Validating SMT Solvers via Automatic Generation of Fusion Functions

Nicola Dardanis*
ndardanis@student.ethz.ch
ETH Zurich
Switzerland

Lucas Weitzendorf*
lweitzendorf@student.ethz.ch
ETH Zurich
Switzerland

Abstract

We introduce an effective and extensible method to automatically generate *fusion functions* to be used in *Semantic Fusion* for Satisfiability Modulo Theory (SMT) solver validation. The key idea of Semantic Fusion is to *fuse* two existing equisatisfiable formulas into a new formula, which combines the structures of its ancestors and preserves the satisfiability by construction. To this end, Semantic Fusion relies on *fusion functions*, whose main characteristic is invertibility.
We realized a fusion function generator tool which aims to overcome the main limitation of Semantic Fusion, namely the manual definition and design of fusion functions. The tool is able to generate functions of arbitrary complexity by nesting operators while guaranteeing the invertibility of the resulting function with respect to all of its arguments. The tool's domain can be extended by adding operators to its definition module along with their inversion rules. As the set of SMT-LIB operators is limited, the structure of our tool also allows for nested operator expressions to be defined as basic building blocks. This provides the ability to incorporate operators which might not be invertible on their own.

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

Keywords: Semantic fusion, SMT solvers, Fuzz testing, Mutation based testing, Code generation

1 Introduction

SMT solvers are extensively used in formal methods, most notably in software verification, systematic test case generation, and program synthesis. Due to their high degree of complexity, even mature solvers still contain many correctness issues. Their wide applicability amplifies the severity of any such issue. In particular, soundness bugs in SMT solvers betray their users' trust and can have severe consequences in safety-critical and security-critical domains.

*Both authors contributed equally to this research.

Semantic Fusion. Semantic Fusion is a general and effective approach to validate SMT solvers. It is a metamorphic testing approach that can work with a single SMT solver. The basic idea behind fusion is to fuse formula pairs into a new formula of known satisfiability (either both sat or both unsat). Given two seed formulas ϕ_1, ϕ_2, and variables x, y of ϕ_1 and ϕ_2 and respectively, the idea is to

1. Concatenate the formulas ϕ_1 and ϕ_2
2. Add a fresh variable $z = f(x, y)$
3. Replace random occurences of $x = g_x(y)$ and $y = g_y(x)$ within the concatenated formula.

f is called a *fusion function* and g_x, g_y are the respective inversion functions for variable x, y. The major limitation of *Semantic Fusion*, as noted by the authors, is to manually design these functions. This approach has been implemented in YinYang [7], an SMT fuzzing tool.

SMT-LIB Language. The SMT-LIB language [3] is the current standard input language for SMT solvers and the base format used in the fusion function specification [6] for YinYang [7]. We focus on the following statements of the SMT-LIB language: `declare-fun`, `declare-const`, `assert`, `check-sat`. Variables are declared as zero-valued functions. For example, the statement (declare−fun x () Int) declares variable x of type integer. An `assert` statement specifies constraints. The predicates within the asserts can be of mixed types, e.g., the assertion (assert (= (/ x 10) (∗ 5 x))) includes predicates of real and Boolean types. Operations are specified in the prefix notation. Multiple asserts are translated as the conjunction of the constraints in each individual assert statement. The `check-sat` statement queries the solver on the satisfiability of a formula. The formula is satisfiable if all constraints are satisfiable; otherwise, the formula is unsatisfiable.

Fusion Function Generation. We introduce an effective and general approach to automatically generate arbitrarily complex fusion functions. Our key insight is to treat the problem at an operator level by encoding rewriting rules for each operator. Not all operators are invertible, so we restrict each theory to the subset of operators for which an inverse function can be defined for each input. When operators are not invertible at the SMT solver level, the definition structure of our tool is able to treat invertible compound operators as basic building blocks.

Results. We have engineered a generator for fusion functions which is capable of producing configuration files for YinYang that are generally able to achieve better code coverage on widely used SMT solvers than its default configuration. Moreover, with a total running period of 2 weeks, we have collected and reported 4 confirmed bugs in Z3 [11], a state-of-the-art SMT solver.

Main Contributions.

