Valuating SMT Solvers via Semantic Fusion

Dominik Winterer*
Department of Computer Science
ETH Zurich, Switzerland
dominik.winterer@inf.ethz.ch

Chengyu Zhang*
Software Engineering Institute
East China Normal University, China
dale.chengyu.zhang@gmail.com

Zhendong Su
Department of Computer Science
ETH Zurich, Switzerland
zhendong.su@inf.ethz.ch

Abstract
We introduce Semantic Fusion, a general, effective methodology for validating Satisfiability Modulo Theory (SMT) solvers. Our key idea is to fuse two existing equisatisfiable (i.e., both satisfiable or unsatisfiable) formulas into a new formula that combines the structures of its ancestors in a novel manner and preserves the satisfiability by construction. This fused formula is then used for validating SMT solvers.

We realized Semantic Fusion as YinYang, a practical SMT solver testing tool. During four months of extensive testing, YinYang has found 45 confirmed, unique bugs in the default arithmetic and string solvers of Z3 and CVC4, the two state-of-the-art SMT solvers. Among these, 41 have already been fixed by the developers. The majority (29/45) of these bugs expose critical soundness issues. Our bug reports and testing effort have been well-appreciated by SMT solver developers.

CCS Concepts: • Software and its engineering → Formal methods; Correctness.

Keywords: Semantic fusion, SMT solvers, Fuzz testing

ACM Reference Format:

1 Introduction
Satisfiability Modulo Theory (SMT) solvers check the satisfiability of first-order logic formulas with functions from different theories, such as the booleans, linear and nonlinear arithmetic, unicode strings, etc. They are important tools for many programming languages advances and applications, e.g., symbolic execution [12, 20], program synthesis [31], solver-aided programming [33], and program verification [16, 17]. SMT solvers’ satisfiability decisions are critical, and incorrect decisions (i.e., soundness bugs) can invalidate the results of their client applications.

Z3 [15] and CVC4 [2] are two state-of-the-art SMT solvers that have been consistently developed for more than ten years. Researchers and practitioners value their extensive theory support and trust the SMT solvers’ results. This is justified since soundness bugs in Z3 and CVC4 are rare, and both solvers have extensive regression test suites. However, like any complex software systems, SMT solvers can still have bugs. In fact, almost all critical SMT solver bugs in Z3 and CVC4 have been uncovered directly by their client applications. Such bugs frustrate application developers, and can be catastrophic in safety-critical domains. Besides regression testing, fuzzing has been used to validate SMT solvers.

In 2009, Brummayer and Biere [7] proposed a grammar-based fuzzer called FuzzSMT, which found several bugs in CVC3 (CVC4’s predecessor) and early versions of Z3. Within the last ten years, SMT solvers have greatly matured, and finding bugs in them has become more difficult. More recent efforts on testing SMT solvers by fuzzing [5, 9] have targeted the unicode string theory and found a few bugs in Z3’s string solvers. Yet none has targeted other SMT theories nor found bugs in recent versions of CVC4.

Semantic Fusion. This paper introduces Semantic Fusion, a general, effective approach to validating SMT solvers. Our key insight is to fuse two tests into a new test that combines the structures of its ancestors. We fuse two equisatisfiable formulas $\varphi_1$ and $\varphi_2$ (i.e., both $\varphi_1$ and $\varphi_2$ are either satisfiable or unsatisfiable) into an equisatisfiable formula $\varphi_{\text{fused}}$. Our approach consists of the following three main steps:

1. Formula Concatenation: Concatenate $\varphi_1$ and $\varphi_2$ by formula conjunction or disjunction;
2. Variable Fusion: Create fresh variables to connect the variable sets of $\varphi_1$ and $\varphi_2$ using fusion functions; and
3. Variable Inversion: Substitute some occurrences of the chosen variables in $\varphi_1$ and $\varphi_2$ by inversion functions.

Figure 1 illustrates Semantic Fusion on two satisfiable formulas $\varphi_1$ and $\varphi_2$. We first concatenate $\varphi_1$ and $\varphi_2$, and obtain $\varphi_{\text{concat}}$ as a result. Then, we introduce a fresh variable $z$ and a fusion function $f(x, y) = x + y$, and construct a relation $z = f(x, y)$, which induces two equations $x = z - y$ and

\[
\begin{align*}
\varphi_1 & = \neg a \\
\varphi_2 & = \neg b \\
\varphi_{\text{concat}} & = \varphi_1 \lor \varphi_2 \\
\varphi_{\text{fused}} & = \varphi_{\text{concat}} \\
& = \neg a \lor \neg b
\end{align*}
\]
We introduce Semantic Fusion, a novel, general, principled methodology for stress-testing SMT solvers; Based on the Semantic Fusion methodology, we design and develop the first highly effective tool, YinYang, for SMT solver validation — the tool is customizable and conveniently supports various SMT theories; We conduct a four-month extensive testing of Z3 and CVC4 using YinYang to demonstrate its effectiveness — we have found and reported a total of 57 bugs in 45 confirmed and 41 fixed in their default arithmetic and string solvers, the largest and most successful testing campaign against modern SMT solvers; and We present several in-depth evaluations to understand YinYang’s effectiveness in terms of improved code coverage and with respect to a survey of the historic bugs in the SMT solvers Z3 and CVC4.

### Main Contributions

- SAT fusion combines two satisfiable formulas into a satisfiable formula. SAT fusion can be described by the following steps: (1) Formula Conjunction, (2) Variable Fusion, and (3) Variable Inversion. Consider the formulas \( \varphi_1 \) and \( \varphi_2 \) in Figure 2. The SAT-LIB code represents the following formulas:

\[
\varphi_1 \equiv (x = -1) \land (w = (x = -1)) \land w
\]

\[
\varphi_2 \equiv (v = (y \neq -1)) \land (v \to \text{false}) \land (\neg v \to (y = -1))
\]

**Paper Organization.** The rest of the paper is structured as follows. Section 2 illustrates the high-level idea behind Semantic Fusion via two examples. Section 3 formalizes our Semantic Fusion approach and describes the implementation of YinYang. Next, we give details on our extensive evaluation (Section 4) and show sampled bugs to highlight the diverse types of bugs that YinYang can find (Section 4.3). Finally, we survey related work (Section 5) and conclude (Section 6).
Formula $\phi_1$ is satisfiable since assigning $x = -1$ and $w = true$ satisfies both conjuncts. Formula $\phi_2$ is also satisfiable since we can set $y$ to $-1$ and $v$ to $false$, which satisfies the formula. In the following, we describe steps 1-3 in detail.

