Skeletal Approximation Enumeration for SMT Solver Testing

Peisen Yao
The Hong Kong University of Science and Technology
Hong Kong, China
pyao@cse.ust.hk

Heqing Huang
The Hong Kong University of Science and Technology
Hong Kong, China
hhuangaz@cse.ust.hk

Wensheng Tang
The Hong Kong University of Science and Technology
Hong Kong, China
wtangae@cse.ust.hk

Qingkai Shi
Ant Group
China
qingkai.sqk@antgroup.com

Rongxin Wu
Xiamen University
China
wurongxin@xmu.edu.cn

Charles Zhang
The Hong Kong University of Science and Technology
Hong Kong, China
charlesz@cse.ust.hk

ABSTRACT

Ensuring the equality of SMT solvers is critical due to its broad spectrum of applications in academia and industry, such as symbolic execution and program verification. Existing approaches to testing SMT solvers are either too costly or find difficulties generalizing to different solvers and theories, due to the test oracle problem. To complement existing approaches and overcome their weaknesses, this paper introduces skeletal approximation enumeration (SAE), a novel lightweight and general testing technique for all first-order theories. To demonstrate its practical utility, we have applied the SAE technique to test Z3 and CVC4, two comprehensively tested, state-of-the-art SMT solvers. By the time of writing, our approach had found 71 confirmed bugs in Z3 and CVC4, 55 of which had already been fixed.

CCS CONCEPTS

• **Software and its engineering** → *Software verification and validation.*

KEYWORDS

SMT solver testing, metamorphic testing, mutation-based testing

ACM Reference Format:
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 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468540

1 INTRODUCTION

Satisfiability Modulo Theory (SMT) solvers evaluate the satisfiability of formulas over first-order theories, such as integers, reals, bit-vectors, and strings [11, 20, 27, 28, 30, 32, 42]. To date, SMT solvers have been widely used in a variety of techniques, such as testing [23, 33], verification [13, 29], program repair [41], program synthesis [14, 50], and others. Despite the tremendous research progress in SMT solving, state-of-the-art SMT solvers are still error-prone [22, 39, 59, 60]. Bugs in SMT solvers can affect the correctness and robustness of the software that depends on the solvers. For instance, in symbolic execution, spurious satisfying assignments (i.e., models) for path conditions are mapped to infeasible test inputs. In program verification, wrong satisfiability results can invalidate the results of the verifiers, which can have detrimental consequences for safety-critical domains [45, 58].

The predominant approach to validating SMT solvers consists of various testing techniques. An important and challenging problem is the test oracle, i.e., the input formula's satisfiability, which is crucial for detecting correctness bugs in SMT solvers. For example, a buggy solver may return "unsat" (i.e., unsatisfiable) for satisfiable formulas, or return "sat" (i.e., satisfiable) for unsatisfiable formulas. To address the oracle problem, there are two categories of techniques, differential testing and oracle-guided approach. Differential testing techniques randomly generate syntactically valid formulas, solve the formulas using multiple SMT solvers, and compare the solving results to identify correctness bugs [15, 19, 48, 59]. However, differential testing cannot be applied when a formula contains some solver-specific extensions. For example, a formula with the specific Z3 [28] extension "assert-soft" cannot be solved by CVC4 [11].

In comparison, the oracle-guided approach systematically synthesizes formulas whose satisfiability results are known by construction [22, 39, 60]. Such information acts as the oracle, i.e., an SMT solver violating the results is buggy. Compared to differential testing, they do not need to run a formula against different solvers and, thus, are usually more lightweight and easier to deploy. However, we observe that existing oracle-guided techniques still face the generality problem, owing to the innate semantic complexity of SMT problems. As illustrated in the last three rows of Table 1, on the one hand, some strategies only apply to specific theories [22, 60]. On the other hand, while the technique in [39] is general for different theories, it only partially addresses the oracle problem, i.e., its mutation strategy can only generate satisfiable mutants. Consequently, the technique can miss certain bugs, i.e., a formula is unsatisfiable, but the solver returns "sat".

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468540

Table 1: Comparison among state-of-the-art techniques. The "Oracle" column represents whether the tool can generate formulas whose satisfiability is known by construction. The "Theory" column represents the supported theories.

Technique	Oracle	Theory
Blotsky et al. [15]	differential	string
Scott et al. [48]	differential	string, float
Winterer et al. [59]	differential	all theories
Bugariu et al.[22]	sat,unsat	string
Winterer et al. [60]	sat,unsat	int, real, string
Mansur et al. [39]	sat	all theories

This paper presents a new oracle-guided approach to testing SMT solvers, which significantly complements existing work, in that our strategy is general enough for all theories and can generate both satisfiable and unsatisfiable mutants. Our approach builds on the metamorphic testing formalization [25], which takes as input a seed formula φ, and tests the solver by identifying the inconsistency between the satisfiability result of φ and its equi-satisfiable mutants. To generate the mutants, we propose to use a less-explored metamorphic relation that rests on a fundamental property of first-order formulas: let φ_o and φ_u be an over-approximation and an under-approximation of φ respectively, we have (1) φ is satisfiable $\Rightarrow$ φ_o is satisfiable, and (2) φ is unsatisfiable $\Rightarrow$ φ_u is unsatisfiable. Thus, in our approach, with the satisfiability of a seed φ determined by the solver, we further compute an equi-satisfiable mutant φ_o or φ_u via logical approximations. If the solver does not provide a consistent result for φ and its approximations, then a potential flaw manifests. By nature of the metamorphic relation, our approach can generalize to different theories and generate both satisfiable and unsatisfiable mutants.

To instantiate the metamorphic relation, the major challenge is how to correctly and efficiently approximate the seed formulas. Different from existing logical approximation techniques that are either theory-specific [16, 18, 21] or costly due to heavyweight logical reasoning [35, 37, 40], this paper introduces a novel general and lightweight approximation approach, namely *skeletal approximation enumeration (SAE)*. SAE views an SMT formula, e.g., $\varphi \equiv (x < 10 \land y > 5) \lor (x > 2 \lor z > 3)$, as two parts: a Boolean structure, e.g., $(\square_1 \land \square_2) \lor (\square_3 \lor \square_4)$, and a set of literals, e.g., $\{x < 10, y > 5, x > 2, z > 3\}$. The essence of SAE is to obtain equi-satisfiable formulas by mutating the literals $\square_i$ locally, which is usually lightweight. We present a practical realization of SAE, which is embodied by a series of literal-level mutation rules, and a systematic way to combine the local mutants. We also prove the correctness of our algorithms.

We have implemented the proposed idea as a tool named Sparrow and applied the tool to testing Z3 and CVC4, two state-of-the-art and comprehensively tested SMT solvers. By the time of writing, Sparrow had found 29 and 42 confirmed bugs in Z3 and CVC4, respectively. Besides, 55 of the bugs had been fixed by the developers, among which 23 are correctness bugs. In summary, this paper makes the following contributions:

- We present a new metamorphic testing approach for SMT solvers, which leverages logical approximations as the metamorphic relation.
- We introduce skeleton approximation enumeration (SAE), a methodology for approximating SMT formulas, and propose a practical realization of SAE.
- We implement our approach as the Sparrow tool, which detects 71 confirmed bugs in Z3 and CVC4, two state-of-the-art SMT solvers. We also present several in-depth evaluations to understand Sparrow's effectiveness.

2 PRELIMINARIES

In this section, we present the notations and terminologies throughout the paper.

Basic Notations. Satisfiability modulo theories (SMT) extend the Boolean satisfiability problem (SAT) with the capability of reasoning with first-order theories, such as integers, reals, arrays, and strings. In a first-order theory, a *term* can be a variable, a constant, or an n-ary function applied to n terms. An *atom* is true, false, or an n-ary predicate applied to n terms. A *literal* is an atom or its negation. A formula is built from atoms using the *Boolean connectives*, e.g., $\neg, \land, \lor, \rightarrow$. Given a formula φ, we denote its free variables and literals by $vars(\varphi)$ and $lits(\varphi)$, respectively. For ease of presentation, in the rest of the paper, we assume that all literals in $lits(\varphi)$ are distinct.

Definition 2.1. (Conjunctive Normal Form) A formula is in the conjunctive normal form (CNF) if it is a conjunction of one or more clauses ($C_1 \land C_2 \land \cdots$), where each clause C_i is a disjunction of one or more literals ($l_1 \lor l_2 \lor \cdots$).

Example 2.1. In the theory of integers, the function symbols include $\{+, -, *, /\}$ and the predicate symbols consist of $\{<, \leq, >, \geq, =, \neq\}$. Consider an integer formula $\varphi \equiv x \geq 1 \lor \neg(x * x > 10)$. The expressions 1, 10, x, and $x * x$ are integer terms, and the expressions $x \geq 1$ and $\neg(x * x > 10)$ are literals. The formula is in CNF, which has only one clause. By contrast, the formulas $x > 2 \rightarrow x > 1$ and $\neg(x > 2 \land y > 10)$ are not in CNF.