- We introduce an effective technique to automatically generate arbitrarily complex fusion functions.
- We implement this technique in an efficient generator tool, which is able to handle multiple theories and is conveniently extensible. To this end, we provide a utility to automatically generate template code for operator definitions from a simple entry in its configuration module.
- We modify YinYang to support a wider range of fusion functions.
- We extensively benchmark generated fusion functions against vanilla YinYang using code coverage metrics calculated with gcov.

Paper organization. The rest of the paper is structured as follows. Section 2 illustrates the high-level idea behind our approach and formalizes it. A description of the implementation and the algorithms is also given. Next, we provide details on our evaluation strategy and results (Section 3). Finally, in Section 4 we give a survey on related work.

2 Approach

This section presents two examples of rewriting rules that our tool is capable of handling. *Atomic Rewriting* produces the inversions of an atomic operator, *Expression Rewriting* uses a nested operator expression to represent more complex rewriting rules. In the end we give a more complete example of fusion function expressed as a tree of operators. A complete description of the idea and implementation of the tree rewriting algorithm as well as the tree generation algorithm follows.

2.1 Atomic Rewriting

Atomic Rewriting simply represents the inverse of a SMT operator as another unique SMT operator. Consider the real addition operator (= z (+ x y)), the inverses are uniquely determined: (= x (− z y)), (= y (− z x).

2.2 Expression Rewriting

As not every inverse has a direct SMT representation, we allow rewriting rules to be made up of more complex nested operator expressions. These inverses are not necessarily unique. For example, string concatenation (= z (str.++ x y)) can be inverted with respect to y using at least three different expressions containing different operators:

1. (= y (str.replace z x ""))
2. (= y (str.substr z (str.len x) (str.len y)))
3. (= y (str.substr z (str.indexof z y (str.len x)) (str.len y)))

2.3 Tree Rewriting

A fusion function can be naturally represented by a tree of operators and their inputs. The internal nodes of such a tree consist of simple operators, while leaves are made up of constants and free variables, serving as inputs to the operators. The root of the tree represents the output of the fusion function. Each edge of the tree represents the result of the sub-tree rooted in the child node, thus having the type of the output of the child node operator. The tree will be subject to the following constraints:

1. Each operator node has a defined inverse operation.
2. Each free variable only appears in a single leaf.
3. Each edge is type-consistent, i.e. its output type matches the input type of its parent.

From these three properties, we can guarantee that there exists a unique path from the root to each free variable. Informally speaking, the unique path property allows us to "undo" each operation in order to retrieve an expression for the variable values from the root output without introducing self-dependencies between branches. This allows us to automatically infer the inverse of the tree w.r.t. each free variable. More specifically, we inductively apply syntactic rewriting rules along the unique path from root to leaf.

Fulfilling the first tree constraint requires manual specification. For a given solver theory, we identify basic invertible operators along with their atomic and Expression Rewriting rules. These basic operators can then be used as operator nodes. The tool is also capable of treating expressions made of more than one basic operator at the SMT solver level as a unique node if the inverse relation for the whole expression is given. This allows it to handle theories in which the basic SMT operators aren't already invertible. To this end, we encode a definition of invertible operators for each theory in a module inside the tool.

Algorithm 1 invert operator tree

```
procedure INVERT($t$, $out$)
    $inv\_map \leftarrow \{\}$
    for $sub\_t \in children(t)$ do
        $inv\_op \leftarrow t.get\_inverse\_op(sub\_t, out)$
        $sub\_map \leftarrow invert(sub\_t, inv\_op)$
        $inv\_map \leftarrow merge(inv\_map, sub\_inv)$
    end for
    return $inv\_map$
end procedure
```

Fusion functions in two variables are a triplet of functions $f(x, y) = z$, $f^{-1}(y, z) = x$ and $f^{-1}(x, z) = y$. We use a recursive approach to make use of these inverses along the tree.

Since the inverses are always defined in terms of the output, we provide it as the argument *out*. To keep track of multiple inverses, we map variable names to their inverses. Inverse trees are collected on down-traversal using *out* and kept track of in *inv_map* by merging maps of multiple branches on the way back up the tree.