**Step 1: Formula Conjunction:** We conjoin formula $\phi_1$ with formula $\phi_2$ and obtain $\phi_1 \land \phi_2$ as a result. In the SMT-LIB format, this conjunction can be carried out by simply merging the variable declaration and assert blocks.

**Step 2: Variable Fusion:** We introduce a fresh variable $z$ to fuse the integer variable pairs $x$ in $\phi_1$ and $y$ in $\phi_2$. We define a fusion function: $f(x, y) = x \cdot y$ and construct an equation $z = f(x, y)$. The choice of the fusion function $f$ is determined by the type of the fused variables (cf. Section 3). We fuse the occurrences of variables $x$ and $y$.

**Step 3: Variable Inversion:** We dissolve the equation $z = f(x, y)$ to $r_s(y, z) = z \ div y$ and $r_p(x, z) = z \ div x$, where $r_s$ and $r_p$ are called inversion functions and $\ div$ denotes integer division. The purpose of the inversion functions is to recover the original values of $x$ and $y$. The inversion function $r_s(y, z)$, for example, recovers $x$ by a term that only depends on $y$ and $z$. We then randomly replace free occurrences of $x$ by $r_s(y, z)$ and free occurrences of $y$ by $r_p(x, z)$. The formula $\phi_{sat}$ is by construction satisfiable. The SMT-LIB code of $\phi_{sat}$ is shown in Figure 3.

Why is $\phi_{sat}$ satisfiable? Intuitively, because we can construct a model for $\phi_{sat}$ from models for $\phi_1$ and $\phi_2$. Let $M_1$ be a model for $\phi_1$ and $M_2$ be a model for $\phi_2$. We construct $M$ for $\phi_{sat}$ with $M = M_1 \cup M_2 \cup \{ z \mapsto M_1(x) \cdot M_2(y) \}$ (see Section 3 for details). Formula $\phi_{sat}$ in Figure 3 is a real case. It triggered a soundness bug in CVC4, which made CVC4 incorrectly report unsat on $\phi_{sat}$. We reported this issue to the GitHub CVC4’s issue tracker. As per the developers, this was a regression introduced by recent code changes, and they promptly fixed the bug.

### 2.2 UNSAT Fusion

UNSAT fusion combines two unsatisfiable formulas into an unsatisfiable formula. We describe the idea behind UNSAT fusion in four steps: (1) Formula Disjunction, (2) Variable Fusion, (3) Variable Inversion, and (4) Adding Fusion Constraints. While steps (1), (2) and (3) are similar as those in SAT fusion, UNSAT fusion needs an additional fourth step to ensure the unsatisfiability of the fused formula.

Consider the formulas $\phi_3$ and $\phi_4$ in Figure 4:

$$\phi_3 = ((1.0 + x) + 6.0) \neq 7.0 + x$$

$$\phi_4 = (0 < y < v \leq w) \land (w/v < 0)$$

Formula $\phi_3$ is trivially unsatisfiable. Formula $\phi_4$ is unsatisfiable, since the left part of the conjunction requires both $w$ and $v$ to be non-negative but the right part requires $w$ and $v$ to be of opposite signs. First, we disjoin the two formulas. We then again choose a pair of free variables in each formula, e.g., variable $x$ in $\phi_1$ and variable $y$ in $\phi_2$, and introduce a fresh variable $z$ and fusion function $f(x, y) = x \cdot y$ with $z = f(x, y)$. We dissolve the equation $z = f(x, y)$ to inversion functions $r_s(y, z) = z/y$ and $r_p(x, z) = z/x$, and randomly substitute the first occurrence of $x$ by $r_s(y, z)$ and both occurrences of $y$ by $r_p(x, z)$.

**Step 4: Add Fusion Constraints:** We add $z = f(x, y), x = r_s(y, z)$ and $y = r_p(x, z)$ to the fused formula. We call them fusion constraints. Since random substitutions may

---

**Figure 3.** Fused formula $\phi_{sat}$ in the SMT-LIB format. It triggered a soundness bug in CVC4. [GitHub Link](https://github.com/CVC4/CVC4/issues/3413)

**Figure 4.** Formulas $\phi_3$ and $\phi_4$ in the SMT-LIB format. Shaded: variables to be replaced by inversion function terms.

**Figure 5.** Fused formula $\phi_{unsat}$ of $\phi_3$ and $\phi_4$ that triggered a soundness bug in Z3. [GitHub Link](https://github.com/Z3Prover/z3/issues/2391)
render the fused formula satisfiable, we need the fusion constraints to ensure that \( r_x \) and \( r_y \) recover \( x \) and \( y \) (see Section 3 details). The SMT-LIB code of the resulting fused formula is shown in Figure 5.

The fused formula \( \phi_{\text{unsat}} \) in Figure 5 has triggered a soundness bug in Z3, i.e., Z3 reports sat on \( \phi_{\text{unsat}} \), which is incorrect since the formula is unsatisfiable by construction. This bug is only triggered by the fused formula; it cannot be triggered by either of the seed formulas nor by the disjunction/conjunction of the two seed formulas \( \phi_3 \) and \( \phi_4 \).

3 Approach

This section presents Semantic Fusion and how we apply it to stress-testing SMT solvers. We propose two approaches, SAT Fusion and UNSAT Fusion, and prove their correctness. We describe our tool YinYang, a practical implementation of SAT Fusion and UNSAT Fusion.

