Automatically testing string solvers

Conference Paper

Author(s):
Bugariu, Alexandra; Müller, Peter

Publication date:
2020-06

Permanent link:
https://doi.org/10.3929/ethz-b-000397450

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
https://doi.org/10.1145/3377811.3380398

Automatically Testing String Solvers

Alexandra Bugariu
Department of Computer Science, ETH Zurich
Zurich, Switzerland
alexandra.bugariu@inf.ethz.ch

Peter Müller
Department of Computer Science, ETH Zurich
Zurich, Switzerland
peter.mueller@inf.ethz.ch

ABSTRACT

SMT solvers are at the basis of many applications, such as program verification, program synthesis, and test case generation. For all these applications to provide reliable results, SMT solvers must answer queries correctly. However, since they are complex, highly-optimized software systems, ensuring their correctness is challenging. In particular, state-of-the-art testing techniques do not reliably detect when an SMT solver is unsound.

In this paper, we present an automatic approach for generating test cases that reveal soundness errors in the implementations of string solvers, as well as potential completeness and performance issues. We synthesize input formulas that are satisfiable or unsatisfiable by construction and use this ground truth as test oracle. We automatically apply satisfiability-preserving transformations to generate increasingly-complex formulas, which allows us to detect many errors with simple inputs and, thus, facilitates debugging.

The experimental evaluation shows that our technique effectively reveals bugs in the implementation of widely-used SMT solvers and applies also to other types of solvers, such as automata-based solvers. We focus on strings here, but our approach carries over to other theories and their combinations.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

automatic testing, soundness testing, string solvers, SMT solvers

ACM Reference Format:
Alexandra Bugariu and Peter Müller. 2020. Automatically Testing String Solvers. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380398

1 INTRODUCTION

SMT solvers have a large variety of applications, from program verification and synthesis to symbolic execution and concolic testing. For all these tools to be reliable and usable in practice, the SMT solvers have to provide *correct answers.*

For a given formula, an SMT solver returns either *sat* (together with a *model* that associates a value to each free variable, such that the formula evaluates to true), or *unsat* (and optionally a set of clauses that lead to a contradiction—the *unsat core*). A solver is *unsound* if it yields an incorrect result, that is, yields sat for an unsatisfiable formula or unsat for a satisfiable one. It is also unsound if it correctly yields sat or unsat, but produces an *invalid model* or an *incorrect unsat core*, i.e., a sub-formula that is satisfiable.

Since SMT solvers support undecidable theories, they cannot always determine whether a formula is satisfiable and may sometimes return *unknown*. However, a solver is *unnecessarily incomplete* if it returns *unknown* for a formula from a decidable theory, such as $\forall\, x, y : Int : x = y$, which falls into Presburger arithmetic. It is also undesirable for a solver to *timeout* (i.e., not to solve a query within a given timespan). Such a result often points to performance issues.

SMT solvers combine multiple communicating decision procedures for various theories (e.g., uninterpreted functions, linear/non-linear arithmetic, bit vectors, arrays, strings, etc.) As a result, they are complex software systems, and checking that their implementations are *sound* is, therefore, challenging.

To illustrate the errors solvers can make, consider for example the SMT formula from List. 1. It uses two string variables, `t` and `u`, and checks if it is possible to obtain the constant string ”*a*” by replacing the first occurrence of `t` by `u` in an empty string. This is the case according to the SMT-LIB standard [12] if `t` is empty and `u` is ”*a*”. Nevertheless, Z3-seq [7, 15] and Z3str3 [13], two widely-used SMT solvers, incorrectly report unsat for this formula.

```
(declare-fun t () String)
(declare-fun u () String)
(assert (= (str.replace "" t u) "a"))
(check-sat)
```

Listing 1: A sat formula that exposes a soundness bug in Z3-seq and Z3str3, written in SMT-LIB syntax (with prefix notation for operators). In the rest of the paper, we show examples in mathematical notation, to improve readability.

State-of-the-art testing techniques for SMT solvers do not reliably detect such errors. Fuzzing [16] generates formulas that may crash the solvers or reveal performance issues, but do not reliably detect soundness problems. Approaches based on differential testing [28] compare the results of different solvers. Different results may indicate a soundness problem in one of them. However, determining which solver is at fault requires manual effort. Moreover, differential testing requires at least one solver that provides the correct result; this might not be the case in situations like the one described above, where Z3-seq and Z3str3 are both incorrect.

In this paper, we propose a novel technique for automatically generating test cases that reveal, besides others, soundness issues in the implementation of SMT solvers. We synthesize input formulas

ICSE '20, May 23–29, 2020, Seoul, South Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380398

that are sat or unsat by construction and use this ground truth as test oracle. Our technique generates sequences of input formulas of increasing complexity by applying satisfiability-preserving transformations. In this way, the bugs are often found with simple inputs, which facilitates error localization and debugging. We automatically construct a model for each satisfiable formula and a minimal, unique unsat core for each unsatisfiable one, and use them as additional oracles. For concreteness, this paper focuses on string solvers, but our technique generalizes to other theories.

Contributions. Our contributions are the following:

- We present an automated approach for synthesizing SMT formulas for the string theory, which are satisfiable or unsatisfiable by construction. Together with the known ground truth, these formulas are used to automatically test the implementations of SMT solvers. Our technique generates satisfiable formulas together with models, and unsatisfiable formulas together with unsat cores; as they are incrementally complex, these formulas facilitate debugging and faster error localization.
- We implemented our technique and evaluated it on three widely-used SMT solvers, Z3-seq [7, 15], Z3str3 [13], and CVC4 [25], as well as on the automata-based solver MT-ABC [11]. Our experimental results show that our technique effectively detects soundness problems, and outperforms state-of-the-art fuzzing techniques in doing so. Our approach can also reveal other types of errors, such as performance, completeness or precision issues.

Outline. The rest of this paper is organized as follows: In Sect. 2 we give an overview of our solution for constructing sat and unsat formulas; the details follow in Sect. 3. We discuss our experimental results in Sect. 4, related work in Sect. 5 and conclude in Sect. 6.

2 OVERVIEW

Our approach automatically generates SMT formulas that are satisfiable or unsatisfiable by construction. These formulas are used as inputs for black-box tests, while the ground truth is used as test oracle. Our formula construction approach consists of two steps:

(1) We generate simple formulas with known truth values.
(2) From these formulas, we derive more complex, equisatisfiable formulas through automatic transformations.

To perform these steps, our generator requires a set of operations supported by the theory under test, together with their *reference semantics.* For concreteness, in this paper we use the SMT-LIB standard [12] as the reference semantics, because it provides a rigourous description of the theories. Most widely-used SMT solvers adhere to SMT-LIB to facilitate comparisons (e.g., in SMT competitions [4]) and to enable the side-by-side usage of multiple solvers [22].

Our technique tests whether the implementation of an SMT solver complies with the provided reference semantics. For solvers that *intentionally* deviate from the SMT-LIB standard, it is straightforward to parameterize our technique with an alternative reference semantics and use that to test the implementation. For instance, the CVC4 documentation does not define the result of the `replace` operation when the second argument is the empty string [1]. One can use our technique with the SMT-LIB semantics to check for such deviations from the standard, and with an alternative reference semantics to check for errors in the implementation.

Table 1: String operations, grouped by their return type

Return a string	Return an integer	Return a boolean
`at`(s, *off*)*	`indexOf`(s, t, *off*)	`contains`(s, t)
`concat`(s, t)	`length`(s)	`equals`(s, t)
`intToStr`(n)	`strToInt`(s)	`prefixOf`(s, t)
`replace`(s, t, u)		`suffixOf`(s, t)
`substr`(s, *off*, *len*)		

*returns a char (i.e., a string of length 1) s, t, u: type String; n, *off*, *len*: type Int

`replace`(s, t, u) = *res*

Figure 1: The simple sat formula for `replace`, generated in step 1. All the variables have type String.

Tab. 1 gives an overview of the operations supported by the string theory; these operations have, according to SMT-LIB, deterministic semantics [5]. As some of the operations take or yield integers, reasoning about them also involves linear integer arithmetic.

In the following subsections, we give an overview of our construction approach for satisfiable and unsatisfiable formulas.

2.1 Generating satisfiable formulas

An SMT formula is *satisfiable* (sat) with respect to some background theory if there exists at least one model (i.e., variable assignment) within the theory such that the formula evaluates to true [14]. For example, the formula $x + x = 3$ is satisfiable in the theory of real numbers, as it is possible to find at least one solution to this equation ($x = 1.5$). Nonetheless, the formula is *unsatisfiable* (i.e., does not have a model) in the integer arithmetic theory.

In the first step of our technique, we construct sat formulas that test each operation in isolation; this allows developers to localize and fix bugs faster. We start with the operations supported by the theory under test and automatically derive a test case for each of them, in which the parameters and the result of each operation are unconstrained. These simple formulas are thus trivially satisfiable.

