# Fuzz-testing SMT Solvers with Formula Weakening and Strengthening


Master's Thesis

ETH Zürich

# Fuzz-testing SMT Solvers with Formula Weakening and Strengthening

*Mauro Bringolf*

supervised by
*Prof. Zhendong Su*
*Dominik Winterer*

August 3, 2021

# Fuzz-testing SMT Solvers with Formula Weakening and Strengthening

MAURO BRINGOLF, ETH Zürich, Switzerland

We propose formula weakening and strengthening as a new approach for fuzz testing SMT solvers and increasing their quality and robustness. The mutation rules used in our approach preserve satisfiability as a metamorphic property across all generated mutants, i.e. the satisfiability of all our mutants is known by construction. We derive a theoretical framework to guarantee this property and implement a fuzzer with almost 100 mutation rules for SMT formulas. We use this implementation to stress-test two state-of-the-art SMT solvers Z3 and CVC5. After finding a few crash bugs in both solvers, we proceeded to explore the completeness boundaries of these solvers. This incompleteness testing campaign was well-received by the solver developers and helped uncover regressions, unwanted side effects of code changes as well as potential for improving the solver's completeness. In total, 9 of our 35 reports were fixed and 21 are confirmed but still open.

## 1 INTRODUCTION

A *satisfiability modulo theories* (SMT) solver decides the satisfiability of first order logic formulas with additional structures such as functions and relations from theories about strings, bit-vectors, real numbers, arrays et cetera. A logic is a restricted class of formulas with specific background theories [2], e.g. `QF_S` is the logic of quantifier-free string formulas. Logics may also combine multiple theories and the satisfiability problem is decidable for some logics but undecidable for others [3]. To deal with this complexity, SMT solvers implement a vast number of heuristics and logic-specific solvers. SMT solving has numerous applications in the field of programming languages such as program analysis, program synthesis and program verification. Tools from these areas are typically built on top of an SMT solver as a core component which makes performance a primary quality criterion for a competitive SMT solver, motivating sophisticated optimization efforts from the side of solver developers. In recent years, Z3 [7] and CVC5[1] [1] have consistently been among the most powerful and widely used SMT solvers evidenced for example by their rankings in SMT-COMP [6] or number of GitHub stars (6.5k for Z3, 504 for CVC5).

Since techniques like program analysis and verification are applied predominantly in safety-critical domains, we argue that SMT solving is safety-critical in itself: An SMT solver making a false decision on a formula can directly translate to a safety-critical program being verified against a specification it does not fulfill. This need for a high degree of reliability spurred the research area of testing and validating SMT solvers. One of the most successful strategies has been random testing (fuzz-testing) and multiple fuzzing campaigns have been conducted leading to the discovery and fixing of hundreds of bugs in both Z3 and CVC5.

A crucial part of a fuzzer is its mutation strategy, i.e. how new test cases are produced from existing ones. Different approaches vary in effectiveness and assessing their quality and finding the most productive ones is still an open research problem.

*Formula weakeneing and strengthening.* As our first contribution, we present a new design for such a mutation strategy which we call **weakening and strengthening**. The basic idea is to either weaken the constraints of a satisfiable formula or strengthen them in an unsatisfiable one. Relaxing the constraints of a satisfiable formula $\varphi$ necessarily results in a satisfiable mutant $\varphi'$,

[1]During the timeframe of our work, CVC4 was renamed to CVC5 but the latest release stayed CVC4-1.8. We will always refer to the development version as CVC5.

Author's address: Mauro Bringolf, mauro@bringolf.com, ETH Zürich, Zürich, Switzerland.

Fig. 1. Formula (a) is satisfiable and the conjunct `(> c 0)` can be weakened to `(not (= c 0))`, maintaining satisfiability since `(> c 0)` implies `(not (= c 0))` for any `c`. Consequently, the resulting formula (b) is satisfiable by construction. However, formula (b) is from an soundness bug report found by another fuzzing campaign (https://github.com/Z3Prover/z3/issues/5329): The solver returned `unsat` at the time.

```
(declare-const a Real)
(assert (forall ((b Real)) (exists ((c Real)) (and
  (= (+ a (* c (+ b c) (+ b c))) 0)
  (< a 0)
  (> c 0)
))))
(check-sat)
```

(a) Seed formula which was correctly decided as `sat` by Z3 with the option `nlsat.randomize=false`.

```
(declare-const a Real)
(assert (forall ((b Real)) (exists ((c Real)) (and
  (= (+ a (* c (+ b c) (+ b c))) 0)
  (< a 0)
  (not (= c 0))
))))
(check-sat)
```

(b) Satisfiable formula for which Z3 with the option `nlsat.randomize=false` returned `unsat`: a soundness bug.

since any model of $\varphi$ is also one of $\varphi'$. Similarly, unsatisfiability is preserved when strengthening constraints in an unsatisfiable formula. Thus, the resulting testing strategy is metamorphic [5], e.g. our mutation rules guarantee unchanged satisfiability of the formula and do not require differential testing of the mutants across multiple solvers. Additionally, this allows generating mutants with known satisfiability which solvers are unable to decide (return `unknown` or do not terminate). We will use such undecided mutants to explore new forms of testing apart from the search for conventional bugs.

*Example.* Let us consider a concrete example which was not found by our fuzzing campaign but reconstructed from a bug report on the Z3 issue tracker. We borrow this case from another fuzzing campaign to show that our technique has the capabilities to find soundness bugs. Consider the two formulas in Fig. 1.

Any model $m$ of $\varphi$ will satisfy all three conjuncts inside the quantifiers for one assignment of `c` for all possible assignments of `a`. Now, replace `(> c 0)` with `(not (= c 0))`, which it implies for any `c`. The resulting formula $\varphi'$ is shown in List. 1. Intuitively, $m$ is also a model for $\varphi'$ since we weakened one of its constraints. The exact reasoning is given in Section 3. This means we can construct new test cases with a correct-by-construction oracle from existing ones. Here, $\varphi'$ revealed a soundness bug in one configuration of Z3 in June 2021 and could have been found as a weakening of a correctly decided formula.

*Incompleteness testing.* Our second contribution is a practical evaluation of testing SMT solvers for incompleteness. We use the described mutation strategy to explore the completeness boundaries of Z3 and CVC5, i.e. search for mutants for which a solver returns `unknown`. This result indicates that the solver deemed the formula unfeasible to decide and stopped. Such behavior is to be expected sometimes, as solvers support not only decidable but also undecidable logics. Although not typically considered by fuzzing campaigns, we have found that `unknown` cases can lead to the discovery of problems in solvers that developers are interested in and willing to fix. Fig. 2 contains one of our reported formulas for which CVC5 returned `unknown`. The responding developer commented *"I'll*

Fig. 2. The formula from one of our reports in the CVC5 solver, shown once as SMT-LIB code in (a) and once in mathematical notation in (b). Lines 3 and 4 from (a) are connected by conjunction $\wedge$ in (b). https://www.github.com/cvc5/cvc5/issues/6798.

```
1 (declare-const a Bool)
2 (declare-const b Real)
3 (assert (or a (= 0 (* b b b))))
4 (assert (> (* b b) 3))
5 (check-sat)
```

$$(a \vee 0 = b \cdot b \cdot b) \wedge (b \cdot b > 3)$$

*work on a fix."* and explicitly labelled the issue as bug. On another issue[2], two CVC5 developers looked at the case in detail and commented their analysis. On the Z3 front, developers have fixed multiple of our reports of unknown formulas (Appendix A).

The remainder of this paper is structured as follows. In Section 2 we briefly discuss the status quo of SMT solver fuzzing. Then, Section 3 presents a theoretical framework and correctness proof of our method, along with templates for mutation rules and the conceptual difficulties we had to overcome. The theoretical design is general and we explain some aspects of how our implementation instantiates the framework in Section 4. In Section 5, we present incompleteness testing as a new testing method which we think can further improve the quality of SMT solvers and complements existing approaches. Finally, Section 6 discusses the results of our fuzzing campaign before Section 7 concludes.