Logical Approximations. A formula φ is satisfiable if there exists an assignment to $vars(\varphi)$, under which the formula evaluates to true. If it is impossible to find such an assignment, the formula is unsatisfiable. A formula φ is a *tautology* iff its negation $\neg\varphi$ is unsatisfiable. We write $\phi \vdash \psi$ to denote that the formula $\phi \rightarrow \psi$ is a tautology. In other words, $\phi \vdash \psi$ iff the formula $\neg(\phi \rightarrow \psi) \equiv \neg(\neg\phi \lor \psi) \equiv \phi \land \neg\psi$ is unsatisfiable.

Definition 2.2. (Over- and Under-Approximations) Let φ be a first-order formula. We say a formula φ_o *over-approximates* φ iff $\varphi \vdash \varphi_o$ (i.e., $\varphi \land \neg\varphi_o$ is unsatisfiable). We say a formula φ_u *under-approximates* φ iff $\varphi_u \vdash \varphi$ (i.e., $\varphi_u \land \neg\varphi$ is unsatisfiable).

Intuitively, the over- and under-approximations of a formula are the necessary and sufficient conditions for the formula to be satisfiable, respectively.

Example 2.2. Consider an integer formula $\varphi \equiv x > 5$. Clearly, the formula $\varphi' \equiv x > 1$ over-approximates φ. If $x > 1$ does not hold, then $x > 5$ must also be unsatisfiable. Conversely, we can say that φ under-approximates φ'.

Metamorphic Testing. A test oracle is a mechanism for determining whether a test has passed or failed. Under certain circumstances, however, the oracle is unavailable or too expensive to achieve. This is known as the test oracle problem [57]. Metamorphic testing [25] can be exploited to alleviate the problem. Based on the existing successful test cases, metamorphic testing generates follow-up test cases by referring to domain-specific *metamorphic relations*, which are the necessary properties of the target program in terms of multiple inputs and their expected outputs. The violation of a metamorphic relation will be suspicious and indicate a potential bug.

Example 2.3. Let F be a program implementing the transcendental function "sin". The equation $sin(\pi - x) = sin(x)$ is a typical metamorphic relation with respect to F. Hence, given a successful test case, say $a = 1.2$, metamorphic testing generates its follow-up test case $a' = \pi - 1.2$, and then runs the program over a'. Finally, the two outputs, i.e., $F(a)$ and $F(a')$, are checked to see if they satisfy the expected relation $F(a) = F(a')$. If the relation does not hold, a bug in F manifests.

An important property of metamorphic testing is that it does not need a reference engine for differential testing, because the metamorphic relation provides an explicit oracle, i.e., the seed and its variants must output the same result. When testing SMT solvers, this property is particularly beneficial when the test inputs contain some solver-specific extensions.

3 PROBLEM FORMULATION

In this section, we first present the metamorphic relation used in our work. We then formulate skeletal approximation enumeration, which aims to realize the metamorphic relation effectively.

3.1 Approximation-based Metamorphic Relation

In SMT solver testing, the satisfiability of randomly generated formulas is typically unknown, without which we cannot decide whether the solver correctly solves the formulas. To address the problem, our work follows the metamorphic testing formulation. Specifically, we propose to use a less-explored metamorphic relation for SMT problems, which builds on the following fundamental property of first-order logic.

Theorem 3.1. *Let φ_o and φ_u be an over-approximation and an under-approximation of a first-order formula φ, respectively (i.e., $\varphi \vdash \varphi_o$ and $\varphi_u \vdash \varphi$). We have (1) if φ is satisfiable, then φ_o is also satisfiable, and (2) if φ is unsatisfiable, then φ_u is also unsatisfiable.*

Example 3.1. Consider the formula $\varphi \equiv x > 5$ in Example 2.2. The formula $\varphi' \equiv x > 1$ over-approximates φ. Clearly, if φ is satisfiable, then φ' is also satisfiable. Conversely, φ under-approximates φ'. If φ' is unsatisfiable, then φ must be unsatisfiable.

Based on Theorem. 3.1, a metamorphic relation can be defined within a pair of formulas (φ, φ'), where φ is a seed formula, and φ' is a mutant. Given a formula ψ and an SMT solver S, we denote $S(\psi)$ the result of using S to solve ψ. We then define the metamorphic relation as follows:

$$S(\varphi) = S(\varphi'), \text{ where} \qquad (1)$$
$$\begin{cases} \varphi \vdash \varphi' & \text{if } \varphi \text{ is satisfiable} \\ \varphi' \vdash \varphi & \text{if } \varphi \text{ is unsatisfiable} \end{cases}$$

To test the solver S, we could obtain and compare the solving results $S(\varphi)$ and $S(\varphi')$. If $S(\varphi) \neq S(\varphi')$, then a bug in the solver is revealed.

3.2 Skeletal Approximation Enumeration

To effectively instantiate the metamorphic relation in Equation 1, we formulate the skeletal approximation enumeration problem, a new methodology for approximating first-order formulas. As mentioned in § 1, an SMT formula φ consists of two parts: a Boolean structure and a set of literals in some first-order theory.

Definition 3.1. (Skeletal Approximation Enumeration) Given a formula φ with a set of literals $lits(\varphi) = \{\square_1, \ldots, \square_n\}$, skeletal approximation enumeration (SAE) computes the approximations of φ by enumerating the approximations of the literals.

For a given literal $\square_i \in lits(\varphi)$, SAE can randomly pick a new literal l'_i to replace the literal, such that $\square_i \vdash l'_i$ or $l'_i \vdash \square_i$. We denote the resulting mutant as $\varphi' \equiv \varphi[l'_i/\square_i]$.

Example 3.2. Consider a satisfiable integer formula $\varphi \equiv (x < 10 \wedge y > 5) \vee (x > 2 \vee z > 3)$. According to the metamorphic relation in Equation 1, we should obtain the mutants by over-approximating φ. Suppose $x < 10$ is chosen for mutation. After replacing $x < 10$ by $x < 11$ (an over-approximation of $x < 10$), we obtain a mutant $\varphi' \equiv (\, x < 11 \, \wedge y > 5) \vee (x > 2 \vee z > 3)$, which over-approximates the seed φ.

To realize skeletal approximation enumeration, there are many potential methods for selecting the literals in a seed formula and picking the new literals for replacement. These methods should address the following challenges.

Correctness. First, we need to preserve the metamorphic relation in Equation 1, i.e., enforce that the mutant does over- or under-approximate the seed formula, subject to the satisfiability of the seed. In essence, SAE generates the mutants by enumerating literal-level approximations. For the skeleton in Example 3.2, over-approximating the literal $x < 10$ can yield an over-approximation of the seed formula. However, simply approximating a literal in a given formula may lead to a nondeterministic mutant that can be either an over-approximation or an under-approximation. Consequently, the mutation strategy can violate Equation 1, making the satisfiability of the mutant unpredictable.

Example 3.3. Suppose we need to over-approximate an integer formula $\varphi \equiv x > 5 \rightarrow y > 10$. Clearly, the formula $x > 1$ over-approximates $x > 5$. However, after replacing $x > 5$ by $x > 1$, the new formula $\varphi' \equiv \, x > 1 \, \rightarrow y > 10$ does not over-approximate φ. This is because

$$\varphi \equiv \neg(x > 5) \vee y > 10 \equiv x \leq 5 \vee y > 10$$
$$\varphi' \equiv \neg(x > 1) \vee y > 10 \equiv x \leq 1 \vee y > 10$$

Since $x \leq 1$ under-approximates $x \leq 5$, we conclude that φ' is an under-approximation of φ.

Efficiency and Generality. Second, it is crucial but challenging to approximate the seed formulas in various theories efficiently, considering SMT problems' innate semantic richness. The formulas can encode relations about diverse variables, such as integers, reals, bit-vectors, and floating points. The relations can be combined with different Boolean connectives ($\wedge, \vee, \neg, \rightarrow$, etc.) in a complicated manner. While there are many algorithms for approximating first-order formulas in the SMT solving literature, most of them are either theory-specific [16, 18, 21], or rely on heavy-weight logical reasoning [35, 37, 40], thereby undermining their applicability in SMT solver testing.

Problem Statement. Based on the discussion above, we aim to address two challenges in using skeletal approximation enumeration to instantiate the metamorphic relation in Equation 1:

> (1) How to guarantee that the generated mutant indeed over- or under-approximates the seed formula? (2) How to design the mutation strategies for efficiently mutating literals in different theories?

4 APPROACH

In this section, we first present and prove the basic principle underlying our approach, which addresses the first challenge. We then present our literal-level mutation strategies, which address the second challenge. Finally, we describe how to combine the literal-level mutants for testing SMT solvers.

4.1 Approximation from CNF

Skeletal approximation enumeration approximates a formula by enumerating the approximations of its literals. To ensure that the literal-level approximations preserve the metamorphic relation in Equation 1, we need to fill the gap between (1) approximating a seed formula that can be an arbitrary Boolean combination of literals and (2) approximating the individual literals in the seed.

To fill the gap, our key idea is to first transform a seed formula into a suitable "internal form", and then design the mutation operators. Our idea draws inspiration from existing SMT solving algorithms that often build on some specific representations, such as the add-inverter graph (AIG) [62] the conjunctive normal form (CNF). Specifically, we lay the foundation of our approach with the following theorem.