An example formula that we synthesize during the first step is shown in Fig. 1. This formula is satisfiable, since for all string arguments `s`, `t`, `u`, there exists a string `res` that is equal to the result of the `replace` operation (because all string operations are total functions). Even though they are very simple, these formulas can still reveal bugs. For example, Z3-seq returns unknown for the SMT formula from Fig. 2, which tests the `indexOf` operation. Since this initial test case is minimal (and, in particular, does not involve any other operations), it facilitates identifying the source of this incompleteness: most likely a bug in handling the `indexOf` operation in the corresponding decision procedure.

To test more complex cases, as well as the interaction between different operations, step 2 of our approach derives additional test cases by automatically applying a set of transformations on the formulas synthesized before. These transformations preserve the satisfiability of the initial formulas, thus creating *equisatisfiable*, but more complex formulas, with more constrained models and more (or more complex) terms. We illustrate a very simple transformation here and present more complex ones in Sect. 3.1.

$$\texttt{indexOf}(s, t, \mathit{off}) = \mathit{res}$$

Figure 2: A simple sat formula generated in step 1 that uncovers an incompleteness in Z3-seq. *off* **and** *res* **have type Int,** *s* **and** *t* **have type String.**

For the simple formulas from Fig. 1 and Fig. 2, the solver can construct arbitrarily many models, as all the variables are unconstrained. We can strengthen the formulas by adding constraints on the values of these variables. A possible transformation is to replace some of the variables by constants. To decide what values can be assigned to which variables such that the complex formula is still satisfiable, we rely on concrete execution. That is, we implement an *executable* version of the reference semantics for the operations under test and use it to determine valid parameters and results. This technique can be applied to sat formulas, because finding *a* model for which the formula holds is enough for proving its satisfiability.

Having the executable semantics, we can evaluate each operation on concrete arguments. In this way, we obtain formulas in which all the arguments and the results are constants. The test formulas are then synthesized by fixing some of them to the constants used in the concrete execution, and leaving the others unconstrained.

Let us consider the `replace` operation from Fig. 1. If we evaluate it on the arguments `s` = ””, `t` = ””, `u` = ”*a*”, the result is, according to the SMT-LIB semantics, the constant string ”*a*”. We can thus transform the formula by replacing the variables `s` and `res` with the constants ”” and ”*a*”. This replacement yields the sat formula from List. 1, which has more constrained models and exposes a soundness bug in two widely-used SMT solvers. This and other satisfiability-preserving transformations, including some that combine multiple operations, are described in more detail in Sect. 3.1.

2.2 Generating unsatisfiable formulas

To show that a formula is *unsatisfiable* (unsat), a solver has to prove that there does not exist an assignment to the free variables within the background theory that satisfies the formula; i.e., the formula evaluates to false for all possible values assigned to its variables [14].

Since many SMT theories include infinite sorts such as integers or strings, it is not possible to enumerate and check all possible value assignments and, thus, we cannot use our executable semantics to determine the ground truth. To synthesize formulas that are unsat by construction, we start from the following observation: conjoining a formula and the negation of an equivalent formula always results in an unsatisfiable formula. That is, for equivalent formulas A and B, the formula F defined as $\neg A \wedge B$ is unsatisfiable.

We obtain interesting formulas F by leveraging equivalences between different operations of the string theory. Out of the 12 string operations from Tab. 1, only `concat`, `length`, and `equals` are considered primitive operations. All the other string operations can be expressed through these primitive operations [35]. Tab. 2 presents an overview of *equivalent formulas* for non-primitive string operations. This table is an extended variant of the preprocessing rules from [35] and of the string function definitions from [10].

In the first step of our input construction technique, we automatically generate a test case from each of the 12 equivalences by conjoining the negation of the formula from column 2 in Tab. 2 and the corresponding formula from column 3. In this step, we omit existential quantifiers in the formulas from column 3 (that is, the existentially-quantified becomes a fresh free variable). This is possible because all existential quantifiers in the equivalent formulas are in positive positions, and $Q(x)$ is (un)satisfiable if and only if $\exists\ x : Q(x)$ is (un)satisfiable. Since the resulting formulas do not use any existentially-quantified variables, we use the terms *quantifier* and *quantified variable* in the rest of the paper to refer to *universal quantifier* and *universally-quantified variable*, respectively. For the quantifiers from Tab. 2, we specified *patterns* (also called *triggers*), which are used by the solver to decide how to instantiate the quantifiers and, thus, affect the search for a model [21].

$$\begin{aligned}
&\neg(\texttt{replace}(s, t, u) = \mathit{res}) \wedge i = \texttt{indexOf}(s, t, 0) \wedge \\
&\quad (0 \leq i \Rightarrow s = s_1 \mathbin{++} s_2 \mathbin{++} s_3 \wedge \texttt{length}(s_1) = i \wedge \\
&\qquad s_2 = t \wedge \mathit{res} = s_1 \mathbin{++} u \mathbin{++} s_3) \wedge \\
&\quad (i < 0 \Rightarrow \mathit{res} = s)
\end{aligned}$$

Figure 3: An unsat formula generated in step 1 for which CVC4 times out and Z3str3 has non-deterministic behavior. *i* **has type Int, all the other variables have type String.**

Our technique can be parameterized with different equivalent formulas. Other equivalences can be obtained, for instance, by rewriting formulas on the right-hand side in Tab. 2 using equalities from the left-hand side. The inputs without quantifiers can be used also for testing SMT solvers that do not support quantification yet.

An example test case that we synthesize in step 1 and corresponds to E3 is shown in Fig. 3. For this test, CVC4 times out, and Z3str3 non-deterministically returns timeout, unsat, unknown, or segmentation fault. In the second step of our approach, we automatically transform the previously generated formulas into equisatisfiable, but more complex ones. These formulas either have larger unsat cores, requiring the solver to combine more terms to derive a contradiction, or contain additional terms that are not relevant for proving unsat, but may complicate the proof search for the solver.

To show, for example, that the formula $x \neq 0 \wedge x = 0$ is unsatisfiable for all integers x, the solver relies on the fact that no number can be at the same time zero and non-zero. If the conjunct $x = 0$ is replaced by $x' = 0 \wedge 2x' - x' = x$ then all three conjuncts contribute to proving unsat. By removing any of them, the formula becomes satisfiable. Thus, these three terms represent the *minimal unsat core*. The transformations that we designed for the unsat case, described in detail in Sect. 3.2, are based on similar rewritings.

This concludes the high-level overview of our approach. Given an executable version of the reference semantics and the equivalent formulas, test case generation is deterministic and fully automatic. The ground truth is always known, so all the synthesized formulas can be directly used for testing, without additional human effort for constructing the test oracles. Our approach produces increasingly complex test cases, which often allows developers to detect errors with simple inputs such that errors are easy to reproduce and debug.

The main ingredients of our technique are not specific to the string theory. One can construct quantifier-free sat formulas from any theory that has an executable semantics, such as fixed-size bit-vectors or integers. Moreover, unsat formulas can be generated as long as operations or constants from the theory can be expressed in multiple ways. For instance, the fact that x is a positive real can be expressed as $x : Real : x \geq 0$ but also as $\exists y : Real : x = y * y$