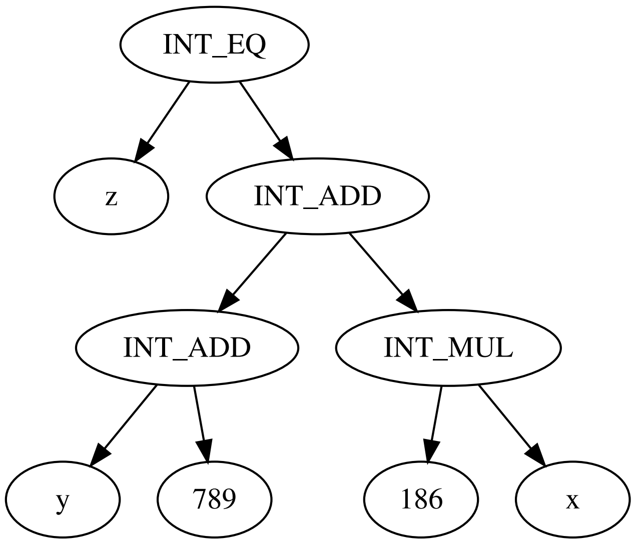

Figure 1. Initial tree

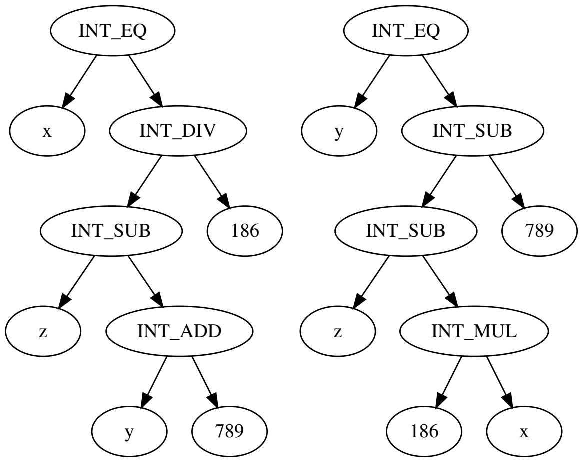

Figure 2. Inverses

2.4 Tree Generation

Generation of the tree is separated into generation of the tree structure and operator selection.

Algorithm 2 generate operator tree

```
$a \leftarrow available\_arities(theory)$
$t\_a \leftarrow get\_arity\_tree(size, a, len(vars\_in))$
$t\_o \leftarrow get\_operator\_tree(t\_a, vars\_in, var\_out)$
return $t\_o$
```

We store the tree structure as a list of branching factors in level order and refer to this as the arity tree. The following algorithms are slight simplifications which disregard certain implementation-specific details. The generation algorithm is given a list of available *arities* of the theory we use, desired *size* of the tree and the minimum number of leaves *min_lf* as we need them to use all of the specified variables.

In each loop iteration, one node is added to the tree until we reach the desired size. The number of currently expected nodes is the sum of arities in the tree + 1 for the root. As this should never be higher than *size*, we subtract it from the desired size to obtain the highest branching factor *max_a* we can use for the next operator. Calculating the lower bound is more complex. We need to make sure that the operator enables us to place sufficiently many leaves in the tree. We therefore calculate the hypothetical number of leaves we would be able to generate if all subsequent operators use the maximum available arity as *max_sub_lf* and the number of empty subtrees of existing operators (which could all be leaves if needed) as *node_gap*. We then subtract both of these from *rem_lf* to get the lower bound *min_a*. Finally, we simply need to select a random arity between *min_a* and *max_a* and add it to the tree. The only scenario in which we can't use a leaf is if it would complete the last subtree and subsequently lead to $len(tree) < size$.

Algorithm 3 generate arity tree

```
$t \leftarrow []$
$rem\_lf \leftarrow min\_lf$
while $len(t) < size$ do
    $max\_a \leftarrow size - sum(t) - 1$
    $rem\_op \leftarrow size - len(t) - rem\_lf$
    $max\_sub\_lf \leftarrow (rem\_op - 1) * (max(arities) - 1) + 1$
    $node\_gap \leftarrow sum(t) - len(t)$
    $min\_a \leftarrow rem\_lf - node\_gap - max\_sub\_lf + 1$
    $a\_opt \leftarrow arities.filter(min\_a...max\_a)$
    if $!(sum(t) == len(t) \&\& max\_a > 0)$ then
        $a\_opt.add(0)$
    end if
    $arity \leftarrow random\_choice(a\_opt)$
    if $arity == 0$ then
        $rem\_lf \leftarrow rem\_lf - 1$
    end if
    t.add(arity)
end while
return $t$
```

Once the generation of the arity tree is done, selecting operators to match is fairly straightforward. As the number of leaves can be obtained from the arity tree and the number of variables is fixed, the ratio of constants to variables is well-defined. We use a shuffled list of Booleans to easily obtain a random distribution of them in the tree. Then, we simply need to choose a random operator with the given arity at each step and recursively initialize each subtree before using them as input to the operator. At the end, we wrap the operator tree in an equality for our output variable.