Theorem 4.1. *Let φ be a first-order formula in CNF and $l \in lits(\varphi)$ a literal in φ. We have (1) if a formula l' over-approximates l, then $\varphi[l'/l]$ must over-approximate φ (i.e., $l \vdash l' \Rightarrow \varphi \vdash \varphi[l'/l]$), and (2) if a formula l' under-approximates l, then $\varphi[l'/l]$ must under-approximate φ (i.e., $l' \vdash l \Rightarrow \varphi[l'/l] \vdash \varphi$).*

Proof. We first prove (1). By the definition of CNF (Definition 2.1), we assume that $\varphi \equiv C_1 \wedge C_2 \wedge \cdots \wedge C_n$, where each C_i ($1 \leq i \leq n$) is a disjunction of literals. Without loss of generality, we suppose l'_k over-approximates l_k in the n-th clause $C_n \equiv (l_1 \vee l_2 \vee \cdots \vee l_k)$, i.e., $l_k \wedge \neg l'_k$ is unsatisfiable. Next, we prove (1) by induction on the structure of CNF formulas.

(a) First, we prove that $C'_n \equiv C_n[l'_k/l_k]$ over-approximates C_n, i.e., the following formula is unsatisfiable.

$$C_n \wedge \neg C'_n \equiv (l_1 \vee \cdots \vee l_{k-1} \vee l_k) \wedge \neg(l_1 \vee \cdots \vee l_{k-1} \vee l'_k)$$

For simplicity, we abbreviate $(l_1 \vee \cdots \vee l_{k-1})$ to B.

$$\begin{aligned} C_n \wedge \neg C'_n &\equiv (B \vee l_k) \wedge \neg(B \vee l'_k) \\ &\equiv (B \vee l_k) \wedge (\neg B \wedge \neg l'_k) \\ &\equiv (B \wedge \neg B \wedge \neg l'_k) \vee (l_k \wedge \neg B \wedge \neg l'_k) \\ &\equiv false \vee (l_k \wedge \neg B \wedge \neg l'_k) \end{aligned}$$

Since $l_k \wedge \neg l'_k$ is unsatisfiable, we conclude that $C_n \wedge \neg C'_n$ is unsatisfiable, i.e., C'_n over-approximates C_n.

(b) Then, we prove that $\varphi' \equiv \varphi[C'_n/C_n]$ over-approximates φ, i.e., the following formula is unsatisfiable.

$$\varphi \wedge \neg\varphi' \equiv (C_1 \wedge \cdots \wedge C_{n-1} \wedge C_n) \wedge \neg(C_1 \wedge \cdots \wedge C_{n-1} \wedge C'_n)$$

For simplicity, we abbreviate $(C_1 \wedge \cdots \wedge C_{n-1})$ to ψ.

$$\begin{aligned} \varphi \wedge \neg\varphi' &\equiv (\psi \wedge C_n) \wedge \neg(\psi \wedge C'_n) \\ &\equiv (\psi \wedge C_n) \wedge (\neg\psi \vee \neg C'_n) \\ &\equiv (\psi \wedge C_n \wedge \neg\psi) \vee (\psi \wedge C_n \wedge \neg C'_n) \\ &\equiv false \vee (\psi \wedge C_n \wedge \neg C'_n) \end{aligned}$$

Since $C_n \wedge \neg C'_n$ is unsatisfiable (as proved in (a)), we conclude that $\varphi \wedge \neg\varphi'$ is unsatisfiable, i.e., φ' over-approximates φ.

Taking (a) and (b) together, we complete the proof for (1). The proof for (2) is similar. We omit the details due to lack of space. □

Theorem 4.1 is crucial for fulfilling the correctness criterion of skeletal approximation enumeration. Specifically, if a seed formula φ has been transformed into CNF, then locally over- or under-approximating a literal in φ can yield a global over- or under-approximation of the whole formula, respectively. As a result, it allows us to turn the problem of approximating a first-order formula into (1) approximating the literals in the formula and (2) composing the literal-level mutants.

Example 4.1. Consider again the formula $\varphi \equiv x > 5 \rightarrow y > 10$ in Example 3.3. We can first transform φ to CNF, and obtain a normalized formula $\varphi_{cnf} \equiv \neg(x > 5) \vee y > 10 \equiv x \leq 5 \vee y > 10$. We then perform the mutation on the CNF formula. For instance, consider a formula $x < 10$ that over-approximates $x \leq 5$. After replacing $x \leq 5$ by $x < 10$ in φ_{cnf}, we obtain the mutant $\varphi'_{cnf} \equiv$ $x < 10 \vee y > 10$ that over-approximates φ_{cnf}.

In what follows, we detail how to realize skeletal approximation enumeration for testing SMT solvers. Our approach has two building blocks: designing mutation strategies for literals in different first-order theories and combining literal-level mutants to obtain the whole formula's mutants.

4.2 Literal-level Mutation Strategies

We present two mutation strategies, predicate symbol transformation (§ 4.2.1) and live predicate injection (§ 4.2.2), which approximates a literal by mutating the predicate symbols and injecting formula snippets, respectively. Both strategies can over- and under-approximate a literal. Without loss of generality, we assume that the seed formula has been transformed into CNF.

4.2.1 *Predicate Symbol Transformation (PST).*

The first strategy is to mutate the predicate symbol in a literal. Our basic observation is that the predicate symbols in a first-order theory can have some partial order relations (ordered by logical implication). Thus, transforming the symbols in a literal can yield the over- or under-approximations of the literal. For example, the following is a small sample of transformations for various theories.

- Let x and y be two integer variables. The predicate $x \leq y$ over-approximates $x < y$.
- Let x and y be two real variables. The predicate $x > y$ under-approximates $x \geq y$.
- Let x and y be two string variables. The predicate "x is a prefix of y" is an under-approximation of "y contains x".

In what follows, we detail the mutation strategies for predicate symbols in literals of different theories.

Mutating an Atom. First, suppose that the literals to be mutated do not contain logical negations, i.e., each literal is an atom. Table 2 summarizes the rules for approximating an atom in integers, reals, bit-vectors, floating points, and strings. The rules have several characteristics. First, for a given atom, there can be more than one mutation rule. For example, to over-approximate $x = y$ where x and y are integers, we can mutate "=" to "≤" or "≥". Intuitively, if $x = y$ holds, then $x \leq y$ and $x \geq y$ must also hold. Second, the mutations may introduce fresh constants in the background theory. For example, when over-approximating $x \leq y$ to $x < y + a$, we can use a randomly-generated integer constant a where $a > 0$. Third, there are some unsupported atoms for bit-vectors and floats, due to the overflow semantics [17]. For example, we cannot over-approximate x `bvule` y to x `bvult` $(y$ `bvadd` $a)(a > 0)$ (like the one for integers), because bit-vectors model bounded integers (e.g, 32-bits integers), where the "add" function may overflow. We will present the strategy for handling such atoms in § 4.2.2.

Mutating Negated Literals. So far, the mutation rules only apply to atoms, i.e., negation-free literals. As our goal is to mutate a literal that can contain negations (e.g., $\neg x > y$), we proceed to discuss the approximations of such cases. The overall idea behind our approach is to eliminate the negations, after which we can reuse the rules in Table 2. The challenge, however, is how to eliminate negations soundly. Our solution is embodied by two parts.

First, we can transform the literal to an equivalent and negation-free atom, and then reuse the mutation rules for that atom. For example, to over-approximate the integer literal $\neg x > y$, we first transform it to an equivalent atom $x \leq y$ and then apply an over-approximation rule for $x \leq y$.

Second, there are some cases where it is hard to eliminate negations by finding equivalent atoms. For example, consider a literal $\neg(x$ `str.prefixof` $y)$ in the string theory, which means "x is not a prefix of y". It is nontrivial to represent the literal as an equivalent atom. To handle such cases, our approach leverages the conversions between under- and over-approximations, established by the following proposition.

Proposition 4.2. *Given two first-order formulas φ_o and φ_u that over-approximates and under-approximates a formula φ, respectively, we have (1) $\neg\varphi_o$ under-approximates $\neg\varphi$ (i.e., $\varphi \vdash \varphi_o \Rightarrow \neg\varphi_o \vdash \neg\varphi$), and (2) $\neg\varphi_u$ over-approximates $\neg\varphi$ (i.e., $\varphi_u \vdash \varphi \Rightarrow \neg\varphi \vdash \neg\varphi_u$).*

Proof. We sketch the proof of (1). Since $\varphi \vdash \varphi_o$, we have that $\neg(\varphi \rightarrow \varphi_o)$ is unsatisfiable. To prove (1), we need to show that $\neg(\neg\varphi_o \rightarrow \neg\varphi)$ is unsatisfible. Observe that $\neg(\neg\varphi_o \rightarrow \neg\varphi) \equiv \neg(\neg\neg\varphi_o \vee \neg\varphi) \equiv \neg(\neg\varphi \vee \varphi_o) \equiv \neg(\varphi \rightarrow \varphi_o)$. Since $\neg(\varphi \rightarrow \varphi_o)$ is unsatisfiable, we conclude that $\neg\varphi_o \vdash \neg\varphi$. □

Proposition 4.2 has two implications. First, it allows us to use over-approximating techniques to generate under-approximations, and vice versa. Second, it allows for eliminating the negation in a literal. As such, we can soundly reuse the rules in Table 2. For example, suppose we need to over-approximate a literal $\neg p$, which can be processed in three steps:

1. Negate $\neg p$ and obtain an atom, i.e., $\neg(\neg p) \equiv p$;
2. Under-approximate p (using the rules in Table 2), and let the result be p' (i.e., $p' \vdash p$);
3. Negate p' and obtain the final result $\neg p'$ (By Proposition 4.2, we have that $\neg p'$ over-approximates $\neg p$, i.e., $\neg p \vdash \neg p'$).