Table 2: Equivalent formulas for non-primitive string operations

Id	String operation	⇔	Equivalent formula
E1	$\mathtt{at}(s, \mathit{off}) = \mathit{res}$	⇔	$\mathit{res} = \mathtt{substr}(s, \mathit{off}, 1)$
E2	$\mathtt{intToStr}(n) = \mathit{res}$	⇔	$(\mathit{res} = \text{""} \textbf{ if } n < 0) \land (\mathit{res} = \text{"0"} \textbf{ if } n = 0) \land \ldots \land (\mathit{res} = \text{"9"} \textbf{ if } n = 9) \land$ $(\mathit{res} = \mathtt{intToStr}(n \,/\, 10) \mathbin{++} \mathtt{intToStr}(n \,\%\, 10) \textbf{ if } n \geq 10)$
E3	$\mathtt{replace}(s, t, u) = \mathit{res}$	⇔	$i = \mathtt{indexOf}(s, t, 0) \land (\exists\, s_1, s_2, s_3 : s = s_1 \mathbin{++} s_2 \mathbin{++} s_3 \land \mathtt{length}(s_1) = i \land s_2 = t \land$ $\mathit{res} = s_1 \mathbin{++} u \mathbin{++} s_3 \textbf{ if } i \geq 0) \land (\mathit{res} = s \textbf{ otherwise})$
E4	$\mathtt{substr}(s, \mathit{off}, \mathit{len}) = \mathit{res}$	⇔	$(\exists\, s_1, s_2, s_3 : s = s_1 \mathbin{++} s_2 \mathbin{++} s_3 \land \mathtt{length}(s_1) = \mathit{off} \land \mathtt{length}(s_2) = \mathit{len} \land \mathit{res} = s_2$ $\textbf{if } \mathit{off} \geq 0 \land \mathit{off} < \mathtt{length}(s) \land \mathit{len} > 0) \land (\mathit{res} = \text{""} \textbf{ otherwise})$
E5	$\mathtt{indexOf}(s, t, \mathit{off}) = \mathit{res}$	⇔	$(\mathit{res} = \mathit{off} \textbf{ if } t = \text{""} \land \mathit{off} \geq 0 \land \mathit{off} \leq \mathtt{length}(s)) \land$ $(\exists\, s_1, s_2, s_4 : s = s_1 \mathbin{++} s_2 \mathbin{++} t \mathbin{++} s_4 \land \mathit{off} = \mathtt{length}(s_1) \land (\forall\, i\, \{\mathtt{substr}(t, 0, i)\} : i \geq 0 \land$ $i < \mathtt{length}(t) \Rightarrow \mathtt{contains}(s_2 \mathbin{++} \mathtt{substr}(t, 0, i), t) = \mathit{false}) \land \mathit{res} = \mathtt{length}(s_1 \mathbin{++} s_2) \textbf{ if } t \neq \text{""} \land$ $\mathit{off} \geq 0 \land \mathit{off} \leq \mathtt{length}(s)) \land (\mathit{res} = -1 \textbf{ otherwise})$
E6	$\mathtt{strToInt}(s) = \mathit{res}$	⇔	$(\mathtt{intToStr}(\mathit{res}) = s \textbf{ if } s \neq \text{""} \land \forall\, j\, \{\mathtt{at}(s, j)\} : j \geq 0 \land j < \mathtt{length}(s) \Rightarrow \mathtt{at}(s, j) = \text{"0"} \lor \ldots \lor$ $\mathtt{at}(s, j) = \text{"9"}) \land (\mathit{res} = -1 \textbf{ otherwise})$
E7	$\mathtt{contains}(s, t) = \mathit{true}$	⇔	$\exists\, s_1, s_3 : s = s_1 \mathbin{++} t \mathbin{++} s_3$
E8	$\mathtt{contains}(s, t) = \mathit{false}$	⇔	$\forall\, s_1, s_2, s_3\, \{s_1 \mathbin{++} s_2 \mathbin{++} s_3\} : (s = s_1 \mathbin{++} s_2 \mathbin{++} s_3) \Rightarrow (s_2 \neq t)$
E9	$\mathtt{prefixOf}(s, t) = \mathit{true}$	⇔	$\exists\, t_2 : t = s \mathbin{++} t_2$
E10	$\mathtt{prefixOf}(s, t) = \mathit{false}$	⇔	$\forall\, t_1, t_2\, \{t_1 \mathbin{++} t_2\} : (t = t_1 \mathbin{++} t_2) \Rightarrow (t_1 \neq s)$
E11	$\mathtt{suffixOf}(s, t) = \mathit{true}$	⇔	$\exists\, t_1 : t = t_1 \mathbin{++} s$
E12	$\mathtt{suffixOf}(s, t) = \mathit{false}$	⇔	$\forall\, t_1, t_2\, \{t_1 \mathbin{++} t_2\} : (t = t_1 \mathbin{++} t_2) \Rightarrow (t_2 \neq s)$

s, t, u, s_1, s_2, s_3, s_4, t_1, t_2: type String; n, off, len, i, j: type Int
All formulas are implicitly universally quantified over all the arguments.

++ denotes string concatenation.
Patterns for universal quantifiers are shown between {}.

(every positive number has a square root). The constant 0 can be also defined through operations from the theory of real numbers: $0 = z + (-z)$. From these equalities, one can generate the formula $\neg[x \geq (z + (-z))] \land (x = y * y)$, unsatisfiable by construction. Similar rewritings and equalities do exist for other theories.

Our technique is also not specific to SMT solvers. Since it treats the solver under test as a black box, it can be also applied to test the soundness and precision of other solvers, for instance, automata-based solvers, like SMC [26], ABC [10], and MT-ABC [11]. These solvers encode the input constraints as finite automata and determine their satisfiability by counting the number of possible models.

```
1  // step 1
2  foreach supported operation op {
3   input = genSimpleSatFormula(op);
4   // input has the form op(args) = res
5   res, model, _ := invokeSolver(input);
6   assert res == sat && correctModel(model, input);
7  }
8
9  // step 2
10 foreach formula f synthesized in step 1 {
11   // constant assignment
12   input := perform constant assignment on f;
13   res, model, _ := invokeSolver(input);
14   assert res == sat && correctModel(model, input);
15
16   // term synthesis
17   input := perform term synthesis on f;
18   res, model, _ := invokeSolver(input);
19   assert res == sat && correctModel(model, input);
20 }
```

Figure 4: Algorithm for synthesizing sat input formulas. `invokeSolver` yields the solver's result on the input formula (i.e., sat, unsat, unknown, timeout or error), a model for sat formulas, and an unsat core for unsat formulas, if available. `correctModel` uses the reference semantics to check the validity of the model with respect to the input formula. We do not check partial models (generated by some solvers for unknown results), as their correctness is not guaranteed.

3 SATISFIABILITY-PRESERVING TRANSFORMATIONS

In this section, we describe our technique for constructing complex SMT formulas from the string theory through satisfiability-preserving transformations. Our transformations for the sat and the unsat case are presented in the following subsections.

3.1 Transformations for satisfiable formulas

In the first step of the sat input construction technique (described in Sect. 2.1 and presented in pseudo-code in Fig. 4, lines 1–7), we synthesize simple formulas, with unconstrained parameters and results, which are trivially satisfiable. The second step (lines 9–20 in Fig. 4) strengthens the initial formulas by adding constraints on the values of the free variables and synthesizes formulas that may contain multiple operations to test their interactions. Our algorithm is deterministic, always generates the same tests, in the same order.

We present two satisfiability-preserving transformations in the following. *Constant assignment* uses an executable semantics to compute models for simple sat formulas and then transforms these formulas by replacing some of their free variables by values from the model. *Term synthesis* enumerates terms from the theory under test and evaluates them using an executable semantics. It then substitutes free variables from the simple input formulas with more complex terms, such that the formulas remain satisfiable. Both transformations yield quantifier-free sat formulas.

Constant assignment transformation. Many software errors are caused by the incorrect handling of corner cases. For this reason, the first transformation is inspired by boundary testing and consists of assigning predefined constants to (some of) the free variables of the initial formulas. The set of predefined constants is configurable; we used typical boundary values in our experiments. For example, for the variables of type String we considered empty strings, strings of length one, as well as strings containing quotes, escape sequences, and non-ASCII characters in hexadecimal format. For integers, we picked a small set of valid and invalid indices and lengths, such as $\{-1, 0, 2\}$. The string operations cannot have other types of parameters, but some of them have boolean results (see column 3 from Tab. 1), for which we considered both true and false.

Given the simple input formulas that we generated in step 1 (see Sect. 2.1), we use concrete executions to determine their models. For this purpose, we implement an *executable semantics* for all string operations based on the *reference semantics*, in our case, the SMT-LIB specification [5]. This implementation is straightforward since most programming languages (e.g., Java) already provide string libraries that offer most of the operations. Therefore, the implementation effort mostly consists of ensuring that the semantics of these library operations and the SMT-LIB standard match. For example, the Java `String.replace` method replaces *all* occurrences of `t` in `s`, whereas the SMT-LIB operation replaces just the *first* occurrence. Moreover, converting any negative integer to a string in Java will yield its textual representation, whereas the SMT-LIB result will be the empty string. To handle these and other similar mismatches, we implement a *wrapper* for the Java string operations according to the SMT-LIB semantics, which represents our executable semantics.

We then exhaustively evaluate each operation on all possible combinations of constant arguments from the set of predefined values. As all string operations are total [5], the evaluation always succeeds, producing models for the simple formulas from step 1. The constant assignment transformation obtains new input formulas by replacing some of the free variables in a simple formula by constants from the computed model. Since this transformation is based on valid models, it is guaranteed to produce satisfiable formulas.

Consider, for example, the `replace` operation, for which we already generated a simple test case during the first step (see Fig. 1). If the set of predefined constants includes the string ”a” and the empty string, then one of the concrete evaluations is `replace`(””, ””, ”a”) = ”a” . We can use this computed model to derive several new inputs. For instance, we can replace the free variables `s` and `res` from the simple formula by the corresponding constants from the model (here, an empty string and ”a”, respectively) to obtain a new sat formula: `replace`(””, t, u) = ”a”. This formula, presented in SMT-LIB notation in List. 1, revealed soundness bugs in Z3-seq and Z3str3.

As shown in Fig. 4 (lines 11–14), we run the solver on the transformed formula and report an error if the result is different from sat (lines 13–14). Otherwise, we check if the solver produces a correct model using our executable semantics (line 14): we evaluate the string operation on the parameters from the model generated by the solver and compare the result of the evaluation to the result from the generated model. If they are unequal, the solver is unsound; we found such cases in our evaluation (see Sect. 4.1).

$\texttt{at}(\texttt{at}(tmp\_str_0,\ tmp\_int_1),\ \texttt{indexOf}(tmp\_str_0,\ tmp\_str_0,\ tmp\_int_1))$
$= tmp\_str_2 \mathbin{++} tmp\_str_2$

Figure 5: A sat formula generated in step 2 through term synthesis, which exposes a soundness bug in Z3str3. tmp_int_1 **has type Int,** tmp_str_0 **and** tmp_str_2 **have type String.**

Term synthesis transformation. To test interactions between different operations, we transform the simple formulas from step 1 by replacing some free variables with more complex terms. We use terms from the string theory, which are sufficient to supply string, integer, and boolean parameters or results for the string operations.