## 2 BACKGROUND

### 2.1 SMT-LIB

SMT problems can be expressed in the SMT-LIB[3] language [2] and in this work we use it interchangeably with mathematical notation for formulas. An SMT-LIB script contains a sequence of commands instructing an SMT solver to create formulas and process them. The commands used in our examples are `declare-const` to declare a constant, `assert` to create a formula and `check-sat` to check satisfiability of the asserted formula. The semantics of multiple assertions in one script is their conjunction and we will identify a script with the conjunction of all its asserted formulas, illustrated in Fig. 2. Quantification of a formula `phi` over a variable `x` with sort `T` is written as `(forall ((x T)) phi)` or `(exists ((x T)) phi)` respectively. Note that the type of a variable is called its sort in SMT terminology and we use both terms.

### 2.2 Past SMT fuzzing campaigns

We briefly present five recent fuzz-testing campaigns on SMT solvers which have employed a variety of techniques. STORM [8] takes an arbitrary seed formula and generates mutants that are satisfiable by construction, testing seven solvers across many logics. Bugariu and Müller [4] synthesize string formulas whose satisfiability is also known by construction using satisfiability-preserving mutation rules. Semantic Fusion [13] combines two equi-satisfiable formulas into a formula by first concatenating them and then rewriting terms with shared variables in order to preserve satisfiability. OpFuzz [12] replaces operators with other operators of conforming type, e.g. `(+ 1 2)` becomes `(* 1 2)` by operator replacement. Falcon [14] focuses on the correlation between solver configurations and formulas in order to test more solver options efficiently. Sparrow [15] is

[2]https://github.com/cvc5/cvc5-projects/issues/279, to be discussed later as List. 4.
[3]We are always referring to SMT-LIB version 2.6.

concurrent[4] work with our testing campaign and employs a very similiar technique but focuses only on correctness testing.

Table 1. A comparison of different SMT fuzzing campaigns. A question mark indicates that the authors did not reveal the length of their testing campaign.

| fuzzing technique | reported | fixed | campaign duration [months] | testing method |
|---|---|---|---|---|
| OpFuzz | 1092 | 685 | 9 | differential |
| Falcon | 518 | 469 | 6 | differential |
| Sparrow | 84 | 55 | ? | metamorphic |
| Semantic Fusion | 45 | 41 | 4 | metamorphic |
| STORM | 29 | 19 | 3 | metamorphic |
| Bugariu & Müller | 5 | 3 | ? | metamorphic |
| Our work | 35 | 9 | 5 | metamorphic |

The results of these approaches together with our own are depicted in Tab. 1. In this comparison we use two categories for the number of bug findings: bugs that the authors *reported* on the issue trackers and reports that have been *fixed* by the developers. We do not further differentiate between bug types (crash, soundness etc.) for multiple reasons: Not all authors provide bug counts for the same categories[5], later campaigns test solver versions robustified by previous testing efforts, different campaigns test different number of solvers and logics and most importantly the options and number of configurations for solvers varies drastically across campaigns. In our view, the category of fixed bugs is mostly robust against these factors and the preferrable measure for the effectiveness of a testing strategy.

## 3 DESIGN

In this section, we first formalize weakening and strengthening of subformulas as mutation rules and prove that they are satisfiability-preserving. Second, we propose a mutation algorithm for applying these rules to a set of seed formulas. Finally, we give two templates for weakening and strengthening rules used in our implementation.

### 3.1 Theoretical framework

*3.1.1 Preliminaries.* We describe first-order logic formulas over many-sorted terms with functions and relations as used in satisfiability modulo theories (SMT):

$$\begin{aligned} \varphi &::= \quad r(t_1, \ldots, t_n) \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists x \colon \varphi \\ t &::= \quad f(t_1, \ldots, t_n) \mid x \end{aligned}$$

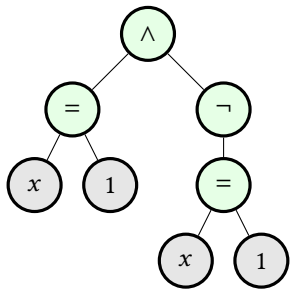
Symbols $r$ and $f$ represent relation and function symbols from background theories and $x$ represents a variable. Terms are many-sorted and boolean terms are called *predicates*. The set of *free variables* in a formula $\varphi$ is denoted as $FV(\varphi)$ and a formula without free variables is called a *sentence*. A *model* of a formula assigns an interpretation (constants, functions, relations, sorts) to each free symbol such that it evaluates to $true$. A formula is called *satisfiable* if such a model exists and *unsatisfiable* otherwise. Moreover, a *tautology* is a formula $\varphi$ for which any interpretation is a model which is denoted by $\models \varphi$. Two formulas are called *equi-satisfiable* if either both are satisfiable or both are unsatisfiable.

[4]To be presented at FSE'21 on 25th of August 2021.

[5]e.g. some authors count soundness and invalid model bugs together as correctness bugs.

*3.1.2 Subformulas and substitution.* Consider a formula generated by the grammar above. Its syntactic structure is described by a unique abstract syntax tree with nodes labelled as non-terminal ($\varphi$ and $t$) or terminal symbols ($x, f, r, \wedge, \exists x:, \ldots$). We define a *subformula* to be a formula identified with a subtree rooted at a node labelled $\varphi$, e.g. one representing a boolean sorted part of the formula. The purpose of this is to distinguish between multiple occurences of syntactically equal parts within a larger formula. As a running example, the formula $x = 1 \wedge \neg(x = 1)$ has four subformulas: itself, $\neg(x = 1)$ and the two occurences of $x = 1$. Its abstract syntax tree is depicted in Fig. 3. Although syntactically equal, the two occurences of $x = 1$ are represented by distinct parts of the abstract syntax tree and thus considered different in our context.

Fig. 3. Abstract syntax tree of the formula $x = 1 \wedge \neg(x = 1)$, with all of its subformulas highlighted in green. Note how the two subtrees labelled = represent the same formula $x = 1$ but must be considered as separate entities for our purposes: Mutations applied to either of them have the opposite effect on overall satisfiability due to ¬ on the right side.




We now define a substitution which replaces a specific subformula as opposed to all occurences as in conventional syntactic substitution. Let $\varphi$ be a formula, $T$ be its abstract syntax tree and $F$ a subformula of $\varphi$. We define $\varphi[F \mapsto G]$ to be the formula represented by $T$ where $F$ is replaced with $G$. In our running example, if $\varphi$ is the overall formula, $F$ the subformula representing the left occurence of $x = 1$ and $G$ an abstract syntax tree representing $y = 2$, then the substitution $\varphi[F \mapsto G]$ yields $y = 2 \wedge \neg(x = 1)$ and not $y = 2 \wedge \neg(y = 1)$ as with conventional syntactic substitution.

*3.1.3 Weakening and strengthening.* Next, we precise the notion of weaker and stronger formulas and introduce the concept of $parity$ of subformulas.