In the above process, we only apply under-approximation rules and logical negations, while the final result over-approximates the literal $\neg p$.

Example 4.2. Let $\neg(x$ `str.prefixof` $y)$ be the string literal to be over-approximated. First, we take its negation and obtain an atom x `str.prefixof` y. Second, we under-approximate the atom x `str.prefixof` y, and let the result be $y = x$ `str.++` "alice", where `str.++` represents "string concatenation". Finally, we negate the result and obtain $\neg(y = x$ `str.++` "alice"$)$, which over-approximates $\neg(x$ `str.prefixof` $y)$.

Remarks. First, we should emphasize that literal-level approximations are not restricted to the rules in Table 2. A richer set of rules can be designed. Second, the strategy is similar to the type-aware operator mutation presented by Winterer et al. [59]. However, their mutations do not guarantee the preservation of satisfiability. This is because they operate over formulas with arbitrary Boolean structures, and perform the mutations randomly. For example, consider a trivially unsatisfiable integer formula $2 > 3$. The approach in [59] may mutate the formula to $2 \geq 3$ or $2 < 3$, the second of which has a different satisfiability result.

4.2.2 *Live Predicate Injection (LPI).*

The PST strategy transforms the predicate symbol of a literal but has two limitations. First, it cannot mutate certain literals such as "over-approximating x `bvule` y". Second, the search space is confined by the seed, e.g., PST cannot change the function symbols and Boolean connectives in the seed. To stress-test SMT solvers, we would like to generate syntactically more complex mutants, which can exhibit diverse control- and data-dependence between variables.

To this end, our second mutation strategy enriches a literal by synthesizing a new formula snippet ψ, and "injecting" it back to the seed formula (using some proper Boolean connectives). The basic idea is to utilize the formula snippet to relax or restrict the solution space of a literal, thereby yielding the logical approximations of the literal.

Example 4.3. Consider an integer formula $\varphi \equiv x + y > 5$ and its two mutants below:

- $\varphi_1 \equiv x + y > 5 \ \vee x < 3 \ (\varphi \vdash \varphi_1)$
- $\varphi_2 \equiv x + y > 5 \ \wedge x < 3 \ (\varphi_2 \vdash \varphi)$

Table 2: Mutation rules for approximating an atom in different theories. Each atom may have one or more possible mutants (separated by ","). "a" denotes a randomly generated constant in the corresponding theory. N/A means unsupported.

Logic	Atom	Over-approximation	Under-approximation
Int & Real	$x < y$ (less than)	$x \leq y, x \neq y$	$x + a \leq y$ $(a > 0)$
	$x \leq y$ (less than or equal to)	$x < y + a$ $(a > 0)$	$x = y, x + a < y$ $(a \geq 0)$
	$x > y$ (greater than)	$x \geq y, x \neq y$	$x \geq y + a$ $(a > 0)$
	$x \geq y$ (greater than or equal to)	$x + a > y$ $(a > 0)$	$x = y, x > y + a$ $(a \geq 0)$
	$x = y$ (equal)	$x \leq y, x \geq y$	$a \leq x \leq a \land a \leq y \leq a$ (any a)
	$x \neq y$ (inequal)	$\neg(x = a \land y = a)$ (any a)	$x > y, x < y$
Bit-Vec	x `bvult` y (unsigned less than)	x `bvule` $y, x \neq y$	N/A
	x `bvule` y (unsigned less than or equal to)	N/A	$x = y, x$ `bvult` y
	x `bvugt` y (unsigned greater than)	x `bvuge` $y, x \neq y$	N/A
	x `bvuge` y (unsigned greater than or equal to)	N/A	$x = y, x$ `bvugt` y
	x `bvslt` y (signed less than)	x `bvsle` $y, x \neq y$	N/A
	x `bvsle` y (signed less than or equal to)	N/A	$x = y, x$ `bvsgt` y
	x `bvsgt` y (signed greater than)	x `bvsge` $y, x \neq y$	N/A
	x `bvsge` y (signed greater than or equal to)	N/A	$x = y, x$ `bvsgt` y
	$x = y$ (signed equal)	x bvsle y, x bvsge y	a bvsle x bvsle $a \land a$ `bvsle` y `bvsle` a (any a)
	$x \neq y$ (signed inequal)	$\neg(x = a \land y = a)$ (any a)	x `bvsgt` y, x `bvslt` y
	$x = y$ (unsigned equal)	x `bvule` y, x `bvuge` y	a `bvule` x `bvule` $a \land a$ `bvule` y `bvule` a (any a)
	$x \neq y$ (unsigned inequal)	$\neg(x = a \land y = a)$ (any a)	x `bvugt` y, x `bvult` y
Float	x `fp.lt` y (less than)	x `fp.leq` y, x `fp.neq` y	N/A
	x `fp.leq` y (less than or equal to)	N/A	x `fp.eq` y, x `fp.lt` y
	x `fp.gt` (greater than)	x `fp.geq` y, x `fp.neq` y	N/A
	x `fp.gt` (greater than)	x `fp.geq` y, x `fp.neq` y	N/A
	x `fp.geq` y (greater than or equal to)	N/A	x `fp.eq` y, x `fp.gt` y
	x `fp.eq` y (equal)	x `fp.leq` y, x `fp.geq` y	a `fp.leq` x `fp.leq` $a \land a$ `fp.leq` y `fp.leq` a (any a)
	x `fp.neq` y (inequal)	$\neg(x$ `fp.eq` $a \land y$ `fp.eq` $a)$ (any a)	x `fp.gt` y, x `fp.lt` y
String	x `str.<` y (lexicographic ordering)	x `str.` $\leq y, x \neq y$	$(x$ `str.++` $a)$ `str.` $\leq$ y $(len(a) > 0)$
	x `str.` $\leq y$ (lexicographic ordering)	x str. $< (y$ `str.++` $a)$ $(len(a) > 0)$	$x = y, x$ `str.<` y
	x `str.prefixof` y (x is a prefix of y)	x `str.` $\leq y, y$ `str.contains` x	$y = x$ `str.++` a $(len(a) \geq 0)$
	x `str.suffixof` y (x is a suffix of y)	x `str.` $\leq y, y$ `str.contains` x	$y = a$ `str.++` x $(len(a) \geq 0)$
	x `str.contains` y (x contains y)	y `str.` $\leq x$	y `str.suffixof` x, y `str.suffixof` x
	$x = y$ (equal)	x `str.suffixof` y, x `str.perfixof` $y,$ x `str.contains` y, x `str.` $\leq y$	x `str.suffixof` $a \land$ x `str.prefixof` $a \land$ y `str.suffixof` $a \land y$ `str.prefixof` a (any a)
	$x \neq y$ (inequal)	$\neg(x = a \land y = a)$ (any a)	x `str.<` y, y `str.<` x

In the mutants φ_1 and φ_2, the snippet $x < 3$ is injected via disjunction and conjunction, respectively. Observe that φ_1 and φ_2 over-approximates and under-approximates φ, respectively.

More concretely, we define the mutation strategy as follows.

Definition 4.1. (Live Predicate Injection) Given a CNF formula φ and a literal $l \in lits(\varphi)$, live predicate injection first randomly generates a formula snippet ψ, and then injects the snippet as follows: (1) if φ is satisfiable, it replaces l by $l \lor \psi$, and (2) if φ is unsatisfiable, it replaces l by $l \land \psi$.

The correctness of live predicate injection (LPI) is enforced by the following proposition.

Proposition 4.3. *Let l be a literal and ψ be an any first-order formula. We have (1) $l \lor \psi$ over-approximates l, and (2) $l \land \psi$ under-approximates l.*

Proof. We sketch the proof of (1). By Definition 2.2, we need to prove $\neg(l \rightarrow (l \lor \psi))$ is unsatisfiable. Since $\neg(l \rightarrow (l \lor \psi)) \equiv \neg(\neg l \lor (l \lor \psi)) \equiv \neg(l \lor \neg l \lor \psi) \equiv false$, we have (1). □

Taking Theorem 4.1 and Proposition 4.3 together, we conclude that the mutants produced by LPI are equi-satisfiable with the seed formula φ, and, thus, preserve the metamorphic relation in Equation 1. Moreover, by Proposition 4.3, the conclusion holds regardless of the satisfiability of the formula snippet ψ, which can contain any variables, function symbols, predicate symbols, and Boolean connectives. This property allows us to generate satisfiability-preserving mutants that exhibit different and diverse control- and data-dependence.