Algorithm 4 choose operators

```
  $rem\_leaves \leftarrow t\_a.count(0)$
  $rem\_vars \leftarrow len(vars\_in)$
  $rem\_const \leftarrow rem\_leaves - rem\_vars$
  $leaves[1 : rem\_const] \leftarrow false$
  $leaves[rem\_const + 1 : rem\_leaves] \leftarrow true$
  $leaves \leftarrow shuffle(leaves)$
  $t\_o, i = recursive\_generation(0)$
  **return** $Equality(Variable(var\_out), t\_o)$

  **procedure** $recursive\_generation(i)$
      $arity \leftarrow t\_a[i]$
      $i \leftarrow i + 1$
      **if** $arity == 0$ **then**
          $rem\_leaves \leftarrow rem\_leaves - 1$
          **if** $leaves[rem\_leaves]$ **then**
              $rem\_vars \leftarrow rem\_vars - 1$
              $name \leftarrow vars_{in}[rem\_vars]$
              **return** $Variable(name), i$
          **else**
              $rem\_const \leftarrow rem\_const - 1$
              $name \leftarrow "c" + rem\_const$
              **return** $Constant(name), i$
          **end if**
      **else**
          $children \leftarrow []$
          **for** $j \in 1...arity$ **do**
              $sub\_node, i \leftarrow recursive\_generation(i)$
              $children.add(sub\_node)$
          **end for**
          $node \leftarrow get\_random\_operator(arity)$
          $node.set\_children(children)$
          **return** $node, i$
      **end if**
  **end procedure**
```

2.5 Implementation Details and Technical Challenges

In order to implement the generation and inversion in an intuitive way, we have made several important design decisions which will be discussed in this section.

Nested Operator Classes. Just like YinYang, our fusion function generator is implemented Python, which provides us with the ability to represent operators as objects. By defining the parameters an operator takes as members of the class, we can easily nest these classes to represent a tree. This reification of the operators also allows us to easily represent arbitrarily complex expressions as a single operator, thus allowing better support for Expression Rewriting.

Tree Visitors. Operator classes provide us with a definition of the basic building blocks for generation. In order to decouple functionality from definition, we decided to implement actions on the operators in separate visitor classes, which also allows code reuse without imposing hierarchical constraints. We implemented visitors for inversion, printing, export to DOT, and emission to fusion function files. These visitors might be used as follows:

```python
inverses = tree.accept(RewriteVisitor())
print_visitor = PrintVisitor()
for inv in inverses:
    print(inv.accept(print_visitor))
```

Here is an example of how these visitors are implemented for integer addition:

```python
def visit_integer_addition(self, operator: IntegerAddition):
    output = self.output[operator]
    self.output[operator.operator_1] =
        IntegerSubtraction(output, operator.operator_2)
    self.output[operator.operator_2] =
        IntegerSubtraction(output, operator.operator_1)
    inverse_1 = operator.operator_1.accept(self)
    inverse_2 = operator.operator_2.accept(self)
    return {**inverse_1, **inverse_2}
```

RewriteVisitor

```python
def visit_integer_addition(self, operator: IntegerAddition):
    return f"(+␣{operator.operator_1.accept(self)}␣" \
           f"{operator.operator_2.accept(self)})"
```

PrintVisitor

Extensibility. This unified way of treating classes and providing functionality in visitors, however, leads to a lot of boilerplate code. As previously mentioned, our tool uses an easy-to-use generation module, which can be easily extended with new operators by simply specifying the name, input types and output type. Once done, we provide a generator to automatically generate definitions for the operator classes and stubs for relevant visitor implementations. The principle behind this architectural choice is the "single source of truth". More precisely, by generating the operator classes and by decoupling the algorithms from the data as described above, we are able to derive all the configuration and boilerplate needed to run the generation algorithm from a single point of truth, thus again providing better support for further extensions. The core generation algorithm is indeed capable of using any new operator automatically as soon as it is included in the configuration file and the basic relevant visitors are implemented to provide rewriting rules and string representation of the new operation.