Starting from a set of constants, we synthesize all type-correct applications of string operations up to a predefined bound and evaluate them using our executable semantics; this produces a pool of terms. We transform a simple formula from step 1 in four steps: (1) We replace the arguments of the operation under test with terms from the pool. (2) We evaluate the resulting term in the executable semantics. (3) We replace the result variable in the simple formula by another term from the pool with the same result, which ensures that the equality holds. (4) The complex formula used to test the solvers is then obtained by replacing the constants in the resulting equality by free variables, which yields a sat formula. It is important to represent multiple occurrences of the same constant by the same free variable, to connect the operations more tightly and to further constrain the set of possible models. The transformation is applied exhaustively for all terms from the pool.

To illustrate our technique, let us assume that the initial set of strings includes the constants ”a” and ””, and the initial set of ints contains the value -1. The pool will then contain, among others, the terms `at`(”a”, -1) (with concrete value ””), `indexOf`(”a”, ”a”, -1) (with concrete value 0), and `concat`(””, ””) (with value ””). Starting from the simple formula for the `at` operation, the transformation proceeds as follows: (1) We substitute the arguments of `at` to obtain, for instance, `at`(`at`(”a”, -1), `indexOf`(”a”, ”a”, -1)). (2) Evaluating this term yields ””. (3) We equate the term with another term with the same result and obtain, for example, `at`(`at`(”a”, -1), `indexOf`(”a”, ”a”, -1)) = `concat`(””, ””). (4) We replace the constants ”a”, ””, and -1 with three fresh variables `tmp_str0`, `tmp_str2`, and `tmp_int1`, which yields the input formula from Fig. 5. This formula exposes a soundness bug in Z3str3, which incorrectly returns unsat. Note that when the operations `at`, `indexOf`, and `concat` were tested separately, the solver returned the expected results. It is their combination that exhibits the error. The formula from Fig. 5 was obtained with bound 1 for the term pool, that is, the arguments and result are all single applications of an operation on constants; larger bounds lead to more complex formulas.

Once we have synthesized the new input, executing the test and checking if the solver returned a correct model is analogous to the constant assignment transformation (see lines 16–19 from Fig. 4). Note that one can easily combine the two transformations we proposed by replacing only *some* of the constant occurrences in the last step of the term synthesis transformation with free variables.

```
1  // step 1
2  foreach supported operation op {
3   input, expectedCore = genSimpleUnsatFormula(op);
4   // input has the form !A(x) && B(x,y), where
5   // A is the formula from column 2 in Table 2 and
6   // B is the corresponding formula from column 3
7   res, _, core := invokeSolver(input);
8   assert res == unsat && core == expectedCore;
9  }
10
11 // step 2
12 foreach formula f synthesized in step 1 {
13  // variable replacement
14  foreach variable x that is free both in A and B {
15   foreach applicable equality eq in NC1–NC8 {
16    input, expectedCore := replace all x in f using eq
17    res, _, core := invokeSolver(input);
18    assert res == unsat && core == expectedCore;
19   }
20  }
21
22  // constant replacement
23  foreach constant c in B {
24   foreach applicable equality eq in C1–C40 {
25    input, expectedCore := replace all c in f using eq
26    res, _, core := invokeSolver(input);
27    assert res == unsat && core == expectedCore;
28   }
29  }
30
31  // redundancy introduction
32  foreach variable y that is free only in B {
33   foreach applicable equality eq in NC1–NC8 {
34    input, expectedCore := replace all y in f using eq
35    res, _, core := invokeSolver(input);
36    assert res == unsat  && core == expectedCore;
37   }
38  }
39 }
```

Figure 6: Algorithm for synthesizing unsat input formulas. invokeSolver yields the solver's result on the input formula (i.e., sat, unsat, unknown, timeout or error), a model for sat formulas, and an unsat core for unsat formulas, if available.

3.2 Transformations for unsatisfiable formulas

In the first step of the unsat input construction technique (described in Sect. 2.2 and presented in pseudo-code in Fig. 6, lines 1–9), we test each non-primitive string operation together with its equivalent formula. Recall that if two formulas A and B are equivalent then the formula $F := \neg A \wedge B$ is by construction unsatisfiable. To obtain more complex unsat formulas, we transform the simple ones into formulas with larger unsat cores. Therefore, the solver needs to reason about more properties to prove unsatisfiability.

Consider a simple input formula F that contains one variable x that occurs in both A and B, and one variable y that is existentially bound in B and, thus, becomes a free variable after the existential quantifier is removed (see Sect. 2.2). This formula can be written as:

$$F(x, y) := \neg A(x) \wedge B(x, y)$$

To obtain an unsat formula with larger unsat core, we replace all the occurrences of x in B by a fresh variable x_{fresh} and conjoin a clause $C(x, x_{fresh})$ from the string theory that implies $x = x_{fresh}$. The resulting formula is still unsat, but the unsat core now also includes $C(x, x_{fresh})$:

$$F(x, x_{fresh}, y) := \neg A(x) \wedge B(x_{fresh}, y) \wedge C(x, x_{fresh})$$

Table 3: Equalities between the string operations and non-constant and constant strings (NC1–NC7 and C1–C15), integers (NC8 and C16–C31), and booleans (C32–C40)

Id	Equality
NC1	$\texttt{at}(s, 0) = s$ **if** $\texttt{length}(s) = 1$
NC2	$\texttt{concat}(s, \text{””}) = s$
NC3	$\texttt{concat}(\text{””}, s) = s$
NC4	$\texttt{replace}(s, s, s) = s$
NC5	$\texttt{replace}(s, t, u) = s$ **if** $\texttt{contains}(s, t) = \mathit{false}$
NC6	$\texttt{replace}(s, t, u) = s$ **if** $\texttt{indexOf}(s, t, 0) = -1$
NC7	$\texttt{substr}(s, 0, \texttt{length}(s)) = s$
NC8	$\texttt{indexOf}(s, \text{””}, \mathit{off}) = \mathit{off}$ **if** $\mathit{off} \geq 0 \wedge \mathit{off} \leq \texttt{length}(s)$
C1	$\texttt{at}(s, \mathit{off}) = \text{””}$ **if** $\mathit{off} < 0 \vee \mathit{off} \geq \texttt{length}(s)$
C2	$\texttt{concat}(\text{””}, \text{””}) = \text{””}$
C3	$\texttt{intToStr}(n) = \text{””}$ **if** $n < 0$
C4	$\texttt{replace}(\text{””}, \text{””}, \text{””}) = \text{””}$
C5	$\texttt{substr}(s, \mathit{off}, \mathit{len}) = \text{””}$ **if** $\mathit{off} < 0 \vee \mathit{off} \geq \texttt{length}(s) \vee \mathit{len} \leq 0$
C6	$\texttt{intToStr}(n) = \text{”0”}$ **if** $n = 0$
...	...
C15	$\texttt{intToStr(n)} = \text{”9”}$ **if** $n = 9$
C16	$\texttt{indexOf}(s, t, \mathit{off}) = -1$ **if** $\mathit{off} < 0 \vee \mathit{off} > \texttt{length}(s)$
C17	$\texttt{indexOf}(s, t, \mathit{off}) = -1$ **if** $\texttt{contains}(s, t) = \mathit{false}$
C18	$\texttt{strToInt}(s) = -1$ **if** $s = \text{””}$
C19	$\texttt{strToInt}(s) = -1$ **if** $\exists\, i : i \geq 0 \wedge i < \texttt{length}(s) \wedge$ $\texttt{at}(s, i) \neq \text{”0”} \wedge \ldots \wedge \texttt{at}(s, i) \neq \text{”9”}$
C20	$\texttt{length}(s) = 0$ **if** $s = \text{””}$
C21	$\texttt{strToInt}(s) = 0$ **if** $s = \text{”0”}$
C22	$\texttt{strToInt}(s) = 1$ **if** $s = \text{”1”}$
...	...
C31	$\texttt{strToInt}(s) = 9$ **if** $s = \text{”9”}$
C32	$\texttt{contains}(s, s) = \mathit{true}$
C33	$\texttt{equals}(s, s) = \mathit{true}$
C34	$\texttt{prefixOf}(\text{””}, s) = \mathit{true}$
C35	$\texttt{prefixOf}(s, s) = \mathit{true}$
C36	$\texttt{suffixOf}(s, s) = \mathit{true}$
C37	$\texttt{contains}(s, t) = \mathit{false}$ **if** $\texttt{indexOf}(s, t, 0) = -1$
C38	$\texttt{equals}(s, t) = \mathit{false}$ **if** $\texttt{length}(s) \neq \texttt{length}(t)$
C39	$\texttt{prefixOf}(s, t) = \mathit{false}$ **if** $\texttt{contains}(t, s) = \mathit{false}$
C40	$\texttt{suffixOf}(s, t) = \mathit{false}$ **if** $\texttt{contains}(t, s) = \mathit{false}$

Based on this general idea, we perform three transformations on the simple input formulas, which are described next and implemented in lines 11–39 of Fig. 6. With all three transformations, the unsat core of the resulting formula is unique and known by construction and, thus, can be used to check the correctness and minimality of the unsat core returned by the solver.