Definition 1 (weaker/stronger). *Let $\varphi_1, \varphi_2$ be formulas such that $FV(\varphi_1) = FV(\varphi_2)$. We call $\varphi_1$ weaker than $\varphi_2$ iff:*

$$\models \forall x_1, \ldots, x_n : \varphi_2 \rightarrow \varphi_1$$

*where $x_1, \ldots, x_n = FV(\varphi_1)$. Vice versa, we call $\varphi_2$ stronger than $\varphi_1$.*

Definition 2 (parity). *For a formula $\varphi$ with a subformula $F$, we define $parity(F, \varphi)$ recursively as[6]:*

[6]Technically, the cases used in this definition are not non-overlapping and contain a slight ambiguity: If $F$ represents $\varphi$ then one of the remaining cases also applies (except when $\varphi = r(t_1, \ldots, t_n)$). In this scenario the first case takes precedence and the parity is 1.

$$parity(F,\varphi) := \begin{cases} 1 & \text{if } F \text{ represents } \varphi \\ -1 \cdot parity(F,\varphi') & \text{if } \varphi = \neg\varphi' \\ parity(F,\varphi_1) & \text{if } \varphi = \varphi_1 \wedge \varphi_2 \text{ and } F \text{ in } \varphi_1 \\ parity(F,\varphi_2) & \text{if } \varphi = \varphi_1 \wedge \varphi_2 \text{ and } F \text{ in } \varphi_2 \\ parity(F,\varphi') & \text{if } \varphi = \exists x\colon \varphi' \end{cases}$$

*If $parity(F,\varphi) = 1$, then $F$ is called positive and negative otherwise.*

The *parity* function is the link between weakening and strengthening a subformula and its overall semantic effect. More precisely, the parity of a subformula captures whether weakening or strengthening it has the same or the oppositve effect on surrounding formula:

Lemma 1. *Let $\varphi$ be a formula with a subformula $F$. For any $G$ weaker than $F$, we have:*

$$\text{if } F \text{ positive in } \varphi \text{ then} \quad \models \forall x_1,\dots,x_n\colon \varphi \to \varphi[F \mapsto G] \tag{1}$$

$$\text{if } F \text{ negative in } \varphi \text{ then} \quad \models \forall x_1,\dots,x_n\colon \varphi[F \mapsto G] \to \varphi \tag{2}$$

*where $FV(\varphi) = x_1,\dots,x_n$.*

Proof. By induction over $\varphi$.

**Case** $\varphi = F$: Both implications (1) and (2) hold, since $parity(F,\varphi) = 1$ and $\varphi[F \mapsto G] = G$.

**Case** $\varphi = \neg\varphi_1$: Let $x_1,\dots,x_n = FV(\varphi) = FV(\varphi_1)$. Assume $parity(F,\varphi) = 1$, then $parity(F,\varphi_1) = -1$ and by induction hypothesis (2) for $\varphi_1$:

$$\models \forall x_1,\dots,x_n\colon \varphi_1[F \mapsto G] \to \varphi_1$$

By contraposition, we obtain

$$\begin{aligned} &\models \forall x_1,\dots,x_n\colon \neg\varphi_1 \to \neg(\varphi_1[F \mapsto G]) \\ \iff &\models \forall x_1,\dots,x_n\colon \neg\varphi_1 \to \neg\varphi_1[F \mapsto G] \end{aligned}$$

so (1) holds for $\varphi$. The proof of (2) is symmetric.

**Case** $\varphi = \varphi_1 \wedge \varphi_2$: Let $x_1,\dots,x_n = FV(\varphi)$. By symmetry, assume $F$ is a subformula of $\varphi_1$. Let $FV(\varphi_1) = x_1,\dots,x_k$ with $k \le n$. Assume $parity(F,\varphi) = 1$, then $parity(F,\varphi_1) = 1$ and by induction hypothesis (1) for $\varphi_1$:

$$\models \forall x_1,\dots,x_k\colon \varphi_1 \to \varphi_1[F \mapsto G]$$

Since $x_{k+1},\dots,x_n \notin FV(\varphi_1)$ this extends vacuously to:

$$\models \forall x_1,\dots,x_n\colon \varphi_1 \to \varphi_1[F \mapsto G]$$

We can then deduce (1) for $\varphi$:

$$\begin{aligned} &\models \forall x_1,\dots,x_n\colon \varphi_1 \wedge \varphi_2 \to \varphi_1[F \mapsto G] \wedge \varphi_2 \\ \iff &\models \forall x_1,\dots,x_n\colon \varphi_1 \wedge \varphi_2 \to (\varphi_1 \wedge \varphi_2)[F \mapsto G] \end{aligned}$$

Where the last equivalence holds because $F$ is a subtree in the abstract syntax tree of $\varphi_1$ and thus the substitution has no effect when applied to $\varphi_2$. The proof of (2) is similar.

**Case** $\varphi = \exists x\colon \varphi_1$:

Assume $parity(F,\varphi) = 1$, then $parity(F,\varphi_1) = 1$ and by induction hypothesis (1) for $\varphi_1$:

$$\models \forall x_1, \ldots, x_n \colon \varphi_1 \rightarrow \varphi_1[F \mapsto G] \tag{3}$$

where $x_1, \ldots, x_n = FV(\varphi_1)$. If $x$ does not occur free in $\varphi_1$, then $FV(\varphi) = FV(\varphi_1)$ and we can directly conclude (1) for $\varphi$:

$$\begin{aligned} &\models \forall x_1, \ldots, x_n \colon (\exists x \colon \varphi_1) \rightarrow (\exists x \colon \varphi_1[F \mapsto G]) \\ \Longleftrightarrow &\models \forall x_1, \ldots, x_n \colon (\exists x \colon \varphi_1) \rightarrow ((\exists x \colon \varphi_1)[F \mapsto G]) \end{aligned}$$

If $x \in FV(\varphi_1)$, let $x = x_n$. Then (3) implies:

$$\begin{aligned} &\models \forall x_1, \ldots, x_{n-1} \colon (\exists x_n \colon \varphi_1) \rightarrow (\exists x_n \colon \varphi_1[F \mapsto G]) \\ \Longleftrightarrow &\models \forall x_1, \ldots, x_{n-1} \colon (\exists x_n \colon \varphi_1) \rightarrow ((\exists x_n \colon \varphi_1)[F \mapsto G]) \end{aligned}$$

Since $FV(\varphi) = FV(\varphi_1) - \{x\}$, this is exactly (1) for $\varphi$. The proof of (2) is similar. □

Given these two relations between implication of a subformula and implication of the overall formula, we can derive four weakening and strengtheing rules:

Theorem 1. *Let $F, F_w, F_s$ be formulas and $\varphi$ sentence such that $F$ is a subformula of $\varphi$, $F_w$ is weaker than $F$ and $F_s$ is stronger than $F$. Then the following statements hold:*

(1) *$F$ positive, $\varphi$ satisfiable $\Longrightarrow$ $\varphi[F \mapsto F_w]$ satisfiable*
(2) *$F$ negative, $\varphi$ satisfiable $\Longrightarrow$ $\varphi[F \mapsto F_s]$ satisfiable*
(3) *$F$ negative, $\varphi$ unsatisfiable $\Longrightarrow$ $\varphi[F \mapsto F_w]$ unsatisfiable*
(4) *$F$ positive, $\varphi$ unsatisfiable $\Longrightarrow$ $\varphi[F \mapsto F_s]$ unsatisfiable*

Proof. (1) By Lemma 1 we have:

$$\models \varphi \rightarrow \varphi[F \mapsto F_w]$$

Thus, if a model of $\varphi$ exists it must also be a model of $\varphi[F \mapsto F_w]$.

(2) Consider $\varphi[F \mapsto F_s]$ with negative subformula $F_s$. Now $\varphi$ weakens $F_s$ to $F$ in a negative position, so by Lemma 1:

$$\models \varphi \rightarrow \varphi[F \mapsto F_s]$$

(3) By Lemma 1 we have:

$$\models \varphi[F \mapsto F_w] \rightarrow \varphi$$

So if no model exists for $\varphi$, then no model can exist for $\varphi[F \mapsto F_w]$.

(4) Consider the formula $\varphi[F \mapsto F_s]$ with positive subformula $F_s$. Now $\varphi$ weakens $F_s$ to $F$ in a positive position, so by Lemma 1:

$$\models \varphi[F \mapsto F_s] \rightarrow \varphi$$

□

*3.1.4 Ambiguous parities.* Unfortunately, the fact that all subformulas have a parity is no longer true when extending our small logic language to the full SMT-LIB language. More precisely, it is not possible to simply extend definition 2 to the syntax of SMT-LIB predicates such that Lemma 1 and consequently Theorem 1 hold. A counterexample for boolean equality is given in List. 1. The same issue occurs with the boolean operators shown in List. 2.

```
(= (= x 1) true) ; p1
(= (>= x 1) true) ; p2
; p1 ==> p2
(= (= x 1) false) ; p3
(= (>= x 1) false) ; p4
; p4 ==> p3
```

Listing 1. The same local weakening step from `= x 1` to `>= x 1` in the same syntactic position weakens p1 to p2 but strengthens p3 to p4.

```
(xor f1 f2)
(= f1 f2)
(distinct f1 f2)
(ite f1 f2 f3)
```

Listing 2. Boolean operators with ambiguous parities where local weakening and strengthening is not directly possible.