3.1 Definitions

We consider first-order logic formulas of the satisfiability modulo theories (SMT). For such a formula \( \phi \), we denote the set of free variables by \( \text{vars}(\phi) \). A substitution of a variable \( x \in \text{vars}(\phi) \) by an expression \( e \) is denoted by \( \phi[e/x] \). A model \( M \) for \( \phi \) is a function that maps all free variables \( x_1, \ldots, x_n \in \text{vars}(\phi) \) to values in their respective domains such that \( \phi[M(x_1)/x_1, \ldots, M(x_n)/x_n] \) simplifies to true. The model count of a formula \( \phi \) is \( C(\phi) = |\{ M | M \models \phi \}| \). Formula \( \phi \) is satisfiable if \( C(\phi) \geq 1 \) and unsatisfiable otherwise. \( \phi[e/x]_R \) denotes the formula where some of the occurrences of \( x \) (possibly none) in \( \phi \) are replaced by \( e \). It holds \( C(\phi[e/x]) \leq C(\phi[e/x]_R) \).

3.2 Semantic Fusion

The insight of Semantic Fusion is to combine two seed tests into a new test that fuses the structures of its ancestors. Applying this to two formulas \( \phi_1 \) and \( \phi_2 \) of same satisfiability, we fabricate a fused formula \( \phi_{\text{fused}} \) that is equisatisfiable.

**Definition 1** (Fusion function). Let \( \phi_1, \phi_2 \) be formulas, \( x \in \text{vars}(\phi_1) \), and \( y \in \text{vars}(\phi_2) \). Let \( z \) be a fresh variable \( z \not\in \text{vars}(\phi_1) \cup \text{vars}(\phi_2) \). We define

\[
z := f(x, y)
\]

Function \( f \) is called a Fusion function.

Having defined fusion function, we next invent inversion functions to recover the original values for \( x \) and \( y \).

**Definition 2** (Inversion function). Let \( \phi_1 \) and \( \phi_2 \) be formulas and \( f \) be a fusion function. For \( x \in \text{vars}(\phi_1) \), \( y \in \text{vars}(\phi_2) \) and \( z = f(x, y) \), we define

\[
x = r_x(y, z) \quad y = r_y(x, z)
\]

functions \( r_x \) and \( r_y \) are called Inversion functions.

As an example, consider the fusion function \( f(x, y) = x + y \). The corresponding inversion functions for \( x \) and \( y \) are: \( r_x(y, z) = z - y \) and \( r_y(x, z) = z - x \). We next present a proposition that shows how we fuse two satisfiable formulas into an equisatisfiable formula.

**Proposition 1** (SAT fusion). Let \( \phi_1, \phi_2 \) be satisfiable formulas with \( \text{vars}(\phi_1) \cap \text{vars}(\phi_2) = \emptyset \). Let further \( x \in \text{vars}(\phi_1) \), \( y \in \text{vars}(\phi_2) \) be variables. Then, the formula

\[
\phi_{\text{sat}} = \phi_1[r_x(y, z)/x]_R \land \phi_2[r_y(x, z)/y]_R
\]

is satisfiable.

*Proof.* Let \( M_1 \) and \( M_2 \) be models for \( \phi_1 \) and \( \phi_2 \), respectively. We construct a model \( M \) for \( \phi_{\text{sat}} \) as follows:

\[
M(v) = M_1(v), \quad \text{for } v \in \text{vars}(\phi_1)
\]

\[
M(v) = M_2(v), \quad \text{for } v \in \text{vars}(\phi_2)
\]

\[
M(z) = f(M_1(x), M_2(y))
\]

Since \( x = r_x(y, z) \) by Definition 2, \( M(x) = M_2(y, z) \). Thus, \( M(\phi_1[r_x(y, z)/x]_R) = M_1(\phi_1) \) via structural induction. By \( M \)'s construction, \( M(\phi_1) = M_1(\phi_1) \), thus \( M \models \phi_1[r_x(y, z)/x]_R \). Similarly, \( M \models \phi_2[r_y(x, z)/y]_R \), and hence \( M \models \phi_{\text{sat}} \). \( \square \)

Proposition 1 enables us to fuse two satisfiable formulas and obtain a satisfiable formula as a result. We would also like to fuse unsatisfiable formulas into an unsatisfiable formula. However, we cannot simply fuse two unsatisfiable formulas using Proposition 1 as the following counterexample shows. Consider the unsatisfiable formulas \( \phi_1 = x > 0 \land x < 0 \), \( \phi_2 = y \neq y \), and the fusion function \( z = x + y \). If we replace the shaded occurrence of \( y \) by \( y - z \) and \( y \) by \( x - z \), we get the following formula: \( x > 0 \land (z - y < 0) \land (z - x \neq y) \). This is a satisfiable formula, e.g., any assignments for \( x \) and \( z \) that satisfy \( x > 0 \) and \( y > z \) realize a model. The problem here is that we can freely choose \( z \) that does not necessarily preserve \( z = f(x, y) \). To prevent this, we add the constraint \( z = f(x, y) \) to the formula. For fusing unsatisfiable formulas, we disjoint the formulas, since this is likely to increase the effort of SMT solvers to prove the formula unsatisfiable.

**Proposition 2** (UNSAT fusion). Let \( \phi_1, \phi_2 \) be unsatisfiable formulas with \( \text{vars}(\phi_1) \cap \text{vars}(\phi_2) = \emptyset \). Let further \( x \in \text{vars}(\phi_1) \), \( y \in \text{vars}(\phi_2) \) be variables. Then, the formula

\[
\phi_{\text{unsat}} = (\phi_1[r_x(y, z)/x]_R \lor \phi_2[r_y(x, z)/y]_R) \land z = f(x, y)
\]

is unsatisfiable.

*Proof.* Assume the contrary, i.e., \( \phi_{\text{unsat}} \) were satisfiable. Then either (or both) of the following would be satisfiable:

\[
\phi_1[r_x(y, z)/x]_R \land z = f(x, y)
\]

\[
\phi_2[r_y(x, z)/y]_R \land z = f(x, y)
\]

Say \( \phi_1[r_x(y, z)/x]_R \land z = f(x, y) \) were satisfiable. The formula \( \phi_1[r_x(y, z)/x]_R \land z = f(x, y) \) is equivalent to the formula \( \phi_1[r_x(y, z)/x]_R[f(x, y)/z] \land z = f(x, y) \), which, by Definition 2, is equivalent to \( \phi_1 \land z = f(x, y) \). This contradicts
We now give exemplary fusion and inversion functions (see Section 3.3).