Algorithm for LPI. To realize the LPI mutation strategy, we need to generate formula snippets automatically. In what follows, we use integer arithmetic to illustrate the essence of our approach, and the handling of other theories is similar. The key observation behind our solution is that, by nature, SMT formulas follow a layered construction. A formula is a Boolean combination of atoms, which are built on top of lower-level terms. For example, the integer formula $\varphi \equiv x > 1 \land y < 2$ consists of two atoms, where the atom $x > 1$ consists of two terms x and 1.

Algorithm 1 describes the process to build a formula snippet. The function random_select randomly picks one element from a set. The function smt_expr takes as input the operator and operands, and returns an SMT expression. We omit the details of the functions as their implementations are straightforward.

At a high level, Algorithm 1 works in a top-down manner. At the top, we first randomly choose a Boolean connective, e.g., negation, conjunction, and disjunction (line 2). We then proceed to build the atoms. To generate an atom, the function generate_atom

Algorithm 1: Generating an integer formula snippet.

```
   Input: A seed integer formula $\varphi$
   Output: A new formula snippet
1  Function generate_formula_snippet($\varphi$)
2      $op \leftarrow$ random_select($\{none, \neg, \wedge, \vee, \rightarrow, xor\}$);
3      if $op == none$ then
4          return generate_atom($\varphi$);
5      else if $op == \neg$ then
6          $l \leftarrow$ generate_atom($\varphi$);
7          return smt_expr($op, l$);
8      else
9          $l_1 \leftarrow$ generate_atom($\varphi$), $l_2 \leftarrow$ generate_atom($\varphi$);
10         return smt_expr($op, l_1, l_2$);
11 Function generate_atom($\varphi$)
12     $op \leftarrow$ random_select($\{<, \leq, >, \geq, =, \neq\}$);
13     $t_1 \leftarrow$ generate_term($\varphi$), $t_2 \leftarrow$ generate_term($\varphi$);
14     return smt_expr($op, t_1, t_2$);
15 Function generate_term($\varphi$)
16     $op \leftarrow$ random_select($\{+, -, *, /\}$);
17     $v_1 \leftarrow$ random_select($vars(\varphi)$);
18     if $\varphi$ in linear arithmetic and $op \in \{*, /\}$ then
19         $v_2 \leftarrow$ randomly generate a constant ;
20     else
21         $v_2 \leftarrow$ random_select($vars(\varphi)$);
22     return smt_expr($op, v_1, v_2$);
```

(line 11-line 14) first synthesizes two integer terms using the function generate_term (line 15-line 22), and then relates the terms with an integer predicate symbol, e.g., <, ≤ and >. Note that when the seed formula is in the theory of linear integer arithmetic, we should not synthesize non-linear terms such as $x * y$ and x/y. Thus, to avoid generating such terms, the function generate_term restricts the second operand v_2 to an integer constant, when the randomly selected operator is $*$ or / (line 19).

Example 4.4. In the following formula pair, φ' over-approximates φ. The mutant φ' is obtained by replacing $y = z$ by $y = z \vee$ $x + z < y - 1 \; xor \; y - z \geq 2 * x$, where the shaded part is a randomly generated formula snippet.

$$\varphi \equiv (x > y \vee \cdots) \wedge y = z$$

$$\varphi' \equiv (x > y \vee \cdots) \wedge (y = z \vee x + z < y - 1 \; xor \; y - z \geq 2 * x)$$

Remarks. First, in principle, one can generate and inject new formulas that are arbitrarily large and complex. However, generating such formulas can be time-consuming. Second, the mutants generated by LPI are not necessarily in CNF. By definition, CNF formulas only contain three Boolean connectives, i.e., ∧, ∨, and ¬, while LPI can use other Boolean connectives such as *xor* and →.

4.3 Bug Detection with Sparrow

Based on the principle introduced in § 4.1 and the mutation strategies presented in § 4.2, we have designed and implemented Sparrow, a tool for stress-testing SMT solvers.

Bug Types. Sparrow can detect three categories of bugs: (1) *soundness bugs*: the solver returns "unsat" for satisfiable formulas, or returns "sat" for unsatisfiable formulas; (2) *invalid model bugs*: a formula is satisfiable and the solver returns "sat", but the solver yields an infeasible model that falsifies the formula; and (3) *crash bugs*: the solver terminates abnormally when solving a formula, which can be caused by some internal assertion failures or memory safety problems such as buffer overflow. We refer to the first two categories as the *correctness bugs*.

Algorithm. Algorithm 2 shows the general workflow of Sparrow, which takes as input a set of seed formulas and an SMT solver under test. The three sets *soundness_bugs*, *model_bugs*, and *crash_bugs* are used to collect soundness, invalid model, and crash bugs, respectively (line 2). In each round of the loop, we first randomly choose a formula φ from the seeds, convert it into CNF, and pass the CNF formula to the SMT solver. If the solve yields "sat", we generate a mutant φ' by over-approximating φ (line 10). Otherwise, we obtain a mutant φ' by under-approximating φ (line 12). We then invoke the solver to solve the mutant, and check whether it gives a consistent answer or not. If not, we have found a candidate soundness bug (line 15). If the answer is consistent and "sat", but the solver returns a model that falsifies φ', we have found an invalid model bug (line 19). Finally, if the solver crashes on the mutant, we have found a candidate crash bug (line 21).

In Algorithm 2, the two sub-procedures over_approximate and under_approximate can be implemented using the rules presented in § 4.2. Briefly, to approximate a CNF formula φ, we randomly select a subset of literals from $lits(\varphi)$, and then apply the corresponding literal-level mutations. Note that, to guarantee the correctness, the approximation type for a formulas' literals should be the same. For example, if we over-approximate one literal but under-approximate the other, it would be hard to figure out whether the solution space of the CNF formula is enlarged or reduced. Consequently, we cannot ensure whether the final mutant over- or under-approximates the seed formula.

Implementation. We have implemented Sparrow in 9,440 lines of Python code, which instantiates Algorithm 2 as follows. First, we convert a seed formula into CNF using Tseitin [54]'s CNF transformation algorithm, whose time complexity is linear in the formula size. Second, for each seed formula, we generate 300 mutants by default (line 8). Third, since the number of literals in a formula can be gigantic, in practice, we bound the number of mutated literals in each mutant as 5. To mutate each selected literal, we randomly apply one applicable strategy from PST (§ 4.2.1) and LPI (§ 4.2.2).

After collecting the candidate bugs, we reduce the sizes of the bug-revealing formulas via delta debugging [63]. We have automated the test case reduction, using ddSMT [2] and pyDelta [3], two open-source delta debuggers for the SMT-LIB2 language. Finally, we contact the solver developers to confirm the bugs.

Algorithm 2: Testing SMT solvers via approximation.

```
Input: A set of seed formulas $Seeds$ and an SMT solver $S$
Output: The candidate bugs
Procedure test_smt_solver($Seeds, S$)
    $soundness\_bugs \leftarrow \emptyset, model\_bugs \leftarrow \emptyset, crash\_bugs \leftarrow \emptyset$;
    while some budget is not reached do
        $\varphi \leftarrow$ randomly select a formula from $Seeds$;
        $\varphi \leftarrow$ transform $\varphi$ into CNF;
        $res \leftarrow$ solve $\varphi$ with the solver $S$;
        /* generate $n$ equi-satisfiable mutants */
        for $i = 1$ to $n$ do
            if $res ==$ "sat" then
                $\varphi' \leftarrow$ over_approximate($\varphi$);
            else
                $\varphi' \leftarrow$ under_approximate($\varphi$);
            $res' \leftarrow$ solve $\varphi'$ with the same solve $S$;
            if $res' \neq res$ then
                $soundness\_bugs \leftarrow soundness\_bugs \cup \{\varphi'\}$;
            else if $res' ==$ "sat" then
                $M \leftarrow$ a model of $\varphi'$ returned by $S$;
                if $M$ does not satisfy $\varphi'$ then
                    $model\_bugs \leftarrow model\_bugs \cup \{\varphi'\}$;
            else if $S$ crashed then
                $crash\_bugs \leftarrow crash\_bugs \cup \{\varphi'\}$;
    return $soundness\_bugs \cup model\_bugs \cup crash\_bugs$;
```

5 EVALUATION

To evaluate the effectiveness of skeletal approximation enumeration, we conduct two sets of experiments. In the first experiment, we examine the effectiveness of Sparrow in finding bugs in Z3 and CVC4. In the second experiment, we compare Sparrow with two existing techniques [59, 60] regarding performance, code coverage, and bug finding.

5.1 Experimental Setup

Tested Solvers. We have selected Z3 [28] and CVC4 [11], the two most popular SMT solvers for the experimental evaluation. We choose the solvers by following four criteria. First, they have been widely used in both academia and industry. Second, they support most of the theories in the SMT-LIB2 standard [12]. Third, they show a state-of-the-art performance, i.e., regularly achieve high ranks in SMT-COMP [4], the annual SMT competitions. Finally, they are mature and have been extensively tested by previous works [15, 19, 39, 48, 59, 60], which means finding their bugs is challenging.

We mainly focus on testing the default modes of the solvers. For CVC4, we use the options `--produce-models`, `--incremental` and `--strings-exp` as needed to support all the seeds. To detect invalid model bugs, we have supplied the `--check-models` option to CVC4 and the `model.validate=true` option to Z3.