Variable replacement transformation. Our first transformation chooses a variable x that occurs freely in A and B and constructs a more complex formula as described above.

The clause $C(x, x_{fresh})$ is obtained from a set of equalities that we derived from the string theory. They are summarized in Tab. 3; we focus on the equalities NC1–NC8 here and will discuss the others later. For example, the equality NC1 expresses that the first character of a string is equal to the string itself, for any string of length 1. This additional constraint about the length of the string represents a *side condition* under which the equality holds.

Note that NC1–NC7 are equalities on strings, whereas NC8 is for integers. Depending on the type of the chosen variable x, we select an appropriate equality and obtain $C(x, x_{fresh})$ by substituting

$$\begin{array}{l}
\neg(\mathtt{replace}(s, t, u) = \mathit{res}) \land i = \mathtt{indexOf}(s, t, 0) \land \\
\quad (0 \leq i \Rightarrow s = s_1 \mathbin{++} s_2 \mathbin{++} s_3 \land \mathtt{length}(s_1) = i \land \\
\qquad\qquad s_2 = t \land \mathit{res\_fresh} = s_1 \mathbin{++} u \mathbin{++} s_3) \land \\
\quad (i < 0 \Rightarrow \mathit{res\_fresh} = s) \land \\
\quad \mathtt{at}(\mathit{res}, 0) = \mathit{res\_fresh} \land \mathtt{length}(\mathit{res}) = 1
\end{array}$$

Figure 7: An unsat formula generated in step 2 by increasing the unsat core, which exposes an unsoundness in Z3str3. i has type Int, all the other variables have type String.

the right-hand side variable by x_{fresh}, all occurrences of the same variable on the left-hand side of the equality by x, and all other variables by fresh variables.

For example, replacing the variable `res` in the formula from Fig. 3 using NC1 yields $C(\mathtt{res}, \mathtt{res}_{fresh}) := \mathtt{at}(\mathtt{res}, 0) = \mathtt{res}_{fresh}$. With this additional clause, we construct the unsat formula from Fig. 7. Note that the side condition `length(res) = 1` of NC1 is conjoined to the formula, making it stronger and preserving unsatisfiability.

To prove that this formula is unsat, an SMT solver can use NC1 (with `res` for s) and the last two conjuncts from Fig. 7 to derive $\mathtt{res} = \mathtt{res}_{fresh}$, which reduces the formula to the one we started from. This shows that the prover needs to perform additional reasoning steps, as the unsat core is extended by the additional conjuncts. For the formula from Fig. 7, Z3str3 unsoundly returns sat.

The same transformation can also be applied to the other free variables `s`, `t`, and `u`. It is also possible to replace multiple variables simultaneously, but we omitted such transformations in our experiments. There, we explore each combination of one variable and a corresponding equality, as summarized in Fig. 6, lines 13–20.

Constant replacement transformation. Analogously to the previous transformation, we can replace a constant c by a term that evaluates to c. Starting from a simple formula $F(x, y) := \neg A(x) \land B(c, y)$, we construct the following formula for some constant c:

$$F(x, z_{fresh}, y) := \neg A(x) \land B[z_{fresh}/c, y] \land C(c, z_{fresh})$$

To obtain the additional clause $C(c, z_{fresh})$, we use known equalities from the string theory. The equalities C1–C40 from Tab. 3 all equate a string term to a constant. For a chosen constant c, we select one of the equalities of the form $t = c$ and define $C(c, z_{fresh}) := z_{fresh} = t$, as shown in pseudo-code in Fig. 6, lines 22–25. As in the previous transformation, this step preserves unsatisfiability and enlarges the unsat core by the additional equality. This information, known by construction, is used as the test oracle (lines 26–27). If instead of rewriting *res*, in Fig. 7 we replace the constant 0 by C20 then we obtain an unsat formula that exposes a soundness bug in Z3-seq.

Redundancy introduction transformation. We also experimented with a variation of the variable replacement transformation, where we apply the same transformation to a variable y that occurs freely in B, but not in A (see Fig. 6, lines 31–39). These variables were initially introduced by existential quantifiers in the equivalent formulas from Tab. 2. Consequently, renaming them to y_{fresh} and conjoining $C(y, y_{fresh})$ to the formula does not extend the unsat core. Nonetheless, it introduces *redundancy*, that is, additional variables and terms that may obfuscate the proof of unsatisfiability.

It is also possible to apply this transformation to universally-bound variables in B. In this case, the additional clause from Tab. 3 is added under the quantifier such that it implies the quantifier's body; the new variables that occur in the clause are added as universally-bound variables, and the pattern of the quantifier is extended to mention all the quantified variables. The new triggering terms are directly derived from the additional equality, and the side condition is added as an implication. In our experiments, redundancy introduction produced test cases that did reveal errors. Fig. 8 shows an example, where the quantified variable `t`$_2$ from E10 was rewritten using NC7. This formula exposes a soundness issue and non-deterministic behavior for Z3str3, which returns sat or segmentation fault.

$$\begin{array}{l}
\neg(\mathtt{prefixOf}(s, t) = \mathit{false}) \land \\
\forall\, t_1, t_2, t_2\_\mathit{fresh}\ \{t_1 \mathbin{++} t_2\_\mathit{fresh}, \mathtt{substr}(t_2, 0, \mathtt{length}(t_2))\} : \\
(\mathtt{substr}(t_2, 0, \mathtt{length}(t_2) = t_2\_\mathit{fresh}) \Rightarrow \\
\qquad\qquad ((t = t_1 \mathbin{++} t_2\_\mathit{fresh}) \Rightarrow t_1 \neq s)
\end{array}$$

Figure 8: An unsat formula generated in step 2 by introducing redundancy for universally-bound variables, which exposes a soundness bug and non-deterministic behavior for Z3str3. All the variables have type String.

Note that the three transformations can be combined and all of them can be applied multiple times, to increase the complexity of the previously synthesized formulas. In our experiments, each transformation was applied *independently* and the second synthesis step was performed only once, to facilitate error localization and to avoid generating redundant tests that fail due to the same bug. As our algorithm is deterministic, it always produces the same tests.

4 EXPERIMENTAL EVALUATION

In this section, we present the results obtained by applying our test case generation technique on three widely-used SMT solvers: Z3-seq (version 4.7.1), Z3str3 (version 4.7.1), and CVC4 (version 1.6). The two Z3 string solvers use different approaches: Z3-seq (the default string solver from Z3) encodes string operations into operations over sequences, while Z3str3 supports strings as built-in types. Our experiments show that our technique is able to synthesize formulas that reveal soundness bugs in two of the three tested solvers. They also uncover completeness and performance issues.

The experimental results show that our approach outperforms fuzzing and is also effective in finding bugs in other types of solvers, as we demonstrate on the automata-based solver MT-ABC [11].

4.1 Testing SMT solvers

In the first experiment, we tested the compliance of Z3-seq, Z3str3, and CVC4 string operations with the semantics defined in the SMT-LIB standard. In the following, we also discuss the impact of each component of our formula synthesis technique in finding the bugs.

Experimental setup. All the formulas that we synthesized are encoded into SMT-LIB 2.6 format [12]; patterns are part of this standard. We used the SMT-LIB Unicode Strings Theory [5] as the reference semantics and our wrapper of the Java string operations for the executable semantics. We set a timeout of 15 seconds for each test and we fixed the seed for the solvers' random number generator, the `sat.random_seed` (for all three solvers) and the `smt.random_seed` (only for Z3-seq and Z3str3), to reduce non-determinism. We used the options `produce-models` and `produce-unsat-cores` to enable the generation of the models and of the unsat cores, respectively.