A similar issue can arise in `let`-terms, e.g. in `(let ((x a1)) (and (not x) x))`. Again, in this formula the predicate `a1` occurs in an ambiguous position. However, `a1` can also have unambiguous parity: Let $p_1, \ldots, p_n$ be the parities of all bound occurences of `x` in the `let`-body. If all $p_i$ are unambiguous and equal, then `a1` has that same parity. Otherwise, `a1` has ambiguous parity as in the example above. We grouped all of these ambiguities into three categories to be handled separately via preprocessing:

- `xor` and Boolean-sorted `=, distinct`: We randomly rewrite each occurence of such an operator with one of `=>, and, or` and re-classify the modified seed according to its satisfiability.
- Boolean-sorted `ite`: Since `ite` terms are known to be involved in many rewrite rules which are a challenging component of solvers and have been one source of unsoundness bugs in the past [10], we decided to keep these terms and ignore (i.e. never mutate) the ambiguous subformula.
- `let` with Boolean-sorted variables: These are only ambiguous if a boolean sorted variable occurs in positive and negative positions inside the `let`-body. In such a case, we insert a `not` around all negative occurrences, making the variable definition in the `let` itself positive.

## 3.2 Mutation algorithm

In fuzz testing we are generally trying to generate as diverse a set of mutants as possible. This means mutations should keep moving away from seed tests in order to reach many different kinds of inputs. Naturally, two iterative strategies for applying single mutation rules lend themselves. With an incremental strategy we keep applying mutation steps to results of previous mutations. In contrast, a non-incremental strategy keeps applying single mutation steps to seed tests, discarding the mutant after testing it. At first glance, it seems that incremental mutation is clearly superior to non-incremental mutation since it covers a lot more ground. One constraint with incremental mutation is that mutation rules must not increase formula size, otherwise the fuzzer will run out of memory rather sooner than later and we ran into this problem. Additionally, through our

experiments we have discovered another subtle condition that the mutation rule set must meet in order to work as expected. This is described in the next section (3.2.1) and leads us to our choice of mutation algorithm (3.2.2).

*3.2.1 Rule applicability and preserving it.* Applying a mutation has the potential side effect of changing the set of applicable rules. Consider for example the following two rules for integers:

$$i_1 = i_2 \implies i_1 \leq i_2 \tag{4}$$

$$i_1 = i_2 \implies |i_1| = |i_2| \tag{5}$$

Note how applying rule (4) to a formula decreases the applicability of (5) in the result. Consequently, when incrementally applying one of these two rules randomly, sooner or later the process will eliminate rule (5) completely by replacing all = with ≤ via rule (4). If there are no other rules, the fuzzer runs out of applicable mutations and gets stuck. As a solution, one could reasonably propose to simply restart the fuzzer in such a case. However, say we were to swap out the second rule for another one and use the following two rules:

$$i_1 = i_2 \implies i_1 \leq i_2 \tag{6}$$

$$i_1 \leq i_2 \implies i_1 + c \leq i_2 + c \tag{7}$$

Rule (6) enables rule (7) which then preserves its own applicability while disabling (6). As a result, the fuzzer does not get stuck but instead keeps applying rule (7). This particular behavior is what we found in our earliest experiments with this fuzzing approach when using a purely incremental strategy and decided it to be undesirable. The reason is that it makes the fuzzer abandon rule (6) completely which makes it questionable whether this is preferable to a non-incremental strategy. Note that the underlying problem is not easily quantified, as both strategies equally produce an infinite set of distinct mutants. However, based on intuition we value a set of mutants generated by distinct mutation rules higher than mutants generated by a single or few rules. In the following we will refer to the above described phenomenon as *trivial rule cycles* and try to avoid it.

*3.2.2 Final choice of mutation algorithm.* The above considerations of trivial rule cycles led us to add a parameter walk-length ($wl$), used to control the number of incremental mutation steps performed. We write $dom(m)$ for the domain of a mutation rule $m$, i.e. the set of formulas to which the rule is applicable and $u.a.r.$ for *uniformly at random*. Our mutation algorithm is parametrized over a set of mutation rules $M$, the number of iterations per seed $i$ and the walk-length $wl$:

**Algorithm 1** Repeated incremental mutation
**Input:** An SMT formula *seed*

$n \leftarrow 0$
$mutant \leftarrow seed$
**while** $n < i$ **do**
  **if** $n \mod wl = 0$ **then**
    $mutant \leftarrow seed$
  **end if**
  $rule \leftarrow$ choose u.a.r. from $\{rule \in M | mutant \in dom(rule)\}$
  $mutant \leftarrow rule(mutant)$
  $test(mutant)$
**end while**

Fig. 4. Weakening of the relation `(= String String Bool)` to `(str.contains String String Bool)` preserving the satisfiability of the formula.

```
(declare-const s String)
(assert (= s "." ))
(check-sat)
```

(a) A satisfiable formula asserting that `s` is equal to `"."`.

```
(declare-const s String)
(assert (str.contains s "."))
(check-sat)
```

A weakened version of (a) saying that `s` contains `"."`.

This design allows choosing independently the number of mutations performed on a single seed and the number of mutations performend incrementally. In theory, this seems equivalent to a purely incremental strategy with $i$ set to $wl$ as seeds are picked many times when the fuzzer is running long enough. However, we wanted a seed once picked to generate multiple random walks instead of just one before having to wait for a long time until it is picked again.

### 3.3 Rule templates

*3.3.1 Operator replacement.* Our approach naturally accomodates for a subset of an existing, highly effective fuzzing technique called Type-aware Operator Mutation [12]. This mutation strategy replaces operators with other operators of conforming type, e.g. `(+ 1 2)` becomes `(* 1 2)`. For boolean operators (predicates) we can apply this strategy while satisfing the semantic constraints of weakening and strengthening. Fig. 4 contains an example where the relation = is weakened to `str.contains`.

Definition 3 (operator replacement rule). *Let $r_1 : \tau^n \to Bool$, $r_2 : \tau^n \to Bool$ such that*

$$\forall x_1, \ldots, x_n \in \tau . r_1(x_1, \ldots, x_n) \implies r_2(x_1, \ldots, x_n)$$

*The corresponding **operator replacement rule** substitutes one for the other:*

$$\texttt{(r1 t1 ... tn)} \Rightarrow \texttt{(r2 t1 ... tn)}$$

Many intuitive rules (including operator replacement) can be seen as instantiations of the following generic rule template:

Definition 4 (Homomorphism rule). *Let $R, S$ be two sorts and $n \in \mathbb{N}$, $r : R^n \to Bool$, $s : S^n \to Bool$, $f : R \to S$, $P : r^n \to Bool$ such that*

$$\forall x_1, \ldots, x_n \in R . P(x_1, \ldots, x_n) \wedge R(x_1, \ldots, x_n) \Rightarrow S(f(x_1), \ldots, f(x_n))$$

*This yields the following **homomorphism rule** template:*

$$\texttt{(R t1 ... tn)} \implies \texttt{(=> (P t1 ... tn) (S (f t1) ... (f tn)))}$$

An example is given in Fig. 5.