Seeds Selection. The seed formulas come from two sources: (1) the regression test suits of several open-source SMT solvers, including Z3, CVC4, Yices2, and OpenSMT [5–8], and (2) the SMT-LIB2 standard benchmark suite maintained by the SMT-LIB Initiative [12]. After collecting the formulas, we preprocess them as follows. First, we use Z3's "`simplify`" tactic to filter out formulas that are trivially satisfiable (e.g., $p \vee \neg p$) or unsatisfiable (e.g., $1 > 2$). These formulas can often be instantly solved in the simplification phase of SMT solvers, which may make our mutations futile. Next, we exclude formulas that cannot be solved by Z3 and CVC4 within 5 seconds, to improve the testing throughput.

Environment. All the experiments are conducted on a Linux workstation with an 80 Core Intel(R) Xeon(R) 2.20GHz processor and 256 GB RAM. We compile Z3 and CVC4 using gcc-5.4.0, with assertions and AddressSantizer [49] enabled. We use Gcov [52] to measure the code coverage. All the tools are set to run in single-threaded mode. For each of the experiments, we perform ten independent runs and report the average results.

5.2 Results of Bug Finding

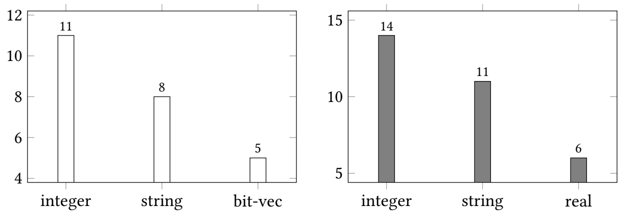
In this section, we present some statistical analyses of the bugs found by Sparrow. All the bug reports are publicly available at the site in [9].

Bug Count. Table 3 summarizes the status of the bugs. "Reported" represents the number of reported bugs; "Confirmed" represents the bugs that the developers confirm as real and unique; "Fixed" represents the fixed bugs; "Duplicate" represents the bugs that the developers identify as duplicates; and "Won't fix" represents the bugs that the developers reject to fix. Overall, Sparrow finds 71 previously unknown, unique, and confirmed bugs in Z3 and CVC4, which are missed by the solver developers, users, and regression testing. By the time of writing, 77.5% (55 out of 71) of the confirmed bugs had already been fixed.

Won't fix Bugs. Some bugs are marked as "won't fix" mainly due to miss-configurations, i.e., improper options are supplied to the solver. For example, in a Z3 bug report, the developer commented that "*I am going to skip these bugs on strings tweaking strange configuration parameters*".

Affected Theories. Sparrow can find bugs in different SMT-LIB2 theories, such as integers, reals, bit-vectors, floating points, strings, and the combinations of these theories. Figure 1 presents the distribution of logic types among the confirmed bugs. Among the top-3 most frequent theories in Z3 are integers, strings, and bit-vectors. Among the top-3 most frequent theories in CVC4 are integers, strings, and reals. We observe that most of the bug-triggering integer formulas are non-linear. The results indicate that decision procedures for non-linear integer arithmetic and strings are among the weak components in SMT solvers.

Bug Types. Table 4 shows the distribution of bug types among the confirmed bugs. The most common bug category is crash bugs (41 out of 71), followed by invalid model bugs (22) and soundness bugs (8). In summary, 42.3% (30 out of 71) of the confirmed bugs are correctness bugs (including soundness bugs and invalid model



(a) Z3 (b) CVC4

Figure 1: Top-3 theory type of the confirmed bugs.

Table 3: Status of the bugs found by Sparrow.

Status	Z3	CVC4	Total
Reported	38	46	84
Confirmed	29	42	71
Fixed	25	30	55
Duplicate	2	2	4
Won't fix	7	2	9

Table 4: Bug type of the confirmed bugs.

Type	Z3	CVC4	Total	Fixed
Soundness	4	4	8	7
Invalid model	10	12	22	16
Crash	15	26	41	32

bugs), which clearly demonstrates the strength of Sparrow in finding logic issues in SMT solvers.

Feedback of Developers. The developers of the solvers are generally responsive in fixing our bug reports, which indicates that they take our bugs seriously. For example, to quote the developers' comments, "*This is due to a fairly obscure extended equality rewrite.*" "*Thanks a lot for the report. It turns out that the issue is a bit less severe (though still pretty bad) than I first thought.*" As shown in Table 4, among the 55 fixed bugs, 23 (41.8%) are correctness bugs. Besides, the developers have added the bug-triggering formulas to the regression test suites of their solvers.

Taken together, we conclude that Sparrow is effective in finding a large number of diverse bugs and its findings are significant.

5.3 Comparison to Existing Techniques

In this section, we present an in-depth study of the mutation strategies in Sparrow, by comparing the following techniques:

- YinYang(Fusion): a metamorphic testing based approach, where the metamorphic relation is based on the semantic fusion strategy [60];
- YinYang(OpFuzz): a differential testing based approach that uses the type-aware operator mutation strategy [59] for the input generation;
- Sparrow(PST): the variant of Sparrow that only applies the predicate symbol transformation strategy (§ 4.2.1);

Table 5: Time (milliseconds) of generating a mutant.

Tool	Avg	Min	Max	StdDev
YinYang(Fusion)	4.5	1.9	78	3.8
YinYang(OpFuzz)	0.6	0.3	2.4	0.6
Sparrow(PST)	0.4	0.2	1.1	0.2
Sparrow(LPI)	1.3	0.6	33	2.4
Sparrow(PST+LPI)	0.7	0.2	3.2	0.5

Table 6: Number of tested mutants per hour.

Tool	#Mutants
YinYang(Fusion)	10,582
YinYang(OpFuzz)	34,417
Sparrow(PST)	33,736
Sparrow(LPI)	26,676
Sparrow(PST+LPI)	32,365

- Sparrow(LPI): the variant of Sparrow that only uses the predicate symbol transformation strategy (§ 4.2.2);
- Sparrow(PST+LPI): the default configuration of Sparrow.

We choose YinYang(Fusion) and YinYang(OpFuzz) because they respectively represent the state-of-the-arts in the oracle-guided approach and the differential testing approach (§ 1).

We perform three experiments: (1) measuring the performance of the tools, (2) comparing the line coverage of the solvers, and (3) applying Sparrow to reproduce the bugs detected by YinYang. The three experiments offer a comprehensive comparison between the tools. To study (1) and (2), we randomly sample 1000 seed formulas, and set each tool to generate 300 mutants per seed, following the settings in [59]. Note that since YinYang(Fusion) [60] supports fewer theories than YinYang(OpFuzz) [59] and Sparrow (c.f., Table 1), we only sample seeds supported by all the tools. The timeout for the solvers is set to 10 seconds per mutant.

Performance. First, we compare the performance of the five mutation strategies, in terms of the mutation cost and the overall testing throughput.

Mutation Cost. Table 5 summarizes the statistics of the time cost. For each tool, we report the average, minimum, maximum, and standard deviation of the time for generating one mutant. We make two observations. First, the time cost of Sparrow to derive satisfiability-preserving mutants is low. On average, it takes the three variants of Sparrow 0.4 to 1.3 milliseconds to generate one mutant. The speed of Sparrow(PST+LPI) lies between Sparrow(PST) and Sparrow(LPI). Recall that PST only mutates the predicate symbols, while LPI needs to generate a new formula snippet. Thus, Sparrow(PST) is often faster than Sparrow(LPI). Second, the mutation speed of Sparrow(PST+LPI) is similar to YinYang(OpFuzz), and is about 6× faster than YinYang(Fusion).

Testing Throughput. To give a picture of the overall testing throughput, Table 6 presents the number of tested mutants per hour. As can be seen, YinYang(Fusion), YinYang(OpFuzz), and Sparrow(PST+LPI) can test 10,582, 34,217, and 32,376 mutants in one hour, respectively.

Table 7: Line coverage by mutating 1000 seeds (300 mutants per seed). The baseline is the coverage of Z3 and CVC4 after solving 1000 seeds (13.5% for Z3 and 8.7% for CVC4).

Tool	Z3	CVC4
YinYang(Fusion)	17.3%	11.3%
YinYang(OpFuzz)	19.8%	14.2%
Sparrow(PST)	18.1%	13.3%
Sparrow(LPI))	27.2 %	14.6%
Sparrow(PST+LPI)	27.9%	16.1%

We observe that most (> 98%) of the CPU time is taken up by the SMT solvers. Therefore, the hardness of the mutant formulas is the key factor of the throughput. For example, we find that the mutants generated by YinYang(Fusion) are often harder to solve than other tools. Thus, the throughput of YinYang(Fusion) is smaller than YinYang(OpFuzz) and Sparrow(PST+LPI).

To summarize, the mutation cost and testing throughput of Sparrow(PST+LPI) are competitive against that of YinYang(OpFuzz).