<table>
<thead>
<tr>
<th>Type</th>
<th>Fusion Function</th>
<th>Variable Inversion Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>$x + y$</td>
<td>$z - y$</td>
</tr>
<tr>
<td></td>
<td>$x + c + y$</td>
<td>$z - c - y$</td>
</tr>
<tr>
<td></td>
<td>$x + y$</td>
<td>$z \div y$</td>
</tr>
<tr>
<td></td>
<td>$c_1 \times x + c_2 \times y + c_3$</td>
<td>$(z - c_2 \times y - c_3) \div c_1$</td>
</tr>
</tbody>
</table>

| Real   | $x + y$         | $z - y$                      |
|        | $x + c + y$     | $z - c - y$                  |
|        | $x + y$         | $z$                          |
|        | $c_1 \times x + c_2 \times y + c_3$ | $(z - c_2 \times y - c_3)/c_1$ |

| String | $x \ str++ y$   | str.substr $z \ 0$ (str.len $x$) |
|        | $x \ str++ y$   | str.substr $z \ 1$ (str.len $x$) |
|        | $x \ str++ c \ str++ y$ | str.replace $z \ "c"$ |

Figure 6. Variable fusion functions with their corresponding variable inversion functions categorized by types Int, Real and String. The coefficients $c_1, \ldots, c_3$ are randomly chosen, and $\div$ denotes integer division.

In addition, we can insert a random string $c$ into $x \ str++ y$ by $x \ str++ c \ str++ y$ to make the fusion function more complex, and then retrieve $y$ by replacing $x$ and $c$ with "" sequentially. We emphasize that Semantic Fusion is not restricted to these fusion and inversion functions of Figure 6. A richer set of fusion and inversion functions can be designed based on the generic Definitions 1 and 2.

### 3.4 YinYang

Based on Semantic Fusion, we have designed and engineered the bug detection tool YinYang to stress-test SMT solvers.

**Algorithm.** Algorithm 1 presents a parameterized algorithm of YinYang. The main procedure takes the oracle of the seed formulas $o \in \{ \text{sat, unsat} \}$, SMT solver under test $S$, and a set of seed formulas $\Phi_o$ as input. Each of the seeds in $\Phi_o$ has the same satisfiability as the oracle $o$ (either all sat or all unsat). The sets of incorrects and crashes are sets for collecting soundness and crash bugs, respectively, and are both initialized to the empty set. While loop body is executed until a termination criterion is met, e.g., a timeout or an interrupt by the user (Line 3). We first randomly choose two formulas $\phi_1, \phi_2$ from $\Phi_o$, and pass them to the fusion function together with the oracle $o$. The fusion function returns the fused formula $\phi_{\text{fused}}$ (Line 6). Then, we check whether the SMT solver $S$ has crashed on solving $\phi_{\text{fused}}$. If so, we have found a crash bug and will add $\phi_{\text{fused}}$ to crashes. Otherwise, if $S$ does not crash, we check whether $S(\phi_{\text{fused}})$ is inconsistent with the oracle $o$ (Line 9). If so, we have observed a soundness issue and will add $\phi_{\text{fused}}$ to the set incorrects.

Algorithm 2 presents the implementation of the fuse function. It takes two seed formulas $\phi_1$ and $\phi_2$ as input and retrieves the sets of their free variables $\text{vars}(\phi_1)$ and $\text{vars}(\phi_2)$, respectively. Then, we create random triplets $T$, for $(x, y, z) \in T$ where $x \in \text{vars}(\phi_1)$ and $y \in \text{vars}(\phi_2)$, and $z$ the assumption, i.e., the unsatisfiability of $\phi_1$. The case for

$$\phi_2[r_{\phi}(x, z)/y] \land z = f(x, y)$$

is symmetric. □

Proposition 2 enables us to fuse two unsatisfiable formulas into an unsatisfiable formula and complements Proposition 1. We could also apply fusion to mixed formula pairs, i.e., when $\phi_1$ is satisfiable and $\phi_2$ is unsatisfiable. We can use $\phi_1[r_{\phi}(y, z)/y] \lor \phi_2[r_{\phi}(x, z)/y]$ for a satisfiable fused formula and $\phi_1[r_{\phi}(x, z)/y] \land \phi_2[r_{\phi}(y, z)/y] \land z = g(x, y)$ for an unsatisfiable fused formula.

### 3.3 Fusion and Inversion Functions

We now give exemplary fusion and inversion functions (see Figure 6) and explain the intuitions behind them. Let us consider the Int and Real categories. The first two fusion and inversion functions in these categories are based on addition/subtraction and multiplication/division. When division and multiplication of variables are used as function and inversion functions, a formula in linear logic might become non-linear. This is because we replace free variables occurrences by variable inversion functions that include the division operator. Another inversion function for real and integer arithmetic is $c_1 \times x + c_2 \times y + c_3$. The intuition behind $c_1 \times x + c_2 \times y + c_3$ is to synthesize arbitrary polynomial combinations of the variables $x$ and $y$ since $c_1, \ldots, c_3$ are random coefficients. Let us consider Strings next. In the first row of the String category, we define $z$ as the concatenation of the two strings $x$ and $y$. Say $x = "\text{bar}"$ and $y = "\text{foo}"$, then $z = x \ str++ y = "\text{fooobar}"$. We retrieve $x$ by the substring of $z$ from 0 to $|x|$, for $y$ the substring from $|y|$ to the end of $z$. Another way to retrieve $y$ is to use the replace function instead of substring. The expression $str.replace z "c"$ denotes the replacement of the first occurrence of $x$ in $z$ by the empty string "", which results in "bar".
Algorithm 1: YinYang’s main process

```plaintext
1 Procedure YinYang \(\langle o, S, \Phi_o\rangle\):
2    incorrects ← 0, crashes ← 0
3    while no termination criterion met do
4        \(\varphi_1\) ← random.choice(\(\Phi_o\))
5        \(\varphi_2\) ← random.choice(\(\Phi_o\))
6        \(\varphi_{\text{fused}}\) ← fuse(\(o, \varphi_1, \varphi_2\))
7        if \(S(\varphi_{\text{fused}}) = \text{crash}\) then
8            crashes ← crashes ∪ \{\varphi_{\text{fused}}\}
9        else if \(S(\varphi_{\text{fused}}) \neq o\) then
10           incorrects ← incorrects ∪ \{\varphi_{\text{fused}}\}
11
return \(\varphi_1\ ∧ \varphi_2\)
```