## 4 IMPLEMENTATION

Our Python implementation first parses and typechecks a given seed formula before running the mutation algorithm described in 3.2.2. The *test* method passes the mutant formula to all solvers under test and records crash or unsoundness bugs. Typechecking allows us to apply rules only when they are definitely applicable, for example the rule shown in Fig. 5 only applies to `(> 2 1)`

Fig. 5. An example instance of our rule template for homomorphisms. The integer ordering relation < is translated into the `str.suffixof` relation among substrings of the same string. Because of the semantics of `str.substr`, this property only holds for non-negative integers which is encoded as the precondition `(and (>= 2 0) (>= 1 0))` on the right handside. Note that the rule holds for any string which is thus picked randomly.

```
(declare-const s String)
(assert (= s "abc"))
(assert (> 2 1))
(check-sat); sat
```

(a) The rule is applied to the highlighted subformula, the highlighted part is implied by `> 2 1`.

```
(declare-const s String)
(assert (= s "abc"))
(assert (=>
  (and (>= 2 0) (>= 1 0))
  (str.suffixof
    (str.substr s 2 (str.len s))
    (str.substr s 1 (str.len s)))))
(check-sat); sat
```

(b) The result of the mutation



for integers but not the syntactically equal formula for Reals. Another common occurence of this dependency on typechecking are weakening or strengthening rules of the equality operator =, such as the one in Fig. 4.

### 4.1 Mutation rules

A rule consists of two patterns, the left and right handside of an implication or equivalence. Free variables are instantiated randomly as described in 4.2. Independent of its logical structure, a rule can be implemented from left to right (weakening), right to left (strengthening) or both. Additionally, equivalence rules work as weakening and strengthening rules in both directions. To illustrate why we did not implement both directions for all rules, consider this rule (rule (11) below in 4.1.1):

$$\texttt{(= t1 ... tn)} \Rightarrow \texttt{(= (f t1) ... (f tn))}$$

It says that for equal arguments, a function `f` returns equal results. Implementation in the strengthening direction means simply removing the function application. But to completely implement this rule from left to right, we would need to provide a function matching the sort of the $T_i$. Thus, chose to only implement the strengthening direction in this case and a few others.

*4.1.1 Core logic rules.* This category collects first order logic implications without relying on any theory semantics except core. In rules (19) and (20) we write $[x \mapsto E]$ for the conventional substitution operation, i.e. all occurences of $x$ are replaced by $E$. In rule (9) the symbol $\oplus$ stands for the `xor` operator.

$$\varphi_1 \wedge \varphi_2 \Rightarrow \varphi_1 \vee \varphi_2 \tag{8}$$
$$\varphi_1 \oplus \varphi_2 \Rightarrow \varphi_1 \vee \varphi_2 \tag{9}$$
$$\forall x.\varphi \Rightarrow \exists x.\varphi \tag{10}$$
$$x_1 = \ldots = x_n \Rightarrow f(x_1) = \ldots = f(x_2) \tag{11}$$
$$\varphi_1 \vee \varphi_2 \Rightarrow \exists b.ite(b, \varphi_1, \varphi_2) \tag{12}$$
$$ite(b, \varphi_1, \varphi_2) \Rightarrow (b \rightarrow \varphi_1) \tag{13}$$
$$ite(b, \varphi_1, \varphi_2) \Rightarrow (\neg b \rightarrow \varphi_2) \tag{14}$$
$$(\varphi_1 \rightarrow \varphi_2) \Rightarrow ite(\varphi_1, \varphi_2, \top) \tag{15}$$
$$(\varphi_1 \rightarrow \varphi_2) \Rightarrow ite(\neg\varphi_1, \top, \varphi_2) \tag{16}$$
$$(\varphi_1 \rightarrow \varphi_2) \Rightarrow \forall b.(\varphi_1 \wedge b \Rightarrow \varphi_2 \wedge b) \tag{17}$$
$$\varphi_1 \vee \varphi_2 \Rightarrow (\neg\varphi_1 \Rightarrow \varphi_2) \tag{18}$$
$$\forall x.\varphi \Rightarrow \varphi[x \mapsto E] \tag{19}$$
$$\varphi[x \mapsto E] \Rightarrow \exists x.\varphi \tag{20}$$
$$\varphi_1 \Rightarrow \varphi_1 \vee \varphi_2 \tag{21}$$
$$\varphi_1 \wedge \varphi_2 \Rightarrow \varphi_1 \tag{22}$$

*4.1.2 Integer and Real rules.* The rules from this category transform relations among integers and reals. In rules (29), (30) and (31) we use the symbol $\odot$ as a placeholder for the relations $=, <, >, \leq, \geq, \neq$ since we implemented many of the possible combinations. Note that depending on the choice of relations, positivity or negativity of the constant $c$ must be controlled as well.

$$n_1 = n_2 \Rightarrow n_1 \geq n_2 \tag{23}$$
$$n_1 > n_2 \Rightarrow n_1 \geq n_2 \tag{24}$$
$$n_1 = n_2 \Rightarrow n_1 \leq n_2 \tag{25}$$
$$n_1 < n_2 \Rightarrow n_1 \leq n_2 \tag{26}$$
$$n_1 < n_2 \Rightarrow n_1 \neq n_2 \tag{27}$$
$$n_1 > n_2 \Rightarrow n_1 \neq n_2 \tag{28}$$
$$n_1 \odot n_2 \Rightarrow (n_1 + c) \odot (n_2 + c) \tag{29}$$
$$n_1 \odot n_2 \Rightarrow n_1 \odot (n_2 + c) \tag{30}$$
$$n_1 \odot n_2 \Rightarrow (n_1 + c) \odot n_2 \tag{31}$$

*4.1.3 String rules.* In the following, we use some convenience syntax in order to make the rules more readable, e.g. $\leq_s$ stands for `str.<=` and from the other operators we omit the prefix `str.` in their proper SMT-LIB names [11].

$$s_1 = s_2 \Rightarrow s_1 \neq (s_1 +\!\!+ s_3) \tag{32}$$
$$prefixof(s_1, s_2) \Rightarrow \exists i\colon substr(s_2, 0, i) \tag{33}$$
$$suffixof(s_1, s_2) \Rightarrow \exists i\colon substr(s_2, i, len(s_2) - i) \tag{34}$$
$$s_1 = s_2 \Rightarrow prefixof(s_1, s_2) \wedge suffixof(s_1, s_2) \tag{35}$$
$$s_1 = s_2 \Leftrightarrow prefixof(s_1, s_2) \wedge prefixof(s_2, s_1) \tag{36}$$
$$s_1 = s_2 \Leftrightarrow suffixof(s_1, s_2) \wedge suffixof(s_2, s_1) \tag{37}$$
$$prefixof(s_1, s_2) \vee suffixof(s_1, s_2) \Rightarrow contains(s_2, s_1) \tag{38}$$
$$s_1 \leq_s s_2 \Rightarrow s_1 \leq_s (s_2 +\!\!+ s_3) \tag{39}$$
$$s_1 \leq_s s_2 \Rightarrow substr(s_1, 0, ite(0 \leq i \leq len(s_1) - 1, i, len(s_1))) \leq_s s_2 \tag{40}$$
$$suffixof(s_1, s_2) \Rightarrow len(s_1) \leq len(s_2) \tag{41}$$
$$prefixof(s_1, s_2) \Rightarrow len(s_1) \leq len(s_2) \tag{42}$$
$$contains(s_1, s_2) \Rightarrow len(s_1) \geq len(s_2) \tag{43}$$
$$contains(s_1, s_2) \Rightarrow len(s_1) \geq len(s_2) \tag{44}$$

*4.1.4 Regex rewrites.* This category of rewrites applies to terms of the form `(str.in_re s r)` by rewriting `r`. The concept of weakening and strengthening naturally extends to the type `RegLan` and we write $r_1 \Rightarrow r_2$ as a shorthand for the mutation rule `(str.in_re s r1)` $\Rightarrow$ `(str.in_re s r2)`. The quantifier in (49) is added around the `str.in_re` operator. The implementation of rule (50) chooses $s_3$ and $s_4$ such that the implication holds, i.e. the corresponding range is larger than that of $s_1$ and $s_2$.