Line Coverage. Second, we compare the line coverage improvement achieved by the fuzz tools. The baseline is the line coverage of Z3 and CVC4 after solving the 1000 sampled seeds, which are 13.5% and 8.7%, respectively. We then use the fuzz tools to mutate these seeds, run Z3 and CVC4 to solve the mutants, and measure the solvers' cumulative line coverage. For each seed, we run each tool to produce 100 mutants. Table 7 presents the comparison results. As can be seen, Sparrow(PST+LPI) can consistently improve the line coverage of the solvers over YinYang. Sparrow improves the coverage over the baseline by 14.4% for Z3 and 7.4% for CVC4, while the best one between YinYang(Fusion) and YinYang(OpFuzz) does by 6.3% for Z3 and 5.5% for CVC4.

Reproducing Bugs. Finally, we conduct an experiment to compare YinYang and Sparrow in terms of finding correctness bugs. In particular, we first select all the correctness bugs that are found by YinYang(Fusion) and YinYang(OpFuzz) in November and December 2020, and fixed by the developers by the time of writing. We then try to reproduce the bugs using Sparrow. We choose the correctness bugs because, as a metamorphic testing approach, Sparrow's primary goal is to find those issues. The selected bugs cover various theories such as integers, reals, bit-vectors, and strings.

For each of the bugs, we first remove the seeds that can directly reveal the bug (without being mutated). We then use Sparrow to derive mutants from all the seed formulas with the identical logic type.[1] We set the number of mutants per seed to 300 and repeat the generation process 100 times. Then, the comparison proceeds as follows. We first check whether Sparrow can generate re-triggering mutants from the seeds or not. If yes, we then check whether Sparrow actually re-triggers the same bug. To achieve this goal, we run the bug-revealing mutants against the first solver commit ID with the corresponding fix. If the solver now answers correctly for the mutants, we count the bug as successfully re-triggered.

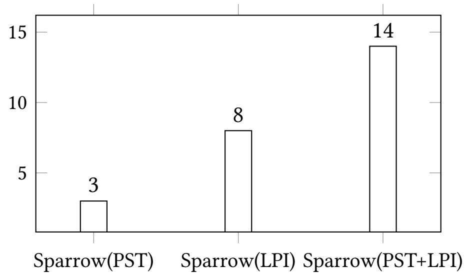


Figure 2: The results of running Sparrow to re-trigger the 19 fixed correctness bugs reported by YinYang(Fusion) and YinYang(OpFuzz) in November and December 2020.

Figure 2 presents the reproduction results. Briefly, we make two observations. First, Sparrow(PST) and Sparrow(LPI) can reproduce 3 bugs and 8 bugs alone, respectively. But there are 7 bugs that can only be found by Sparrow(PST+LPI), i.e., the combination of the two strategies. Second, Sparrow(PST+LPI) successfully reproduces 73.7% (14 out of 19) the bugs. There are 5 bugs that cannot be reproduced, because our current implementation has limited support for the theories of recursive functions and abstract data types.

In summary, we find that (1) Sparrow is competitive against or complementary to YinYang(Fusion) and YinYang(OpFuzz) in finding correctness bugs and (2) both of the two mutation strategies in Sparrow contribute to its effectiveness.

5.4 Threats to Validity

The threat to internal validity mainly lies in the implementation of our approach. To validate our implementation, we have used several SMT solvers to cross-check if the mutants generated by Sparrow are indeed the over- or under-approximations of the seed formula. This validates the implementation to some extent. The threat to external validity lies in the representativeness of the subjects. The solvers we select for the evaluation are mature, widely-used, and extensively tested by previous works [15, 19, 39, 59, 60]. The threat to construct validity is the selection of the seed formulas and the randomness of the mutations. To mitigate the threat, we run each experiment ten times and use the average data.

6 DISCUSSION

Limitations of Sparrow. Our study demonstrates Sparrow's effectiveness for testing SMT solvers, but limitations exist in our current implementation. First, for the predicate symbol transformation strategy (§ 4.2.1), Sparrow relies on manually given rules to transform a literal. In the future, it would be interesting to synthesize new transformation rules automatically. For example, the CVC4 developers have applied syntax-guided synthesis (SyGuS) to generate term-level and equivalence-preserving rewriting rules [43]. Second, for the live predicate injection strategy (§ 4.2.2), Sparrow can suffer from performance issues if it attempts to generate a large formula snippet. However, this limitation does not mean that Sparrow can only generate small mutants, because it can mutate a seed "incrementally", i.e., approximates the mutants produced in the previous rounds. Third, Sparrow has limited support for a few

[1] To offer an apples-to-apples comparison, we should have used the same seed for each bug. However, we cannot know which seed YinYang has used to trigger a bug.

logics such as recursive functions and abstract data types, which are recently introduced into the SMT-LIB2.6 standard.

Generality of SAE. Beyond SMT solver testing, skeletal approximation enumeration suggests a general strategy for deriving semantic approximations of a problem via (lightweight) syntactical mutations. Specifically, it can be profitable to transform a seed into some suitable representations and then design the mutation operators. There are several avenues for further exploring the applicability of skeletal approximation enumeration. First, the techniques and tools can facilitate testing other SMT solvers that take SMT-LIB2 files as their input. Second, the general idea could be extended to test other software systems that reason about programs' logic properties, such as static analyzers and program verifiers.

Future Work on Automated Debugging. While the focus of this paper is bug detection, it could be promising to use our techniques to ease the automatic debugging. First, a possible future work is to aid delta debugging. Specifically, we can trace the mutations made by Sparrow, and perform some backtracking of the mutations in the stage of delta debugging. The backtracking-based strategy could assist or complement existing general-purpose delta debuggers. Second, showing the minimal differences to seed formulas that trigger bugs is helpful for the developers to understand the bugs. Thus, providing the minimal literal-level mutations and the original seed in the bug report would be another potential direction to assist debugging.

7 RELATED WORK

SMT Solver Testing. FuzzSMT [19] is the first grammar-based fuzzing tool for SMT solvers. StringFuzz [15] and BanditFuzz [48] follow the idea to test string and floating points solvers, respectively. Winterer et al. [59] present a type-aware mutation strategy, which mutates operators of conforming types within the seed formulas to generate well-typed mutants. Falcon [61] explores the combined formula-configuration space for testing SMT solvers. All the above-mentioned techniques need to combine differential testing to find soundness bugs. To address the test oracle problem, several recent works generate SMT formulas whose satisfiability is known by construction [22, 39, 60], which we term the oracle-guided approach. Bugariu and Müller [22] propose an approach to generating increasingly complex string formulas via satisfiability-preserving transformations. Semantic fusion [60] fuses formula pairs that generate mutants that are by construction either satisfiable or unsatisfiable. However, their implementations only support integers, reals, and strings. [2] Storm [39] mutates the Boolean structure of a seed but can only generate satisfiable mutants. Compared to the previous works, we present a new technique for the oracle-guided approach, which applies to all theories and can generate satisfiable and unsatisfiable mutant formulas.

Metamorphic Testing. The key idea of metamorphic testing [25] is to detect violations of domain-specific metamorphic relations by comparing the outputs between a seed test and its corresponding mutant tests. Metamorphic testing has been successfully applied in many application domains such as bioinformatics [26], web services [24], compilers [36, 51], debuggers [53], databases [47], machine learning-based systems [31, 38], model counters [55], and SMT solvers [60].

Our approach is an instance of metamorphic testing. A closely related work is semantic fusion [60], which generates equi-satisfiable mutants from the concatenation of two seed formulas. Our approach differs in two aspects. First, semantic fusion mutates variables using the fusion functions, which can only introduce new variables and function symbols in the mutants. In comparison, skeletal approximation enumeration can inject formula snippets, which can contain new variables, function symbols, predicate symbols, and Boolean connectives. Second, semantic fusion requires that the two seeds are both satisfiable or unsatisfiable. In comparison, our algorithm does not assume that the satisfiability of the seeds is known prior, as it determines the mutation strategy according to the solving result of the SMT solver under test.

Mutation-based Testing. A common technique for input generation is to mutate the seed corpus. For example, American fuzzy lop (AFL) [1] is a well-known security-oriented fuzzer, which employs bit-level and byte-level mutations to generate new test cases. However, such an efficient input generation approach cannot handle inputs with a highly formatted structure or grammar. Thus, grammar-aware mutation-based fuzzing has been proposed. Superion [56], AFLSmart [46], and Nautulius [10] are general grammar-aware grey-box fuzzers that employ AST-based mutations, and use code coverage to guide the mutations. CodeAlchemist [34] preserves the semantic requirement, e.g., type correlation, as the constraint during input generation. Zest [44] combines the coverage feedback with property-based testing to provide better guidance for seed prioritization. In comparison, our approach can be regarded as an instance of grammar-aware mutation. Specifically, Sparrow not only generates syntactically correct mutants but also guarantees their satisfiability results, which can serve as the ground truth for finding correctness bugs.

8 CONCLUSION

This paper presents skeletal approximation enumeration, a new methodology for testing SMT solvers. Our approach helped discover 71 confirmed bugs in Z3 and CVC4, two state-of-the-art and comprehensively tested SMT solvers. More than 50 of the bugs have been fixed, and a significant fraction of them are correctness bugs. Our technique is general and may be adapted to other constraint languages (such as Datalog and MiniZinc) and settings (such as static analyzers and model checkers).