Table 4: Overview of our results for Z3-seq, Z3str3, and CVC4

			# of tests with actual result (All random seeds = 0)																				
Expected	**Category /**	**# of tests**	**Z3-seq**							**Z3str3**							**CVC4**						
result	**Transformation**	**generated**	**IM**	**IC**	**S**	**U**	**K**	**T**	**E**	**IM**	**IC**	**S**	**U**	**K**	**T**	**E**	**IM**	**IC**	**S**	**U**	**K**	**T**	**E**
sat	operation	12	**0**	-	10	**0**	2	0	0	**0**	-	12	**0**	0	0	0	**0**	-	12	**0**	0	0	0
sat	constant assignment	4714	**24**	-	4158	**14**	518	0	0	**24**	-	4580	**105**	0	5	0	**0**	-	4714	**0**	0	0	0
sat	term synthesis	1394	**2**	-	842	**0**	483	67	0	**7**	-	1027	**109**	0	133	118	**0**	-	1394	**0**	0	0	0
unsat	equivalent formula	12	-	**0**	**0**	9	0	3	0	-	**0**	**0**	8	0	3	1	-	**0**	**0**	5	0	7	0
unsat [+p]	equivalent formula	12	-	**0**	**0**	9	0	3	0	-	**0**	**0**	8	0	3	1	-	**0**	**0**	5	0	7	0
unsat	larger unsat core	268	-	**0**	**1**	153	10	104	0	-	**10**	**5**	120	12	71	50	-	**0**	**0**	89	0	177	2
unsat [+p]	larger unsat core	268	-	**0**	**1**	153	9	105	0	-	**7**	**6**	120	7	78	50	-	**0**	**0**	89	0	176	3
unsat	redundancy introduction	178	-	67	**0**	50	37	24	0	-	21	**10**	58	5	41	43	-	23	**0**	26	0	125	4
unsat [+p]	redundancy introduction	178	-	67	**0**	42	36	33	0	-	22	**16**	52	8	30	50	-	25	**0**	24	0	123	6
total # of sat failed tests (out of 6120)					1110							501							0				
total # of unsat failed tests (out of 458)					254							272							338				
total # of unsat [+p] failed tests (out of 458)					254							278							340				

IM = incorrect model; **IC** = incorrect unsat core; **S** = sat; **U** = unsat; **K** = unknown; **T** = timeout; **E** = error; **[+p]** = with patterns for quantifiers; ▨ **n** = # of tests that failed due to soundness issues

We also used the option `strings-exp` for CVC4 to enable non-primitive string operations and the option `full-saturate-quant` to enable enumerative instantiation [31]. For the Z3-based solvers, we set the option `smt.core.minimize` to true to obtain the minimal unsat core; this option was not supported by CVC4 at the time of writing. For all other options, we used the default values; in particular, we did not use the solvers' own ability to check the validity of the generated models. The experiments were conducted on a 2.5 GHz Intel Core i7 CPU with 16 GB memory.

Results. The results obtained for the three solvers are summarized in Tab. 4. In this table, we report the expected result (column 1), the test category/transformation as described in Sect. 2 and Sect. 3 (column 2), the total number of tests generated for each of these categories (column 3), and the actual result returned by each solver (in the remaining columns). This result can be: incorrect model (when the solver correctly returned sat, but the produced model is not valid, i.e., evaluating the original formula on the model, using our executable semantics, yields false), incorrect unsat core (when the solver returned unsat, but the generated unsat core is not the minimal, expected one), sat, unsat, unknown, timeout, or error (when the solver crashed or returned an error message). For the unsat formulas, we report the results for each category *without* using patterns for quantifiers and *with* the patterns specified in Tab. 2 for the formulas that are quantified (unsat [+p] in Tab. 4). All our sat formulas are quantifier-free. Note that when the patterns are not provided, the solvers will try to infer them automatically.

The categories *operation* and *equivalent formula* refer to simple formulas synthesized in step 1 for testing each operation in isolation, or together with its equivalent formula from Tab. 2, respectively. The category *larger unsat core* includes the test cases obtained by applying the variable and constant replacement transformations from Sect. 3.2. For *term synthesis*, each variable was obtained by exactly one operation. For the *larger unsat core* and *redundancy introduction* transformations, each variable and constant was rewritten in one step, by independently applying all the corresponding equalities from Tab. 3. In each test case, only one of them was rewritten, with all its occurrences (as shown in Fig. 6).

Soundness issues. The number of tests that failed due to soundness issues, for each category, is showed with gray background.

`indexOf`$(tmp\_str_0, tmp\_str_1, tmp\_int_2) = 0$

Figure 9: A sat formula generated in step 2 through constant assignment for which Z3str3 produces an incorrect model. ***tmp_int***$_2$ **has type Int,** ***tmp_str***$_0$ **and** ***tmp_str***$_1$ **have type String.**

We classify an answer as being *unsound* if the solver returned sat instead of unsat, or vice versa, or if it generated an invalid model. An incorrect unsat core represents a soundness problem if the generated unsat core is *not* unsatisfiable. We observed this kind of error only for Z3str3, for the *larger unsat core* transformation and for the *equivalent formula* category. For *redundancy introduction*, the cores generated by all the solvers were always valid, but not necessary minimal; we consider that an imprecision, not a soundness issue.

For CVC4, none of the test cases revealed soundness issues. By contrast, Z3str3 has the highest number of tests that fail due to soundness bugs for both sat and unsat formulas. Some example inputs for which Z3str3 returned an incorrect result have already been presented in the previous sections. Fig. 9 shows another type of unsoundness, i.e., a sat formula obtained through the constant assignment transformation for which the solver correctly answered sat, but generated an invalid model: `tmp_str`$_0$ = ”*3MayMayMaZ*”, `tmp_str`$_1$ = ”*MayM*”, `tmp_int`$_2$ = 1; with these inputs, the result of `indexOf` is 1, not 0, as prescribed by the input formula in Fig. 9.

Note that Z3str3 does not support non-ASCII strings yet. Out of the 245 sat formulas unsoundly solved by Z3str3, 22 contain such strings. For a fair evaluation, we replaced them with ASCII strings and repeated the experiments. The results were the same. Moreover, the solvers use mathematical integers, while our executable semantics uses bounded integers. We manually inspected the few models rejected by our executable semantics that contained large numbers. All of them were valid and are, thus, *not* reported as errors in Tab. 4.

Other issues. Besides soundness problems, our tests revealed various completeness, performance, and implementation errors. For instance, the unknown result points to a completeness issue. Z3-seq returned unknown for approx. 17% of our sat formulas, blaming incompleteness in the sequence theory in all 1003 cases. We reported several failing tests and some of them have already been fixed. The timeout result suggests a performance problem, frequently observed for unsat formulas with all three solvers. Several tests

contains(intToStr(tmp_int_0), at(tmp_str_2, tmp_int_0))
= contains(tmp_str_2, tmp_str_2)

Figure 10: A sat formula generated in step 2 through term synthesis for which the result of Z3-seq depends on the random seeds. tmp_int_0 has type Int, tmp_str_2 has type String.

Table 5: Known bugs for Z3-seq, Z3str3, and CVC4

	Z3-seq			Z3str3			CVC4		
# issues	**T**	**WS**	**F**	**T**	**WS**	**F**	**T**	**WS**	**F**
closed	9	6	5	3	3	3	5	3	1
open	5	3	1	10	6	5	0	0	0

T = *total* number of issues; **WS** = *within the scope* of this paper: issues caused by regular expressions, user-defined functions, bit-vectors, or different configurations were excluded; **F** = issues we could *find* by manually inspecting the failing tests.

also failed for Z3str3 and CVC4 due to implementation errors. For CVC4, many tests hint at completeness or performance problems for unsat formulas, both with and without patterns for quantifiers. The reason, confirmed by the developers, is that with the given patterns, many of the quantifiers are not instantiated by default through E-matching [20]. Moreover, the enumerative instantiation [31], which is used by CVC4 when E-matching saturates, does not work optimally for non-primitive string operations. We reported the problem, and for some of our test cases it has been already fixed.

Adding patterns for quantifiers did not improve the results for Z3-seq and Z3str3. This experimental result suggests that the patterns we specified are similar to the ones automatically inferred by the two Z3-based solvers, which use the same engine for instantiating quantifiers. Other reason for the unknown result is incompleteness in the sequence theory, reported by Z3-seq for 33% of the failed tests. Z3str3 does not provide details on the reason of the incompleteness.

Our approach can be also used for discriminating between various configurations. For example, to test the solvers' robustness, we set the random seeds to 1465 (a value chosen arbitrarily), and we repeated the experiments. For CVC4 the results were the same. Z3-seq and Z3str3 were less robust. Fig. 10 shows a test case for which Z3-seq correctly returned sat when the seeds were 0, but answered unknown for the seeds 1465. Note that all the other examples from this paper were obtained with the random seeds set to 0.

The test cases that failed in our experiments do not necessarily refer to unique bugs. This is a general problem of any testing tool and is orthogonal to our formulas synthesis technique. Several approaches have been proposed in the literature for clustering static analysis alarms [29]; we could apply these ideas to our work, to automatically group the failing tests into similarity-based clusters. Note that our synthesis algorithm reduces by construction the number of redundant test cases; step 2 applies *individual* transformations to the simple formulas generated in step 1 (see Fig. 4 and Fig. 6), that is, it does not chain together transformations. However, we do apply step 2 even to those formulas that already lead to a failing test in step 1, which may uncover additional bugs. For example, Z3str3 timed out during step 1 for the formulas based on E3 and E5, but was unsound for tests derived in step 2 from these inputs.

Known bugs. Due to the large number of failed tests and the complexity of the implementation of the SMT solvers, it is not feasible to manually determine how many distinct bugs we uncovered.

Table 6: Failed tests on the latest versions of the SMT solvers

total # of still failing tests	**Z3-seq (4.8.6)**	**Z3str3 (4.8.6)**	**CVC4 (1.7)**
sat	6 [out of 1110]	237 [out of 501]	0 [out of 0]
unsat	156 [out of 254]	267 [out of 272]	335 [out of 338]
unsat [+p]	156 [out of 254]	274 [out of 278]	337 [out of 340]

[+p] = with patterns for quantifiers (given in Tab. 2)

To evaluate how effective our technique is in detecting distinct bugs, we assessed how many of a set of known bugs are found by our test cases. For Z3-seq and Z3str3, we considered the closed issues reported by the users from 23rd May 2018 until 26th January 2019, as well as all the currently open issues with the labels `string` or `z3str3` that were confirmed by the developers and do not explicitly refer to other versions than 4.7.1. Similarly, for CVC4 we considered the issues reported from 25th June 2018 until 17th April 2019 (both closed and still open) related to the string theory.