$$r \Rightarrow r^+ \tag{45}$$
$$r \Rightarrow loop(1, n, r) \tag{46}$$
$$r \Leftrightarrow opt(r) \tag{47}$$
$$r_1 +\!\!+ \ \ldots +\!\!+ r_n \Rightarrow union(r_1, \ldots, r_n)^n \tag{48}$$
$$r \Rightarrow \forall x\colon union(r, x) \tag{49}$$
$$range(s_1, s_2) \Rightarrow range(s_3, s_4) \tag{50}$$
$$r \Rightarrow inter(r, r) \tag{51}$$
$$r \Rightarrow union(r, r) \tag{52}$$
$$r^+ \Rightarrow r^* \tag{53}$$
$$inter(r_1, r_2) \Rightarrow union(r_1, r_2) \tag{54}$$

## 4.2 Generating random terms

In many of our mutation rules, there are terms that can be chosen freely. For example, the two quantifier instantiation rules (19) and (20) from 4.1.1 contain a free term variable $E$:

$$\forall x.\varphi \Longrightarrow \varphi[x \mapsto e]$$
$$\varphi[x \mapsto e] \Longrightarrow \exists x.\varphi$$

For mutant diversity and complexity, we adopt the following two-step generation process to produce a random instance for $E$:

(1) Search the current SMT-file for terms of the required sort. If there are any, choose one uniformly at random.
(2) Otherwise, choose uniformly at random from a set of literals.

## 5 INCOMPLETENESS TESTING

Besides finding bugs, we use the approach for incompleteness testing of SMT solvers. In general we refer to a formula for which a solver returns unknown as an incompleteness. Such cases routinely occur when testing SMT solvers with randomly generated formula, so we propose additional constraints for an incompleteness to be interesting from a testing perspective. Let $\varphi_{n-1}, \varphi_n$ be two consecutive mutants in the fuzzing process. By design, either $\varphi_{n-1}$ is satisfiable and $\varphi_{n-1} \Rightarrow \varphi_n$ or $\varphi_{n-1}$ is unsatisfiable and $\varphi_n \Rightarrow \varphi_{n-1}$. Moreover, in both cases the implication comes from a local weakening or strengthening of a subformula in $\varphi_{n-1}$. We investigate two scenarios:

Definition 5 (Incompleteness types). *Let $S$ be a solver and $S_{old}$ an earlier version of it. We distinguish the following two types of incompletenesses:*

- ***Regression (type 1)**: $S_{old}(\varphi_n) = sat/unsat \wedge S(\varphi_n) = unknown$*
- ***Unsupported implication (type 2)**: $S(\varphi_{n-1}) = sat/unsat \wedge S(\varphi_n) = unknown$*

The rationale for type 1 incompletenesses is simple: A recent development has removed support for a previously decidable formula. But of course, with the complexity of the problem domain we cannot and do not expect SMT solvers to increase their completeness monotonically only. But fuzzing for such cases could reveal regressions which turn out to be surprising to the developers and help them discover unwanted side effects of their code changes that do not manifest as soundnenss bug or lead to a crash. With regressions we aim at revealing subtle problems early on during the development process. Note that this type of incompleteness does not require the mutation strategy to be metamorphic, as the mutants are simply tested differentially across multiple solver versions.

List. 3 contains an example of such a regression, found and reported by us in June 2021 and fixed within a week by the Z3 developers.

```
(assert (forall ((v Int)) (= 0 v)))
(assert (= 0 (mod 0 0)))
(check-sat)
```

Listing 3. An unsatisfiable formula which we reported on the Z3 issue tracker in June 2021. The solver's previous release (Z3 version 4.8.10) decided this formula as unsat, while the development version could no longer do so and returned unknown at the time. (https://github.com/Z3Prover/z3/issues/5338)

Type 2 has a different goal, namely to help improve completeness of solvers by searching for two semantically related formulas of which one is decided but the other is not. The idea is that such pairs could suggest improvements to formula rewriters used by solvers in order to rewrite $\varphi_n$ into a form close enough to $\varphi_{n-1}$ such that it becomes decidable. Thus, unsupported implications are not suggested to be defects in solvers but aim to reveal potential for improved rewriting strategies.

For example, in the formula $\varphi$ in List. 4 the obvious simplification of `(or false (= 0.0 s))` to `(= 0.0 s)` suggests itself. At the time of writing, July 2021, the development version of CVC5 cannot decide $\varphi$, returning unknown. But when the rewrite is performed manually, the solver readily returns `sat` for the modified formula. The same behavior occurred using the latest previous release at the time. Thus, from a user perspective it seems as if the solver does not apply the simplifcation, which is undesirable for such a simple proof step. More interestingly, the solver definitely had this simplification implemented and the developers deemed the observed behavior worth investigating. It turned out that the rewrite was indeed applied, but as a side effect led the solver onto an infeasible path and returning unknown. CVC5 is sensitive to the order of operands in `or` and the rewrite

affected this order in an intermediate form of the formula. Two developers indicated in their feedback that this behavior is undesirable but unfortunately also not feasible to avoid in general.

```
(declare-const s Real)
(assert (or (or false (= 0.0 s)) (< (* s (+ 6 (* s 12))) (- 1))))
(check-sat)
```

Listing 4. A formula we reported on the CVC5 issue tracker since it returns unknown, although when manually rewriting `(or false (= 0.0 s))` to `(= 0.0 s)` it is decided correctly. https://github.com/cvc5/cvc5-projects/issues/279.

Bearing in mind that this is a slightly special case: the mutation is an equivalence instead of just an implication. Thus, the fuzzer finds this case by rewriting `(= 0.0 s)` to `(or false (= 0.0 s))` which, in the other direction, is then exactly the suggested rewrite step. In general this is not required and our search is more open-ended as we select interesting cases manually based on rules which are mostly not equivalences.

## 6 EVALUATION

### 6.1 Experimental setup

At the time of writing, our fuzzing campaign has been running for five months during which we concurrently developed the fuzzer. For the first three months we searched exclusively for unsoundness and crash bugs before starting incompleteness testing around three months into the campaign. We deployed the fuzzer on a AMD Ryzen Threadripper 3990X 64-Core Processor with 256GB of RAM running Ubuntu 18.04. The specific fuzzing configuration was continuously experimented with, but typically the number of iterations per seed ($i$) was set to $50 - 500$, the number of incremental mutations ($wl$) to $10 - 100$ and the solver timeout to $3 - 30$ seconds. We tested multiple solver configurations of daily rebuilt development versions of Z3 and CVC5. As seed files we used a publicly available, labelled set by the YinYang project[7]. We reduced all found bugs with `ddsmt` [9].

### 6.2 Statistical properties of the generated mutants

In this section we present a few statistics of the mutant set generated during a 24 hour long example run under the same experimental conditions as described above. The fuzzer configuration used is typical of what was used during the whole campaign: 300 iterations per seed, 50 incremental iterations, a solver timeout of 10 seconds and 50 concurrent such fuzzer instances. This tested 6043 seeds, generated a total of $1,331,576$ mutants[8] of which 99.6% were unique. These mutants were tested on 8 solver configurations (four of Z3 and CVC5 each) resulting in $10'658'346$ solver calls of which:

- $215,011$ timed out (0.02%).
- $127,920$ returned unknown (0.01%).

