< x 1)))
```

Fig. 2. A Single Iteration of Boolean Expression Coercion for an Input Formula ϕ

[2]Differential testing involves testing different versions or configurations of solvers against each other to find discrepancies

We demonstrate this approach through Figure 2 in steps ① through ⑥. Starting with a seed formula ϕ, we first replace each constant value with fresh variables, in this case `h1` and `h2`, representing holes in the formula giving us the *skeleton* of ϕ (step ①). We then create a *meta-formula*, by inserting expressions that monitor the truth values of the boolean expressions with set of fresh variables, `m1 ... m3` (②). Next, we enforce one or more such boolean expressions, here `(< x h2)`, to be either $true$ or $false$ (③). We pass the resulting formula, ϕ^{test}, to an SMT solver (④). If the formula is satisfiable, we take the values given for the variables representing constant holes in the model, and substitute them back into the skeleton of the original formula (⑤). This gives us a new formula, ϕ^{sub}, which we can use to exercise the SMT solvers under test (⑥). If the generated ϕ^{test} is unsatisfiable, we enforce a different constraint and attempt to generate a new value mutation. We repeat this process until each boolean sub-expression has been enforced to be both $true$ and $false$.

***Additional Refinements*.** We also have begun investigating refinements of BEC for choosing Value Mutations. One refinement that has shown promise is adding monitor expressions for variables and sub-expressions of non-boolean types, dictating whether or not they fall within a given abstract domain. For example, we might include `(assert (= m4 (< x 0)))` in step ③ of Figure 2 and later enforce that constraint to ensure that the integer variable `x` is negative.

3 EVALUATION AND CONCLUSIONS

RQ1 - Is value mutation an effective technique for testing SMT solvers? We exercised ValFuzz for three months in parallel with its development on the trunk versions of state-of-the-art SMT solvers Z3 and CVC4. We utilized the SMT-LIB Benchmarks [4], in addition to the test suites for these solvers as seed files. ValFuzz uncovered four bugs in CVC4 and seven bugs in Z3, including one critical soundness bug, all of which were fixed by developers. Project developers responded with positive feedback for our efforts, including the following:

> *"Nice catch, I was able to make some major improvements to the regex algorithm and model construction with this one. Thanks for the report!"*
>
> (Z3 #4271)

RQ2 - Does BEC improve the effectiveness of value mutation testing? We hand-picked bug triggering formulas from the issue trackers of CVC4 and Z3 that contained constant values and used these formulas as seed files for a secondary evaluation. We tracked the quantity and class of each bug found by ValFuzz and a baseline randomized value mutation fuzzer, as well as an efficiency measure.[3] In this evaluation, we found that basic BEC was relatively highly efficient at triggering hard-to-find, critical soundness bugs, finding 2.02 ± 0.20 ($\mu \pm \sigma$) bugs per 1000 solver calls compared to the baseline of only 0.61 ± 0.14 bugs per 1000 solver calls. This translated to a modest, but still statistically significant, increase in the absolute number of bugs found as well.[4] For other kinds of bugs, BEC remained more efficient, but found slightly fewer bugs overall. With additional refinements, such as adding abstract domain constraints, BEC outperforms the baseline in all aspects, finding approximately twice as many soundness bugs while maintaining a high level of efficiency.

Conclusions: We have introduced Value Mutation Testing via Boolean Expression Coercion and shown it can be effective at finding real-world issues in state-of-the-art SMT Solvers. Looking forwards, we aim to further explore BEC and other value mutation strategies to fully realize the potential of this technique.

[3]We use $\textsc{Efficiency} = \frac{\textsc{Bugs Found}}{\textsc{1000 SMT Solver Invocations}}$ for our efficiency metric, as calls to the solvers dominate execution time
[4]Statistical significance determined by a single-tailed Welch's t-test at $p<0.05$

REFERENCES

[1] AdaCore. 2020. *AdaSpark*. Retrieved 2020-11-20 from https://www.adacore.com/about-spark

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*. Springer, 364–387.

[3] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification*. Springer, 171–177.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. *The Satisfiability Modulo Theories Library (SMT-LIB)*. Retrieved 2020-11-01 from www.SMT-LIB.org

[5] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2020. Fuzzing: Challenges and Reflections. *IEEE Software* (2020).

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[7] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Queue* 10, 1 (2012), 20–27.

[8] Uri Juhasz, Ioannis T Kassios, Peter Müller, Milos Novacek, Malte Schwerhoff, and Alexander J Summers. 2014. *Viper: A verification infrastructure for permission-based reasoning*. Technical Report. ETH Zurich.

[9] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 701–712.

[10] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[11] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 653–663.

[12] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, et al. 2020. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 983–1002.

[13] Microsoft Research. 2004. *FStar Language*. Retrieved 2020-11-20 from https://github.com/FStarLang/FStar

[14] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*, Vol. 16.

[15] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25.

[16] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *PLDI*. 718–730.