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. We also appreciate the developers of Z3 and CVC4 for discussing and addressing our bug reports. Rongxin Wu is supported by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (BK20202001) and NSFC61902329. Other authors are supported by the RGC16206517 and ITS/440/18FP grants from the Hong Kong Research Grant Council, Ant Group through ant Research Program, and the donations from Microsoft and Huawei. Heqing Huang is the corresponding author.

[2] Note that, in theory, it could be possible to extend the idea of semantic fusion to support other theories.

REFERENCES

[1] 2014. AFL: american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: 2014.

[2] 2021. ddSMT. https://github.com/aniemetz/ddSMT.

[3] 2021. pyDelta. https://github.com/nafur/pydelta.

[4] 2021. SMT-COMP. https://smt-comp.github.io/.

[5] 2021. Z3 test scripts. https://github.com/Z3Prover/z3/src/test.

[6] 2021. CVC4 regression test suit. https://github.com/CVC4/CVC4/test/.

[7] 2021. Yices2 regression test suit. https://github.com/SRI-CSL/yices2/tests.

[8] 2021. OpenSMT regression test suit. https://github.com/usi-verification-and-security/opensmt/regression.

[9] 2021. https://smtfuzz.github.io/.

[10] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[11] Clark Barrett, Christopher Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 171–177. http://dl.acm.org/citation.cfm?id=2032305.2032319

[12] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The satisfiability modulo theories library (SMT-LIB). www. *SMT-LIB. org* 15 (2010), 18–52.

[13] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. 2017. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA (LIPIcs, Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:12. https://doi.org/10.4230/LIPIcs.SNAPL.2017.1

[14] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 643–659. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko

[15] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A Fuzzer for String Solvers. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10982)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 45–51. https://doi.org/10.1007/978-3-319-96142-2_6

[16] Cristina Borralleras, Daniel Larraz, Enric Rodríguez-Carbonell, Albert Oliveras, and Albert Rubio. 2019. Incomplete SMT Techniques for Solving Non-Linear Formulas over the Integers. *ACM Trans. Comput. Log.* 20, 4 (2019), 25:1–25:36. https://doi.org/10.1145/3340923

[17] Aaron R. Bradley and Zohar Manna. 2007. *The calculus of computation - decision procedures with applications to verification*. Springer. https://doi.org/10.1007/978-3-540-74113-8

[18] Robert Brummayer and Armin Biere. 2009. Effective Bit-Width and Under-Approximation. In *Computer Aided Systems Theory - EUROCAST 2009, 12th International Conference, Las Palmas de Gran Canaria, Spain, February 15-20, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5717)*, Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia (Eds.). Springer, 304–311. https://doi.org/10.1007/978-3-642-04772-5_40

[19] Robert Brummayer and Armin Biere. 2009. Fuzzing and Delta-Debugging SMT Solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories* (Montreal, Canada) *(SMT '09)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/1670412.1670413

[20] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. 2010. The OpenSMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6015)*, Javier Esparza and Rupak Majumdar (Eds.). Springer, 150–153. https://doi.org/10.1007/978-3-642-12002-2_12

[21] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady. 2007. Deciding Bit-Vector Arithmetic with Abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4424)*, Orna Grumberg and Michael Huth (Eds.). Springer, 358–372. https://doi.org/10.1007/978-3-540-71209-1_28

[22] Alexandra Bugariu and Peter Müller. 2020. Automatically testing string solvers. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1459–1470. https://doi.org/10.1145/3377811.3380398

[23] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[24] W. K. Chan, S. C. Cheung, and Karl R. P. H. Leung. 2005. Towards a Metamorphic Testing Methodology for Service-Oriented Software Applications. In *Fifth International Conference on Quality Software (QSIC 2005), 19-20 September 2005, Melbourne, Australia*. IEEE Computer Society, 470–476. https://doi.org/10.1109/QSIC.2005.67

[25] 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).

[26] Tsong Yueh Chen, Joshua Wing Kei Ho, Huai Liu, and Xiaoyuan Xie. 2009. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinform.* 10 (2009). https://doi.org/10.1186/1471-2105-10-24

[27] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. 2012. SMTInterpol: An Interpolating SMT Solver. In *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7385)*, Alastair F. Donaldson and David Parker (Eds.). Springer, 248–254. https://doi.org/10.1007/978-3-642-31759-0_19

[28] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[29] David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (May 2005), 365–473. https://doi.org/10.1145/1066100.1066102

[30] Bruno Dutertre. 2014. Yices2.2. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, Berlin, Heidelberg, 737–744. https://doi.org/10.1007/978-3-319-08867-9_49

[31] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 118–128. https://doi.org/10.1145/3213846.3213858

[32] Vijay Ganesh and David L Dill. 2007. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany) *(CAV'07)*. Springer-Verlag, Berlin, Heidelberg, 519–531. http://dl.acm.org/citation.cfm?id=1770351.1770421

[33] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[34] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/codealchemist-semantics-aware-code-generation-to-find-vulnerabilities-in-javascript-engines/

[35] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1613–1627. https://doi.org/10.1109/SP40000.2020.00063

[36] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 216–226. https://doi.org/10.1145/2594291.2594334

[37] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. 2014. Symbolic Optimization with SMT Solvers. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 607–618. https://doi.org/10.1145/2535838.2535857

[38] Pingchuan Ma, Shuai Wang, and Jin Liu. 2020. Metamorphic Testing and Certified Mitigation of Fairness Violations in NLP Models. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, Christian Bessiere (Ed.). ijcai.org, 458–465. https://doi.org/10.24963/ijcai.2020/64

[39] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting critical bugs in SMT solvers using blackbox mutational

fuzzing. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 701–712. https://doi.org/10.1145/3368089.3409763

[40] Kenneth L. McMillan. 2006. Lazy Abstraction with Interpolants. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4144)*, Thomas Ball and Robert B. Jones (Eds.). Springer, 123–136. https://doi.org/10.1007/11817963_14

[41] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[42] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014. Boolector 2.0. *J. Satisf. Boolean Model. Comput.* 9, 1 (2014), 53–58. https://doi.org/10.3233/sat190101

[43] Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark W. Barrett, and Cesare Tinelli. 2019. Syntax-Guided Rewrite Rule Enumeration for SMT Solvers. In *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11628)*, Mikolás Janota and Inês Lynce (Eds.). Springer, 279–297. https://doi.org/10.1007/978-3-030-24258-9_20

[44] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 329–340. https://doi.org/10.1145/3293882.3330576

[45] Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. 2016. Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 23–41. https://doi.org/10.1007/978-3-319-41540-6_2

[46] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* (2019), 1–17. https://doi.org/10.1109/TSE.2019.2941681

[47] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428279

[48] Joseph Scott, Federico Mora, and Vijay Ganesh. 2020. BanditFuzz: A Reinforcement-Learning Based Performance Fuzzer for SMT Solvers. In *Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20-21, 2020, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12549)*, Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel (Eds.). Springer, 68–86. https://doi.org/10.1007/978-3-030-63618-0_5

[49] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[50] Armando Solar-Lezama and Rastislav Bodik. 2008. *Program synthesis by sketching.* Citeseer.

[51] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 849–863. https://doi.org/10.1145/2983990.2984038

[52] G Team. 2014. Gcov-using the gnu compiler collection (gcc). *Online, disponıvel em http://gcc. gnu. org/onlinedocs/gcc/Gcov. html-Ultimo acesso em* 26, 02 (2014), 2015.

[53] Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. 2019. Interactive metamorphic testing of debuggers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 273–283. https://doi.org/10.1145/3293882.3330567

[54] Grigori S Tseitin. 1983. On the complexity of derivation in propositional calculus. In *Automation of reasoning*. Springer, 466–483. https://doi.org/10.1007/978-3-642-81955-1_28

[55] Muhammad Usman, Wenxi Wang, and Sarfraz Khurshid. 2020. TestMC: Testing Model Counters using Differential and Metamorphic Testing. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 709–721. https://doi.org/10.1145/3324884.3416563

[56] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[57] Elaine J Weyuker. 1982. On testing non-testable programs. *Comput. J.* 25, 4 (1982), 465–470. https://doi.org/10.1093/comjnl/25.4.465

[58] Virginie WIELS, Robert Delmas, David Doose, Pierre-Loïc Garoche, J. Cazin, and Guy Durrieu. 2012. Formal Verification of Critical Aerospace Software. *AerospaceLab* 4 (May 2012), p. 1–8. https://hal.archives-ouvertes.fr/hal-01184099

[59] 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 (2020), 193:1–193:25. https://doi.org/10.1145/3428261

[60] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 718–730. https://doi.org/10.1145/3385412.3385985

[61] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Fuzzing SMT Solvers via Two-Dimensional Input Space Exploration. In *ISSTA'21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA.*

[62] Cunxi Yu, Maciej J. Ciesielski, and Alan Mishchenko. 2018. Fast Algebraic Rewriting Based on And-Inverter Graphs. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 37, 9 (2018), 1907–1911. https://doi.org/10.1109/TCAD.2017.2772854

[63] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why? 1687 (1999), 253–267. https://doi.org/10.1007/3-540-48166-4_16