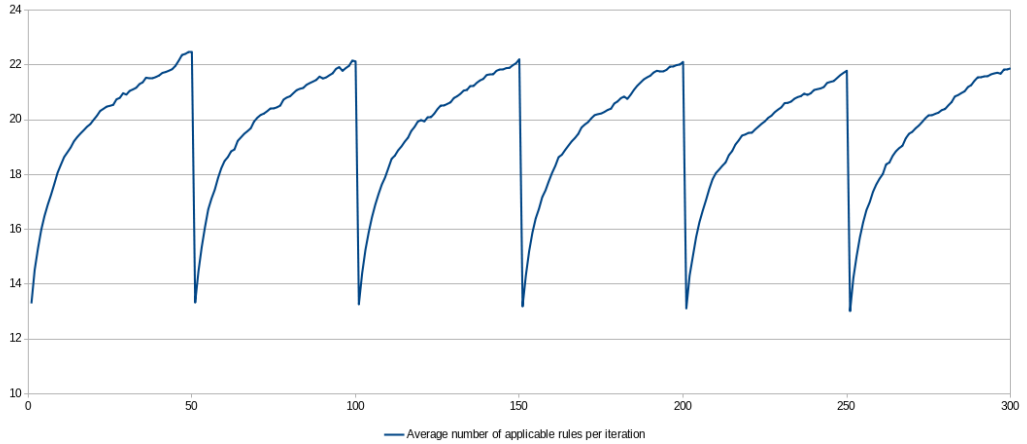
*6.2.1 Rule applicability.* Referring back to our considerations of rule applicability during design, Fig. 6 shows the number of applicable rules per iteration on average over all seeds. The emerging pattern is very clear: The combination of seeds and rule we chose yields roughly 13 mutation rules that can be applied to a seed. Interestingly, this number consistently increases during incremental mutation before dropping back to 13 when the seed is reset (each 50 iterations in this configuration). This is consistent with our reservations about purely incremental mutation, as the increase is

[7] https://github.com/testsmt/semantic-fusion-seeds

[8] The number of mutants is smaller than the expected $300 \cdot 6043 = 1'812'900$ because the experiment was terminated without letting all last instances run to the end of their 300 iterations.

most likely due to increased size of the formulas. It also suggests that we have likely avoided the problem of trivial rule cycles, explained in Section 3.2.1. Since the mutation rule is always chosen uniformly at random from all applicable ones, increasing the number of applicable rules decreases the probability of the fuzzer choosing the same few rules in succession.

Fig. 6. The average number of applicable rules per iteration from a test run with 50 incremental mutations.




*6.2.2 Rule diversity.* In Fig. 7, another perspective on rule applicability is taken by inspecting how often individual rules apply. The diagram depicts the number of applications per rule as percentage of the total number of mutations performed in this experiment. The distribution shows a slight skew towards a few rules, but only the top 5 rules take more than 5% each:

(1) `DROPCONJ`: 6.6%
(2) `ADDDISJ`: 6.0%
(3) `NUMRELSHIFTBALANCED`: 5.9%
(4) `NUMRELSHIFTSKEWED`: 5.7%
(5) `EQUAL[STRTOINT]`: 5.4%

We interpret this as further evidence that our fuzzer avoids running through the same mutation cycles as this would manifest itself in a drastic skew towards the rules involved in such cycles. Instead, there are simple explanations for these rules to come out on top: `DROPCONJ` weakens the prevalent operator and without further constraints. `ADDDISJ` is applicable to every seed and mutant as it transforms an arbitrary boolean term `T` into `(or T S)`, where `S` is chosen randomly (c.f. 4.2). Rules number 3 and 4 are slightly concerning, as this is exactly the type of rule used in our illustration of trivial mutation cycles. But again, these rules should be expected to come out on top as their target syntax (=, <=, <, etc.) is widely used in seeds. Moreover, both rules are applicable and implemented as weakening and strengthening to any of those operators. The rule `EQUAL[STRTOINT]` is a rewrite rule among equal terms which can be applied in both directions. Thus it applies to its own result which gives it high applicability.

## 6.3 Results

*6.3.1 Soundness and crash bugs.* We first look at the effectiveness of our fuzzing approach in a conventional sense, i.e. when testing for unsoundness and crash bugs. We found only three such bugs during our six month fuzzing campaign, listed in Tab. 2. One of the bugs is shown in List. 5,

Fig. 7. The ratio of how many times each of our mutation rules used in our evaluation run has been applied. The top five rules and their percentages are listed in 6.2.2.

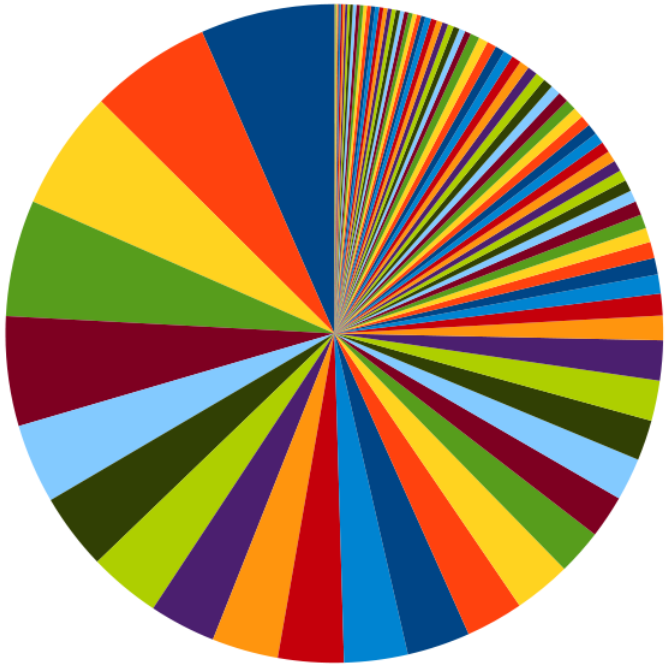


Table 2. The three conventional bugs found by our fuzzing campaign from February 2021 to July 2021 when testing for soundness and crashes. The formulas can be found in the corresponding reports in Appendix B.

| solver | options | type | status |
|---|---|---|---|
| Z3 | `proof=true` | crash | fixed |
| Z3 | `model_validate=true` | invalid model | fixed |
| CVC5 | `-q --strings-exp --full-saturate-quant` | crash | confirmed/open |

where Z3 correctly returned `sat` but generated an invalid model for the formula. Most notably, no unsoundness bugs were found at all. To put this into perspective, searching the issue trackers we find 70+ soundness bug reports submitted during the time of our campaign. This number includes duplicates and bugs in solver configurations not tested by us. Still, we do not think that our approach is inherently incapable of finding unsoundness bugs as our fuzzer readily finds them in older solver versions and experimental configurations. In 6.5 we illustrate some of the potential difficulties involved.

```
1 (declare-const ar_7 String)
2 (assert (str.in_re ar_7 (re.++ (str.to_re ar_7) ((_ re.loop 1) (re.range "" ""))))
      )
```

Table 3. The number of reports from our incompleteness testing campaign.

| solver | type | reported | confirmed | fixed | rejected |
|---|---|---|---|---|---|
| CVC5 | 1 | 9 | 9 | 1 | 0 |
| Z3 | 1 | 21 | 10 | 6 | 4 |
| total | 1 | 30 | 19 | 7 | 4 |
| CVC5 | 2 | 2 | 2 | 0 | 0 |
| Z3 | 2 | 0 | 0 | 0 | 0 |
| total | 2 | 2 | 2 | 0 | 0 |

```
(check-sat)
```

Listing 5. A formula error using regular expressions which we reported on the Z3 issue tracker in June 2021 because it triggered an invalid model error . It was confirmed after three days and fixed after one month. https://github.com/Z3Prover/z3/issues/5343.

*6.3.2 Incompleteness reports.* Combining Z3 and CVC5, we reported 35 cases of which 9 were fixed and 4 rejected. The feedback by the developers from both Z3 and CVC5 was generally positive, they discussed and fixed several issues in both solvers. Some of our reports were false positives in the sense that we simply discovered an SMT feature not supported in reasoning engines by the solvers, for example `str.replace_all`[9] and similar string operations[10]. This is only a small drawback to the overall testing effort, as after one such report we simply filtered out the corresponding cases.

### 6.4 Developer feedback on incompleteness testing

The most important feedback is whether developers confirm and fix our bug reports which we evidence with two examples.