The known bugs are summarized in Tab. 5. From the total number of issues (column 1 for each solver), we removed the ones that are not in the scope of this paper, that is, contain regular expressions, user-defined functions based on string operations, or formulas combining string operations with bit vectors, which we do not support. We also excluded the issues explicitly caused by additional configuration options that we do not use. We report as *found* (column 3) only those bugs for which we could manually find a failing test case that exhibits it, based on the description from the comments or inferred from the fix. Known bugs *not* reported as found might still be detected by our test suite, but we were not able to clearly identify an appropriate test case. That is, the reported number of found bugs is a *lower* bound on the actual number. This experiment shows that our technique effectively detects bugs that occurred in actual applications of the tested SMT solvers and were reported and confirmed. In total, we found 15 of the 21 bugs (71%).

Sensitivity analysis. The effectiveness of our technique depends on three ingredients: (1) the set of predefined constants, (2) the combinations of operations used in a formula, needed to test their interactions, and (3) the usage of different random seeds. The manual inspection of the tests that detected the known bugs from Tab. 5 shows that all three ingredients are necessary. Finding some of the bugs required specific ways of constructing the inputs; e.g., the implementation errors from CVC4 are revealed only by the tests that use the equalities NC1 and NC5 from Tab. 3 to rewrite variables from E10 and E12. Similarly, a soundness bug in Z3str3 is detected only by the test that replaces the result variable from E3 using the equality NC1. Other bugs are revealed by several inputs that follow a common pattern, such as formulas obtained through constant assignment that test the `indexOf` operation with negative or out-of-bounds offset, or formulas generated through term synthesis that include `intToStr` or `strToInt` as arguments for other operations. The bug from Fig. 10 can be observed only when testing the `contains` operation twice, with different random seeds. Our experiments do not suggest that certain equivalences (from Tab. 2) or transformations are substantially more useful than others.

Recent improvements. As we mentioned above, we reported several bugs for the three solvers, many of which were confirmed or fixed. To assess the recent improvements, we re-ran the failed tests on the latest versions of the solvers at the time of writing, i.e., 4.8.6 for Z3-seq and Z3str3 and 1.7 for CVC4. The cumulative results for all types of errors are presented in Tab. 6. In the following, we discuss our main observations, focusing on the soundness bugs.

Compared to the results from Tab. 4, summarized between square brackets, the number of failing tests decreased substantially for the Z3-based solvers, at least in part due to our bug reports. For Z3-seq, no sat test still failed due to soundness bugs. For 69 unsat formulas based on `intToString`, Z3-seq returned sat; we reported 1 new soundness bug and it was confirmed. For Z3str3, 40 sat and 33 unsat tests failed due to soundness problems. Some of them correspond to open bugs, and we reported 3 additional soundness errors. For CVC4 we did not find soundness bugs; the tests failed due to performance issues and because the minimization of the unsat cores was not yet supported. This feature was added in the meantime and the developers further improved the performance in response to our bug reports, but these changes are not yet part of the main branch.

All these results were obtained with respect to the SMT-LIB semantics. Even though for some operations the semantics described in the documentation of a particular solver may be slightly different, none of the bugs we reported were considered false positives by the developers. All three solvers intend to comply to the standard. Our experiments show that soundness and completeness bugs in decision procedures are not uncommon. They are due to various issues, including mis-interpretations of the expected semantics, flaws in the used algorithms, and coding errors. These findings have implications for solver developers, who need to systematically test for such bugs. Our work offers a technique to accomplish that.

4.2 Comparison with fuzzing

In this subsection, we compare our technique with StringFuzz [16], a state-of-the-art test case generator for string formulas. For this experiment, we ran parts of a test suite generated by StringFuzz (from the folder generated.zip [6]) on the three SMT solvers, using the same versions as for our main experiments, i.e., 4.7.1 for Z3-seq and Z3str3, and 1.6 for CVC4. We discarded the tests for which the expected result was not specified, as for them we could not automatically classify the actual result as correct or not. In total, we included 700 tests in SMT-LIB 2.5 format, from nine different categories: `lengths-short`, `lengths-long`, `lengths-concats`, `concats-small`, `concats-big`, `concats-balanced`, `regex-small`, `regex-big`, and `different-prefix`. All of them are quantifier-free and 120 tests include regular expressions. We used the same experimental setup as for our tool (with the random seeds set to 0) and we set the `lang` option to `smt2` for CVC4, to avoid parsing errors.

As StringFuzz cannot check if the generated models and unsat cores are correct, we considered that a test passed when the solver answered sat or unsat, as expected. Z3-seq and Z3str3 timed out for 82 tests, and returned correct results for all the others. Similarly, 74 tests failed for CVC4 due to timeout, while all the others passed.

The experiment shows that StringFuzz could detect performance problems, but no soundness or completeness bugs. A reason may be that, besides regex, StringFuzz can generate only formulas with primitive string operations, which are not enough for revealing these classes of errors. In contrast, our technique also uncovered several soundness and completeness errors in the same versions of the SMT solvers, including confirmed bugs.

Table 7: Overview of our results for MT-ABC

Expected result	Category / Transformation	# of tests generated	# of tests with actual result		
			S	U	E
sat	operation	12	11	**1**	0
sat	constant assignment	3568	3145	**275**	148
sat	term synthesis	1394	974	**175**	245
unsat	equivalent formula	6	4*	2	0
unsat	larger unsat core	121	63*	56	2
unsat	redundancy introduction	71	36*	35	0
total # of sat failed tests (out of 4974)				844	
total # of unsat failed tests (out of 198)				105	

S = sat; **U** = unsat; **E** = error; *imprecision due to over-approximation
n = # of tests that failed due to soundness issues

4.3 Testing automata-based solvers

Our technique is not limited to SMT solvers; it can also be applied for testing other types of solvers, such as automata-based solvers. In this subsection, we present the results for MT-ABC [11], an automata-based solver that performs model counting. It supports both string and numerical constraints, and classifies an input formula as satisfiable if the counted number of models is greater than 0. For some constraints, it may over-approximate the set of solutions, thus *imprecisely* answering sat for an unsat formula. Nonetheless, a *sound* implementation should not classify a sat formula as unsat.

As MT-ABC accepts as input a subset of the SMT-LIB format, we used a modified version of our tests that contains only supported features. We also replaced the escape sequence for double quotes within a string literal with the corresponding one in MT-ABC. As non-ASCII strings are not yet supported, we discarded the sat tests that included these constants in the formulas or in the possible model. The unsat tests based on E2, E5, E6, E8, E10, E12 from Tab. 2 could not be handled by MT-ABC because it does not support quantifiers and the mod operator, so we removed them as well.

We tested the code version [3], using default options. This commit includes a fix for E7, E9, and E11, based on a crash that we found and reported. We used the SMT-LIB Unicode Strings Theory [5] as the reference semantics, because MT-ABC recently updated the implementation of the string operations to match this standard.

The results are summarized in Tab. 7. The soundness problems are showed with gray background, while the imprecise answers are marked with *. As it can be observed, our technique effectively synthesized formulas that exposed various soundness and precision issues, as well as implementation failures for different string operations. For example, MT-ABC unsoundly returned unsat for the formulas from Fig. 2 and Fig. 5. We already reported 6 distinct soundness bugs and 6 crashes for *constant assignment* and for the *operation* category, because they are easier to debug. The developers confirmed them and appreciated that we sent simple formulas that expose the bugs. As a result, some of them were fixed within a day.

4.4 Threats to validity

We identified four threats to the validity of our experiments:

Ground truth. Our technique relies on an executable semantics for the string operations, on the equivalent formulas from Tab. 2 and on the equalities from Tab. 3. Errors in these components could lead to incorrect tests. To mitigate this threat, we carefully reviewed all the components of our approach. Since they are simple variations of the SMT-LIB semantics, we are confident that they are correct.

Non-deterministic behavior. The solvers use randomized algorithms, which can lead to non-deterministic behavior (see Sect. 2.2 for an example). We mitigated this problem by fixing the random seeds and by performing the experiments from Sect. 4.1 with two different values. Nevertheless, some of the results from Tab. 4, Tab. 6, and Sect. 4.2 may require multiple runs to be reproduced.

Pattern selection. The patterns chosen for the quantified formulas are independent of the way in which each solver handles quantifier instantiations. Other patterns could have been more efficient for proving that certain formulas are unsat or alternative rewritings of the formulas could have better triggered particular instantiations. Nonetheless, our patterns are configurable and we ran the experiments with and without patterns to asses their impact.

Known bugs. To determine if our technique generates test cases that can detect known bugs, we manually matched some of the failed tests against the confirmed bug reports. As we reported only clear matches as found bugs, we are confident that the values from Tab. 5 are a *lower* bound on the number of known bugs we detected.