Algorithm 2: Semantic Fusion on two SMT formulas

```plaintext
Function fuse(\(o, \varphi_1, \varphi_2\)):
1        vars(\(\varphi_1\)) ← free_variables(\(\varphi_1\))
2        vars(\(\varphi_2\)) ← free_variables(\(\varphi_2\))
3        \(T\) ← random_map(\(\varphi_1\), vars(\(\varphi_2\)))
4        \(\varphi'_1, \varphi'_2\) ← variable_fusion(\(T, \varphi_1, \varphi_2\))
5        if \(o = \text{sat}\) then
6           return \(\varphi'_1 \land \varphi'_2\)
7        else
8           \(\varphi'\) ← \(\varphi'_1 \lor \varphi'_2\)
9           foreach \((x, y, z) \in T\) do
10              \(\varphi'\) ← \(\varphi' \land z = f(x, y)\)
11           return \(\varphi'\)
```

```plaintext
Function variable_fusion(\(T, \varphi_1, \varphi_2\)):
14       \(\varphi'_1, \varphi'_2\) ← \(\varphi_1, \varphi_2\)
15       foreach \((x, y, z) \in T\) do
16           \(\varphi'_1\) ← \(\varphi'_1[r_x(y, z) \times]_R\)
17           \(\varphi'_2\) ← \(\varphi'_2[r_y(x, z) / y]_R\)
18       return \(\varphi'_1, \varphi'_2\)
```

is the fresh variable. In the variable_fusion function, we substitute randomly chosen occurrences of \(x\) in \(\varphi_1\) and \(y\) in \(\varphi_2\) by the inversion function terms \(r_x(y, z)\) and \(r_y(x, z)\) from Table 6. If oracle \(o\) is \(\text{sat}\), we perform SAT fusion (Proposition 1) and return the conjunction of \(\varphi'_1\) and \(\varphi'_2\) directly (Line 7). If oracle \(o\) is \(\text{unsat}\), we perform UNSAT fusion, i.e., we disjoin \(\varphi'_1\) and \(\varphi'_2\), and add a fusion constraint for each triplet \((x, y, z) \in T\) (Lines 9-11) and return the result.

In principle, YinYang guarantees the absence of false positives, given that the seed formulas \(\Phi_o\) are correctly labeled. In practice, the solvers may report unknown, which could be either seen as a crash or ignored.

**Implementation of YinYang.** We implemented YinYang in a total of 1,032 lines of Python 3.7 code. YinYang is able to run in multiple-threaded mode, which significantly increases its throughput. Users can customize the command-line interface of YinYang for customized SMT solvers and/or features. YinYang accepts SMT solver binaries as test targets and obtains the solving results from the stdout stream, which makes YinYang compatible with most SMT solvers. A lightweight SMT-LIB v2 parser is implemented for getting free variables and assertions. The formula concatenation and variable substitution are implemented by string operations, which makes YinYang compatible to most of the formulas without additional implementation. When there are multiple fusion/inversion functions choices for \(f, r_x\) and \(r_y\), YinYang makes a random choice.

### 4 Empirical Evaluation

This section presents the details of our extensive evaluation of YinYang, demonstrating the practical effectiveness of the Semantic Fusion methodology. Between June and October 2019, we ran YinYang to test the default arithmetic and string solvers of Z3 and CVC4. We chose Z3 and CVC4 since they (1) are popular and widely-used in academia and industry, (2) support a rich set of logics, and (3) adopt an open-source development model. During our testing period, we filed numerous bugs on their GitHub issue trackers. This section describes the outcome of our testing effort.

**Result Summary and Highlights.**

- **Many confirmed bugs:** In four months, YinYang found 45 unique bugs in Z3 and CVC4. Out of these, 41 were already fixed by the developers.
- **Many soundness bugs:** YinYang found 24 soundness bugs in Z3 and 5 in CVC4. These represent 16% of the reported Z3 soundness bugs of the last five years and 11% of the reported CVC4 soundness bugs of the last nine years. Some of the bugs affect multiple historical release versions.
- **Bugs in various logics:** YinYang found bugs in various logics, e.g., QF_NRA, QF_NIA, NRA, NIA, QF_S, and QF_SLIA. Most of the bugs in Z3 were found in NRA (15) and QF_S (15), while most of the bugs in CVC4 were found in QF_S (4).
We choose the following logics: LIA, LRA, NRA, QF_LIA, QF_LRA, QF_NRA, QF_NIA, QF_SLIA, and QF_S. L represents linear, QF represents quantifier-free, N represents non-linear, IA represents integer arithmetic, RA represents real arithmetic, QF represents quantifier-free, and S represents string logic.

Besides the SMT-LIB benchmarks, we also used the benchmarks from StringFuzz [19]. The formulas from the StringFuzz benchmarks are in the QF_S logic and in the SMT-LIB 2.6 language. They do not trigger any bugs in the latest versions of Z3 and CVC4. We preprocessed all formulas (from the SMT-LIB benchmarks and StringFuzz) with Z3 to sub-divide them into a satisfiable and an unsatisfiable set. We cross-checked with CVC4 to ensure the correctness of these ground truths. In total, we obtained 75,097 seed formulas, 46,760 of which are satisfiable and 28,337 are unsatisfiable.

**SMT Solvers.** We selected the SMT solvers Z3 and CVC4 for the evaluation of YinYang. We chose them because:

- Z3 and CVC4 are mature and widely-used in academia and industry.
- Z3 and CVC4 have state-of-the-art performance. Both regularly rank high in the annual SMT competitions [14].
- Z3 and CVC4 support most of the features and logics in the SMT-LIB standard, while the other SMT solvers only partially support the SMT-LIB standard.
- Z3 and CVC4 have open source issue trackers on GitHub, and their developers are active and responsive. This helps our testing effort as we can quickly get feedback on our bug reports, and filed bugs are fixed promptly.