*6.4.1 Discovering a lost* `ite` *rewrite in CVC5.* Although not previously exposed to, developers were generally open to our reports of completeness regressions. List. 6 contains one of our reported cases and illustrates the potential benefits of testing completeness boundaries of solvers.

```
(declare-const T Bool)
(declare-const v String)
(assert (ite T T true))
(assert (or T (and (str.prefixof v "") (exists ((x Int)) (= "t" (str.substr v 0 x)
    )))))
(check-sat)
```

Listing 6. The term `(ite T T true)` is equivalent to `true`, but CVC5 at the time was unable to perform this simplification and returned unknown. The report led to the discovery of unwanted side effects produced by a newer set of rewrite rules over older ones. (https://github.com/cvc5/cvc5/issues/6717)

The formula is satisfiable by setting `T` to `true`. Similarly to List. 4, a manual rewrite step replacing `(ite T T true)` with `true` is enough for the solver to correctly decide the formula. Moreover the previous releases of the solver correctly decided this case, i.e. it is a type 1 incompleteness. Through our report, the developers discovered that a set of newer rewrite rules taking precedence over older ones accidentally prevented this simple term to be fully rewritten. They promptly adjusted the precedences and successfully closed the incompleteness[11].

[9] https://github.com/Z3Prover/z3/issues/5344#issuecomment-861695688
[10] https://github.com/cvc5/cvc5/issues/6742#issuecomment-860225800
[11] https://github.com/cvc5/cvc5/pull/6723

*6.4.2 Discovering a lost* `str.replace` *rewrite in Z3.* A similar case occured with Z3 when we found the formula shown in List. 7. It contains the term `(str.replace "" va "")` which can be rewritten to `""`. Z3 returned unknown for the shown formula, but correctly decided it as `sat` after we manually performed this rewrite step. To fix the issue, the developers added this exact rewrite step to the string rewriter component of the solver[12].

```
(declare-const x Int)
(declare-const T Int)
(declare-const va String)
(assert (distinct (str.from_int T) (str.replace va (str.replace "" va "") (str.
    from_int (- x)))))
(check-sat)
```

Listing 7. At the time of reporting, Z3 returned unknown for this formula. The term `(str.replace "" va "")` is equal to `""` irrespective of the value of `va`, since `(str.replace s t t')` replaces all occurrences of `t` in `s` with `t'`[11]. Manually performing this rewrite thus creates an equivalent formula, for which Z3 then returned `sat`. (https://github.com/Z3Prover/z3/issues/5399)

### 6.5 Example: Reconstructing a previously known soundness bug

As an illustration of the constraints involved in our approach, we look at a soundness bug found by another fuzzer and try to recreate it using weakening and strengthening rules. This could hint at reasons for the ineffectiveness of our approach as a soundness testing technique. To this end, consider formula $\varphi$ given in List. 8. We are asking if and how our fuzzer could have found this bug. Since this is an unsatisfiable formula, our fuzzer can only produce it via a strengthening step from an unsatisfiable $\varphi'$, i.e. $\varphi \implies \varphi'$ should hold by one of our rules. More precisely, a strengthening step is either a strengthening of a positive subformula or a weakening of a negative subformula. Additionally, the mutation should reveal the bug which means that Z3 should return the correct result `unsat` for $\varphi'$. Does such a $\varphi'$ exist for this specific case? Here are three of our rules which potentially apply:

- Rule (25) from 4.1.2: This would mean that = is a strengthening of $\leq$. But applying the rule in the reverse direction, replacing = with $\leq$ in List. 8, also results in `sat` which means that this rule does not reveal the bug.
- Rule (23) from 4.1.2: As the rule above, this would produce the '=' as a strengthening of '>='. However, the formula with '>=' is satisfiable which means that our fuzzer only weakens but never strengthens it and thus cannot discover the bug this way.
- Rule (11) from 4.1.1: For this rule, there are infinitely many possibilites and we only inspect two. The rule instantiated with the function `abs` still has the bug. With a fresh uninterpreted function `f` the formula becomes satisfiable.

```
(declare-const x String)
(assert (= 0 (str.to_int (str.++ (str.replace "" x "")
        (str.replace "" x (str.substr x 1 0))))))
(check-sat)
```

Listing 8. A string formula that was reported on the Z3 GitHub issue tracker because it returned `sat` but the formula is unsatisfiable (https://github.com/Z3Prover/z3/issues/5096).

While this illustrates the constraints our approach has to work under, there are two caveats to be aware of here. First, the formula in List. 8 is a highly reduced version of the issue. Typically fuzz testing produces larger mutants which are then reduced to a minimal reproducing example, so

[12]https://github.com/Z3Prover/z3/commit/0f8d2d1d51b814edda853dacd8a7b88a45fad33a

in practice the fuzzer does not need to find this small formula in order to find the bug. In larger formulas there are likely more applicable strengthening and weakening rules, so the analysis above is a bit too strict. Second, we only considered single step mutations whereas the fuzzer was usually configured to perform 10-50 incremental mutations (See 3.2).

## 7 CONCLUSION

We presented the weakening and strengthening approach for fuzz-testing SMT solvers. With our implementation, we tested Z3 and CVC5 for unsoundness, crashes and incompleteness over the course of five months, resulting in a total of 35 reports of which 9 were fixed. Of those reports, two are crashes, one is an invalid model bug and none are unsoundness bugs. This could be due to a saturation process after continuous efforts in fuzz-testing SMT solvers in recent years, but we cannot exclude a fundamental ineffectiveness of our approach. The rest of the reports are incompletenesses and were well received by the developers, leading to the discovery of regressions and missing rewrite steps in both solvers. This evidence suggests that there is potential for incompleteness testing to improve the quality of SMT solvers in ways that conventional fuzz-testing cannot.

## ACKNOWLEDGMENTS

To Prof. Zhendong Su and Dominik Winterer of the AST Lab at ETH Zürich, for their continued support and lively discussions of SMT formulas. To researchers and developers tackling the challenging problem of SMT solving and interacting with our reports.

## REFERENCES

[1] 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.

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[3] Aaron R Bradley and Zohar Manna. 2007. *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media.

[4] Alexandra Bugariu and Peter Müller. 2020. Automatically testing string solvers. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1459–1470.

[5] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2020).

[6] The International SMT Competition. 2021 (accessed July 20, 2021). SMT-COMP.

[7] 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.

[8] 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*. 701–712.

[9] Aina Niemetz and Armin Biere. 2013. ddSMT: a delta debugger for the SMT-LIB v2 format. In *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT*. 8–9.

[10] Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark Barrett, and Cesare Tinelli. 2019. Syntax-guided rewrite rule enumeration for SMT solvers. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 279–297.

[11] Cesare Tinelli, Clark Barnett, and Pascal Fontaine. 2020 (accessed July 19, 2021). SMT-LIB Standard String Theory. http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml.

[12] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 193 (Nov. 2020), 25 pages. https://doi.org/10.1145/3428261

[13] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 718–730.

[14] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Fuzzing SMT solvers via two-dimensional input space exploration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 322–335.

[15] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Skeletal Approximation Enumeration for SMT Solver Testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.*

## A LIST OF FIXED INCOMPLETENESS REPORTS

- https://github.com/Z3Prover/z3/issues/5381
- https://github.com/Z3Prover/z3/issues/5376
- https://github.com/Z3Prover/z3/issues/5349
- https://github.com/Z3Prover/z3/issues/5340
- https://github.com/Z3Prover/z3/issues/5338
- https://github.com/Z3Prover/z3/issues/5399
- https://github.com/cvc5/cvc5/issues/6717

## B LIST OF SOUNDNESS AND CRASH REPORTS

- https://github.com/Z3Prover/z3/issues/5197
- https://github.com/Z3Prover/z3/issues/5343
- https://github.com/cvc5/cvc5/issues/6750

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

## Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

---

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

Fuzz-testing SMT Solvers with Formula Weakening and Strengthening

**Authored by** (in block letters):

*For papers written by groups the names of all authors are required.*

| **Name(s):** | **First name(s):** |
|---|---|
| Bringolf | Mauro |
| | |
| | |
| | |
| | |

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

| **Place, date** | **Signature(s)** |
|---|---|
| 17.7.2021 Zürich |  |
| | |
| | |
| | |
| | |

*For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*