5 RELATED WORK

The developers of SMT solvers usually create their own test suites, which include manually-written tests and regression tests derived from bug reports [2, 8]. Our approach automates parts of this time-consuming process by automatically generating test cases of incremental complexity; this facilitates debugging and error localization.

Differential testing. An effective approach used in practice is differential testing [28], which compares the results produced by different solvers on a set of benchmarks [13, 25]. Different results suggest a bug in one of the solvers; determining which one is at fault requires additional effort. In our case, the ground truth is known upfront, so our synthesized inputs can be directly used for testing, without requiring a reference implementation as test oracle. As opposed to differential testing, our technique can be also applied when there only exists one implementation of a given semantics.

Fuzzing. Other test case generation techniques are based on fuzzing. Brummayer et al. apply fuzzing for testing SMT [18], SAT, and QBF solvers [19], while Cyrille et al. [9] and Niemetz et al. [30] target the solver's API. Note that [18] generates only quantifier-free formulas over fixed-size bit-vectors, thus a direct comparison with our work would not be meaningful. However, all these approaches generate inputs that may cause the solver to crash or may exhibit performance issues. As opposed to our approach, they do not have a test oracle, so they do not reliably detect soundness and completeness bugs. The existing techniques require delta debugging [33] to minimize the inputs that lead to a failure; in our case, the errors are usually found with formulas that are small by construction.

The closest related work to ours is StringFuzz [16], a state-of-the-art fuzzer and generator of SMT-LIB instances. StringFuzz can create input formulas with various properties (e.g., predefined number of variables, configurable depth of expressions, predefined length for string literals, etc.), but it has generators only for primitive string operations and regular expressions. We do not support regular expressions, but our formulas cover complex string operations. StringFuzz can also apply a set of transformations on already existing benchmarks, but very few of them guarantee equisatisifability. Using these formulas for soundness testing requires manually-written test oracles. In contrast, our synthesized formulas are sat or unsat by construction, and all our transformations preserve their satisfiability, therefore soundness testing is fully automatic.

Formal verification. Formal verification has been used to verify SAT and SMT algorithms [23, 24, 27], but not their implementations. SMT solvers are complex, highly-optimized software systems, thus formally verifying their implementation is very challenging. In contrast, our black-box testing technique can handle such complex implementations and can find bugs with minimal effort.

Validation and proof checking. A complementary body of work focuses on checking the proofs generated by the solvers. Zhang and Malik [34] synthesize a checker for validating the traces produced by a SAT solver during refutation proofs. Böhme and Weber [17] encode the proofs generated by Z3 in Isabelle, while Stump et al. [32] propose a meta-logic for describing and checking proofs for SMT. All these techniques require either modifications of the original solvers or translations of the proofs into other formats. Our technique treats the solvers under test as black boxes and does not depend on a specific implementation and proof format.

6 CONCLUSIONS

We have presented a novel technique for automatically generating SMT formulas from the string theory that are satisfiable or unsatisfiable by construction. These formulas are used as inputs for testing mostly the soundness of the implementation of a solver, but can also reveal completeness and performance issues. Our experimental evaluation shows that our approach effectively finds errors in the implementation of widely-used SMT solvers and is also applicable to automata-based solvers. We synthesize sat formulas together with models and unsat formulas together with minimal unsat cores; having increasing complexity, our inputs facilitate error localization and debugging. This paper focuses on strings, but the approach can be directly extended to other theories and their interactions. As future work, we plan to enhance our technique to also cover other classes of solvers and other components. For example, we plan to automatically test MAX-SMT solvers and the quantifier instantiation mechanism of an SMT solver. Applying our approach to testing decision procedures with machine-checkable proofs is another research direction we would like to explore in the future.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their insightful comments. We are also grateful to the developers of Z3-seq, Nikolaj Bjørner, of Z3str3, Murphy Berzish, of CVC4, Andrew Reynolds and Andres Nötzli, and of MT-ABC, William Eiers, for their detailed explanations and support. We also appreciate their valuable feedback on previous drafts of this paper. We are grateful to Vytautas Astrauskas and Arshavir Ter-Gabrielyan for their comments on our artifact.

REFERENCES

[1] [n.d.]. CVC4 Documentation for the String Theory. http://cvc4.cs.stanford.edu/wiki/Strings.

[2] [n.d.]. CVC4 Regression Test Suite. https://github.com/CVC4/CVC4/tree/master/test/regress.

[3] [n.d.]. MT-ABC Tested Version. https://github.com/vlab-cs-ucsb/ABC/commit/86b00141fddd183de7b9ae5c92c240e19dda1950.

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

[5] [n.d.]. SMT-LIB Unicode Strings Theory. http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml/.

[6] [n.d.]. StringFuzz Test Suite. http://stringfuzz.dmitryblotsky.com/problems/.

[7] [n.d.]. Z3 SMT Solver. https://github.com/Z3Prover/z3/.

[8] [n.d.]. Z3 Test Suite. https://github.com/Z3Prover/z3/tree/master/src/test.

[9] Cyrille Artho, Armin Biere, and Martina Seidl. 2013. Model-Based Testing for Verification Back-Ends. In *Tests and Proofs*, Margus Veanes and Luca Viganò (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 39–55.

[10] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-Based Model Counting for String Constraints. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 255–272.

[11] Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan, and Fang Yu. 2018. Parameterized Model Counting for String and Numeric Constraints. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 400–410. https://doi.org/10.1145/3236024.3236064

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

[13] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A String Solver with Theory-aware Heuristics. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. 55–59. https://doi.org/10.23919/FMCAD.2017.8102241

[14] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. 2009. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands.

[15] Nikolaj Bjørner, Vijay Ganesh, Raphaël Michel, and Margus Veanes. 2012. An SMT-LIB Format for Sequences and Regular Expressions. *Strings* (01 2012).

[16] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A Fuzzer for String Solvers. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 45–51.

[17] Sascha Böhme and Tjark Weber. 2010. Fast LCF-Style Proof Reconstruction for Z3. In *Interactive Theorem Proving*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 179–194.

[18] Robert Brummayer and Armin Biere. 2009. Fuzzing and delta-debugging SMT solvers. *ACM International Conference Proceeding Series* (01 2009), 1–5. https://doi.org/10.1145/1670412.1670413

[19] Robert Brummayer, Florian Lonsing, and Armin Biere. 2010. Automated Testing and Debugging of SAT and QBF Solvers. In *Theory and Applications of Satisfiability Testing – SAT 2010*, Ofer Strichman and Stefan Szeider (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–57.

[20] Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Automated Deduction – CADE-21*, Frank Pfenning (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–198.

[21] 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

[22] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—Where Programs Meet Provers. In *Programming Languages and Systems (ESOP) (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 125–128.

[23] Jonathan Ford and Natarajan Shankar. 2002. Formal Verification of a Combination Decision Procedure. In *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*. Springer-Verlag, Berlin, Heidelberg, 347–362. http://dl.acm.org/citation.cfm?id=648238.751562

[24] Stéphane Lescuyer and Sylvain Conchon. 2008. A Reflexive Formalization of a SAT Solver in Coq. In *In Proceedings of TPHOLs*.

[25] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2016. An Efficient SMT Solver for String Constraints. *Form. Methods Syst. Des.* 48, 3 (June 2016), 206–234. https://doi.org/10.1007/s10703-016-0247-6

[26] Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. 2014. A Model Counter for Constraints over Unbounded Strings. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 565–576. https://doi.org/10.1145/2594291.2594331

[27] Filip Mari. 2010. Formal Verification of a Modern SAT Solver by Shallow Embedding into Isabelle/HOL. *Theor. Comput. Sci.* 411, 50 (Nov. 2010), 4333–4356. https://doi.org/10.1016/j.tcs.2010.09.014

[28] William M. McKeeman. 1998. Differential Testing for Software. *DIGITAL TECHNICAL JOURNAL* 10, 1 (1998), 100–107.

[29] Tukaram Muske and Alexander Serebrenik. 2016. Survey of Approaches for Handling Static Analysis Alarms. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 157–166. https://doi.org/10.1109/SCAM.2016.25

[30] Aina Niemetz, Mathias Preiner, and Armin Biere. 2017. Model-Based API Testing for SMT Solvers. In *Proceedings of the 15th International Workshop on Satisfiability Modulo Theories, SMT 2017), affiliated with the 29th International Conference on Computer Aided Verification, CAV 2017, Heidelberg, Germany, July 24-28, 2017*, Martin Brain and Liana Hadarean (Eds.). 10 pages.

[31] Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. 2018. Revisiting Enumerative Instantiation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 112–131.

[32] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. 2013. SMT Proof Checking Using a Logical Framework. *Form. Methods Syst. Des.* 42, 1 (Feb. 2013), 91–118. https://doi.org/10.1007/s10703-012-0163-3

[33] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 183–200. https://doi.org/10.1109/32.988498

[34] Lintao Zhang and Sharad Malik. 2003. Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications. In *2003 Design, Automation and Test in Europe Conference and Exhibition*. 880–885. https://doi.org/10.1109/DATE.2003.1253717

[35] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. ACM, New York, NY, USA, 114–124. https://doi.org/10.1145/2491411.2491456