For CVC4, we use its -strings-exp option to enable support for the string logic and default configuration for the other logics. For Z3, we use smt.string_solver=z3str3, its default configuration for string logic, and default configuration for the other logics. We compiled both solvers with assertions enabled.

**Bug Reduction.** When a bug is found, we need to reduce the fused formula to a small enough size for reporting. We use C-Reduce [29], a C code reduction tool, which also works for the SMT-LIB language. We implemented a pretty printer to help with the bug reduction process, i.e., when C-Reduce has converged to a still very large formula or hangs. The pretty-printer makes simple modifications to the AST of a formula, i.e., flattens nestings of the same operator, removes additions and multiplications with neutral elements and returns the modified formula in a human-readable format.

**4.2 Quantitative Evaluation**

We guide our quantitative evaluation by four consecutive research questions.

**RQ1: How many bugs can YinYang find?**

From July 2019 to October 2019, we extensively tested Z3 and CVC4 with YinYang. YinYang usually reports many bug-triggering test cases in one testing round. To avoid duplicate bug reports, we always use the trunk versions of the solvers for testing. Once the developers have fixed a bug, we validate the fixed version on the rest of the formulas which triggered bugs in the previous testing round. If the solvers passed all formulas and no bug was triggered, we started a new testing round. During our four months of testing, YinYang generated around 800 million test formulas. On
average, YinYang generates 41.5 test formulas per second when run in the single-threaded mode. Figure 8a shows the bug counts categorized by reported, confirmed, fixed, duplicate and won’t fix. From the 57 reported bugs, 45 bugs were confirmed by the developers as real bugs and 41 bugs were fixed. Although we devoted equal testing effort to both solvers, YinYang found more bugs in Z3 (37 confirmed bugs) and clearly fewer bugs in CVC4 (8 confirmed bugs). Having observed that YinYang can find a significant number of bugs in Z3 and CVC4, Figure 8b shows the bug type overview of YinYang’s findings. For bug reporting, we distinguish the following three types of bugs:

- **Soundness bugs**: A formula triggers a soundness bug if the solver reports an incorrect solving result.
- **Crash bugs**: A formula triggers a crash bug if the solver terminates abnormally or throws internal errors while processing the formula.
- **Performance and unknown bugs**: A formula triggers a performance bug if the solver reports unknown or cannot terminate on a simple formula and the developers confirm implementation issues.

Overall, the most common bug category is for soundness bugs (29 out of the 57 reported bugs) followed by crash bugs (12 out of the 57 reported bugs). This is consistent for both solvers, which shows the strength of YinYang in finding soundness bugs. Although we designed YinYang to target soundness and crash bugs, we also considered performance bugs. We have found these bugs during the reduction process of C-Reduce. As performance bugs are less interesting than soundness and crash bugs, we stopped reporting performance bugs after several bug reports and solely focused on soundness and crash bugs subsequently. Figure 8c shows the logic distribution among the confirmed bugs. In Z3, we found most of the bugs in NRA (15) followed by QF_S (15), QF_SLIA (3), NIA (2) and QF_NRA (2). In CVC4, we found most of the bugs in QF_S (4).

**RQ2: How significant are the bug-finding results?**

To approach this question, we consider the most critical bugs in SMT solvers, i.e., the soundness bugs. Soundness bugs in SMT solvers are rare and heavily penalized when detected in the SMT competitions [14]. We have conducted a study on soundness bugs based on the GitHub issue trackers of Z3 and CVC4. The results are shown in Figure 9. For Z3, we considered April 2015 as the start date, right after Z3 was released on GitHub. For CVC4, we have data since July 2010 as CVC4’s previous Bugzilla issue tracker was migrated to GitHub. Z3 supports a myriad of logics and has become very popular on GitHub (5, 196 stars). However, there were only 146 soundness bugs reported on the Z3 issue tracker from April 2015 to October 2019. For CVC4 this number is even lower. Since July 2010, there were only 42 soundness bugs. Of all the soundness bugs in Z3, we found 24 out of 146 (16%). For CVC4, we found 5 soundness bugs out of 43 (11%) in only four months. We found 18 out of the 25 soundness bugs in non-linear logics in Z3 since 2015 and 15 out of the 53 soundness bugs in its string logic. As an intermediate conclusion to RQ2, we state that YinYang has found a significant number of the soundness bugs in both Z3 and CVC4. To more deeply understand the significance of our soundness bug findings, we studied the influence of soundness bugs in different releases of Z3 and CVC4. Figure 10 shows the results. We selected all released versions of Z3 and CVC4 that support the formulas triggering soundness bugs. Z3 4.5.0 was released on November 8, 2016, and CVC4 1.5 was released on July 10, 2017, which means that YinYang found 8 soundness bugs in Z3 that were latent for 3 years, and 2 soundness bugs in CVC4 that were latent for 2 years. YinYang has found long-latent bugs missed by solver developers, users, regression testing, and prior automated testing. This confirms the significance of our bug findings.

**RQ3: Can YinYang improve code coverage?**

In this research question, we use code coverage, a standard evaluation metric for software testing, to understand whether YinYang can cover additional code inside the SMT solvers. To investigate the coverage improvement of YinYang, we consider the following steps:

1. Run Z3 and CVC4 on all formulas in each benchmark.
2. Measure the line, function, and branch coverage of the solvers. The results are labeled as Benchmark.
3. After running Z3 and CVC4 on each benchmark, run YinYang for one hour in single-threaded mode on each benchmark.
4. Measure the line, function, and branch coverage of the solvers. The results are labeled as YinYang.
The study investigates whether we can achieve high coverage respectively. The highest coverages are shaded.

The results show that both YinYang and ConcatFuzz can consistently increase the coverage achieved by the Benchmark. This indicates that YinYang can enhance benchmark formulas and exercise previously uncovered code. Furthermore, YinYang can achieve this noticeable coverage improvement in only one hour in the single-threaded mode, showing the effectiveness of YinYang.

**RQ4: Is Semantic Fusion necessary for finding bugs?**

This research question tests whether we can obtain our bug findings with a simpler approach. As mentioned earlier, Semantic Fusion consists of two main steps: (1) formula concatenation and (2) variable fusion and inversion. Step (2) is the core technique of Semantic Fusion. Let ConcatFuzz be the simple concatenation tool where we solely perform step

---

**Figure 11.** Code coverage evaluations. The numbers represent the percentage (%) coverage for the corresponding coverage metric. Column \( l, f, b \) represent line coverage, function coverage, and branch coverage respectively. Higher coverage between Benchmark and YinYang is shaded.

**Figure 12.** Coverage improvement (%) of ConcatFuzz (in gray) and YinYang (in black) over Benchmark (in white) averaged over all logics.

(1) and disable step (2), i.e., ConcatFuzz only combines formulas by conjunction (for satisfiable formulas) and disjunction (for unsatisfiable formulas) without variable fusion and inversion. To see whether Semantic Fusion is necessary for triggering bugs, we ran ConcatFuzz on the ancestor seeds of 50 reported bugs that YinYang found. In only 5 out of 50 cases, ConcatFuzz was able to retrigger the bug. This indicates that simple formula concatenation is unable to trigger most of the bugs found by YinYang, which shows the necessity of the core technique of Semantic Fusion. In addition, we also repeated the code coverage evaluation of RQ3 to understand the code coverage difference between ConcatFuzz and YinYang. Figure 12 shows the code coverage of ConcatFuzz and YinYang averaged over all logics. The results show that both YinYang and ConcatFuzz consistently achieve higher line, function, and branch coverage than the simple concatenation tool where we solely perform step
the Benchmark. The coverage improvement of ConcatFuzz over the benchmark, partially explains why ConcatFuzz can retrigger some of the bugs. However, the results also reflect that the average code coverage of YinYang dominates ConcatFuzz. YinYang achieves on average 1.1% more lines (approx. 2,800 lines) in Z3 and 0.3% (approx. 480 lines) in CVC4, which can partially explain why YinYang can trigger more bugs than ConcatFuzz. In summary, our results show that semantic fusion is necessary for YinYang’s effectiveness, and code coverage and bug count correlate.

### 4.3 Assorted Bug Samples

This section presents a selection of bugs that we found in CVC4 and Z3. We have found soundness bugs, segmentation faults, assertion violations and performance issues in multiple logics. The original seed formulas \( \phi_{\text{ fused}} \) that trigger the bugs are too large to be presented. We therefore present the reduced formulas which we have obtained from bug reduction on the original bug-triggering formulas.

**Figures 13a** shows an unsatisfiable formula in the string logic QF_S. The formula has two asserts. The second assert demands variable \( a \) to be the concatenation of \( b \) and \( c \). The first asserts includes the check on whether \( c \) is matched by regex ”aa*”. This is conjoined with a check on whether \( c \) equals ”0” (see lines 8 to 11). The formula is unsatisfiable since the two conditions contradict each other. However, Z3 reports \( \text{sat} \) on Formula 13a, which is incorrect.

**Figure 13b** also shows a formula in the string logic QF_S. The formula is unsatisfiable but CVC4 incorrectly reported \( \text{sat} \) on it. The formula has been reduced from the same original test case as Formula 13a which also triggered a Z3 soundness bug. CVC4 and Z3 both report \( \text{sat} \) on the original formula. These two bugs show the benefits of our approach over differential testing. In this case, differential testing would not be able to capture these bugs as the results from both solvers are the same, but incorrect. The root cause of the bug is a missed corner case in the \text{str.to.int} reduction function for an empty string. The bug was labeled as major in the CVC4 bug tracker.

**Figure 13c** shows an unsatisfiable formula in the non-linear real arithmetic logic QF_NRA. Z3 reported \( \text{sat} \) on this formula and gave the following incorrect model:

```lisp
(define-fun e () Real 1.0)
(define-fun f () Real 2.0)
(define-fun a () Real 1.0)
(define-fun b () Real (- 1.0))
(define-fun c () Real 0.8)
(define-fun d () Real 1.0)
```

This model does not satisfy the formula. It causes conflicts between two constraints — the first is the constraint on line 10, while the second is on line 11. According to the SMT-LIB standard, an arbitrary but consistent value \( v \) may be chosen for such division-by-zero predicates. Thus, the formula is unsatisfiable. However, Z3 reported \( \text{sat} \) on the formula. Z3 chose a positive \( f \), and therefore \( v \) has to be positive contradicting line 11.

**Figure 13d** shows an unsatisfiable formula in the QF_SLIA logic. CVC4 incorrectly reported \( \text{sat} \) on this formula and gave an incorrect model. The root cause for this bug is an unsound formula simplification of CVC4. The bug is labeled major by the CVC4 developers. They rewrote the simplification strategy to fix the bug.

**Figure 13e** shows an unsatisfiable formula in the string logic QF_S. Z3 incorrectly reported \( \text{sat} \) on this formula and gave an incorrect model. The developers made some major changes to fix this bug — 28 files with 486 additions and 144 deletions were necessary to fix it. The bug is triggered by an incorrect implementations of the \text{suffixof} and \text{prefixof} operators.

**Figure 13f** shows a formula in quantified real arithmetic (NRA). Z3 crashed when solving this formula with the following message:

```plaintext
Failed to verify: m_util.is_numeral(rhs, _k)
[2] 25132 segmentation fault (core dumped)
```

According to the bug fix of the developer, the root cause for this crash was an error in the rewriting strategy for the comparison operators \( <= \) and \( >= \).

### 4.4 Discussion

**Summary.** Our extensive evaluation demonstrates that YinYang can find many bugs in state-of-the-art SMT solvers Z3 and CVC4 (RQ1) and its findings are significant (RQ2). The further evaluation shows that YinYang can improve the code coverage (RQ3) and Semantic Fusion is necessary for YinYang’s effective bug finding (RQ4).

**Quality of the bug-finding results.** Our 4-month testing effort produced significant, high-quality results. We focused specifically on finding bugs in the default modes of arithmetic and string solvers. As most users invoke SMT solvers in their default modes, such bug reports are thus particularly valuable. YinYang found 29 such critical soundness bugs, which shows the effectiveness of Semantic Fusion. In addition, the SMT solver developers greatly appreciated our bug finding effort and reports with comments like "great find!", "excellent find!", "nice catch!", etc. All of our CVC4 soundness bug reports were labeled major, which are rare on the CVC4 issue tracker — in fact, only 13 soundness bugs in the CVC4 issue tracker are labeled major. Our bugs in Z3 have been fixed in the recent release versions, which makes Z3 more reliable and robust.

**Limitations and future work.** Semantic Fusion is shown to be effective for SMT solver testing. It does also come with some limitations. First, Semantic Fusion relies on the given
We discuss three strands of related work: (1) SAT/SMT solver validation, (2) validation of program analyzers, and (3) metamorphic testing.

**SAT/SMT Solver Validation.** FuzzSMT [7] is the first effort targeting SMT solver validation. It is based on grammar-based blackbox fuzzing and differential testing. Brummayer and Biere evaluated FuzzSMT on the bit-vector logic and found 16 defects in five solvers, but no soundness bugs in Z3. BtorMBT [26] is a model-based testing tool for Boolector [6], an SMT solver for bit-vectors with arrays. It generates sequences of API calls to exploit the features of the solver. BtorMBT did not find bugs in any mature solvers. The more recent StringFuzz [5] focuses on performance issues in the

![Figure 13](https://github.com/Z3Prover/z3/issues/2391)

seed formulas. Second, one needs to manually design the fusion and inversion functions; devising variable fusion and variable inversion functions by hand can be difficult. It would be interesting future work to explore the automatic construction of variable fusion and inversion functions.

**5 Related Work**

We discuss three strands of related work: (1) SAT/SMT solver validation, (2) validation of program analyzers, and (3) metamorphic testing.
string logic. It generates test cases by either mutating and transforming the benchmarks, or generating random valid formulas. It found 3 performance and implementation bugs in z3str3 by differential testing. Different from these differential testing-based approaches, Semantic Fusion tackles the test oracle problem by construction, rather than cross-checking, making it capable of testing solver-specific features.

The recent work of Bugariu and Müller [9] proposed a formula synthesis approach to generating SMT formulas in the string logic with known satisfiability. It generates increasingly-complex formulas via satisfiability-preserving transformations. Several bugs in Z3 were reported, while no reported bugs in CVC4. We instead fuse two formulas and preserve the satisfiability via fusion/inversion functions. A closely-related problem to testing SMT solvers is the testing of SAT solvers. FuzzSAT, CNFuzz, and 3SAT [8] randomly generate valid formulas and found 14 bugs in 7 SAT solvers via differential testing.

Our work found 45 confirmed bugs in modern, mature and widely-used SMT solvers, significantly more than any prior work on SMT/SAT solver testing. Semantic Fusion is also general and can find bugs in various logics while much prior work focuses on the string logic.

Validation of Program Analyzers. With program analyzers becoming increasingly practical and adopted, it is critical to ensure their reliability [11]. Several recent efforts explored this problem, targeting software model checkers, symbolic execution engines, and various static analyzers. Both Zhang et al. [35] and Klinger et al. [22] developed approaches to testing software model checkers — the approach by Zhang et al. [35] is based on reachability queries, while the approach by Klinger et al. [22] is based on differential testing. Kapus et al. [21] used random program generation and differential testing to find bugs in symbolic execution engines. Wu et al. [34] found bugs in alias analyses via cross-checking with dynamic aliasing information. Bugariu et al. [10] proposed an approach for finding soundness and precision bugs in numerical abstract domains. Qiu et al. [28] and Pauck et al. [27] reported experiences in testing and finding defects in analyzers for Android apps.

Many of these program analyzers, such as software model checkers, symbolic execution engines, and program verifiers, critically rely on SMT solvers. Thus, although our work targets SMT solvers, it also helps improve the reliability of program analyzers.

Metamorphic Testing. The test oracle problem is a long-standing challenge in software testing. Metamorphic testing is a general approach to this problem [13, 30]. Its key idea is to leverage existing tests to construct additional ones with expected results via certain metamorphic relations. For example, the technique of equivalence modulo inputs (EMI) [23] is a notable instance of metamorphic testing for compilers. It constructs equivalent test programs for a seed program with respect to a given input by strategically mutating the seed program. To date, the general EMI-based approach and its variants [24, 32, 36] have found more than 1,600 bugs in GCC and Clang/LLVM. EMI and metamorphic testing in general were also adapted to test shader compilers [18, 25].

The Semantic Fusion methodology introduced in this paper is also an instance of metamorphic testing — it generates new test formulas by fusing two existing test cases preserving their oracle. Semantic Fusion is a new and highly generic metamorphic testing approach that we successfully applied to SMT solver testing.

6 Conclusion

We have introduced Semantic Fusion, a novel, principled testing methodology for validating SMT solvers. The key idea behind Semantic Fusion is to construct diverse test formulas for SMT solvers by fusing pairs of equisatisfiable formulas. It effectively tackles both challenges of test input and oracle generation by constructing equisatisfiable formulas as the seed formulas via the concept of fusion and inversion functions. We have designed and engineered the practical bug detection tool YinYang to stress-test SMT solvers. Within four months of extensive testing, we have used YinYang to find 45 confirmed/fixed, unique bugs in Z3 and CVC4, 29 of which are the critical soundness bugs. Our effort is well-appreciated by the SMT solver developers — it is the largest, most successful testing campaign against SMT solvers, leading to drastically more bugs than any previous approaches.

For future work, within the context of SMT solvers, it would be interesting to explore the automatic generation of fusion and inversion functions, and to construct effective customized reducers for SMT formulas. Because solver clients often interact with the solvers via their provided APIs, it would also be interesting to adapt Semantic Fusion to test the solver APIs. Beyond validating SMT solvers, Semantic Fusion is a general technique and may be adapted to other domains; examples include the testing of database engines, compilers, and numerical solvers. This work opens up this exciting new direction via a successful application of Semantic Fusion to the testing of SMT solvers.

Acknowledgements

We thank our shepherd, Isil Dillig, and the anonymous PLDI reviewers for their valuable feedback. We are also grateful to Tim King for a corrected proof of Proposition 1. Our special thanks goes to the Z3 and CVC4 developers, especially Nikolaj Bjørner, Andrew Reynolds, and Andres Nožti, for useful information and addressing our bug reports. Chengyu Zhang was partially supported by China Scholarship Council, and NSFC Projects No. 61632005 and No. 61532019.
References