YinYang Extensions. Since our tool is able to generate fusion functions of arbitrary size, we fill all leaves that are not variables with symbolic or literal constants. We therefore modified YinYang to handle fusion functions containing multiple symbolic constants, allowing for greater flexibility and delaying the instantiation of actual values to happen at runtime. Furthermore, to enable better error tracking, we modified the tool to use the name of both seeds in the mutant file name.

3 Empirical Evaluation

In this section we present the results of our evaluation of the generator. We first explore whether the generated fusion functions can enhance Semantic Fusion by comparing coverage information obtained both with the original functions and some generated ones. Second, we provide an overview of the bugs we obtained during 2 weeks of fuzzing using generated fusion functions.

3.1 Do generated fusion functions improve Semantic Fusion coverage?

To evaluate our generator, we compare the coverage, a conventional evaluation metric for software testing, obtained by fuzzing for a limited amount of time with YinYang. We run the fuzzer using a sample of generated fusion functions of different sizes and the ones from the original (vanilla) implementation.

Test Seed Formulas. The seeds have already been classified and are available as an open source project on GitHub [1]. The repository contains 114615 files in SMT-LIB format. We choose the following logics: LIA, LRA, NRA, QF_LIA, QF_LRA, QF_NRA, QF_SLIA, and QF_S, for a total of 38459 seeds where L = linear, N = non-linear, IA = integer arithmetic, RA = real arithmetic, QF = quantifier-free and S = string logic. However, two seeds are misclassfied as of June 4 2022. We corrected their classifications before each test run to avoid false positives.

SMT Solvers. We selected the SMT solvers Z3 [11] and CVC5 [2] for the evaluation for the following reasons:

- Z3 (7,498 stars on GitHub) and CVC5 (615 stars on GitHub) are the two most popular solvers. They are widely used in academia and industry.
- Z3 and CVC5 support most of the logics and features in the SMT-LIB standard.
- Z3 and CVC5 have open source issue trackers on GitHub with active and responsive developers.

The solvers are compiled in debug mode without optimizations. The coverage measurement tool is Gcov [8]. In CVC5, we use the `--strings-exp` option to enable support for string logic and `smt.string_solver=z3str3` in Z3. For every other logic, we use the solvers' default configuration.

Fusion Functions. We want to explore how function size affects coverage, the we generate 6 configuration files, each containing 60 functions, with the following sizes (number of nodes): 5, 10, 20, 25, 30, 50.

YinYang. For each fusion function configuration file and logic we run YinYang with two solver timeouts: 30 and 60 seconds. It is indeed possible that, given functions of different sizes, the solver is unable to solve the bigger test case within the time allocated. We bound the execution of the fuzzer for 1 and 2 hours to compare the different results. When running for 1 hour we allocate 30 seconds as timeout for each call to the solver and 60 seconds for the 2 hour long run.

Infrastructure. The experiments are conducted in a Dockerized environment and the project is published as an open source project on GitHub [5]. In the benchmarking folder, we provide the Dockerfile used to build the image containing all the dependencies and utilities required to run the tests, collect and analyse the results. Moreover, we provide the fusion functions used for the benchmarks. Z3 was compiled at commit sha `ea365de`, 4 June 2022, while CVC5 was compiled at commit sha `2abc3c30`, 4 June 2022. To use these exact heads, the Dockerfile should be slightly modified as it is configured to always use the latest commit.

Results. Table 1, Table 2, Table 3 and Table 4 show the results. The numbers are all expressed in percentages (%) of line (l), functions (f) and branch (b) coverage respectively. The highest coverages are shaded. The various run configurations used are identified as follows:

- `Vanilla - 30`: 1 hour long run of the original YinYang and fusion functions with a timeout of 30 seconds for the solver.
- `Vanilla - 60`: 2 hour long run of the original YinYang and fusion functions with a timeout of 60 seconds for the solver.
- `FFG-5 - 30`: 1 hour long run of the modified YinYang with fusion functions of size 5 and a timeout of 30 seconds for the solver.
- `FFG-5 - 60`: 2 hour long run of the modified YinYang with fusion functions of size 5 and a timeout of 60 seconds for the solver.
- `FFG-10 - 30`: 1 hour long run of the modified YinYang with fusion functions of size 10 and a timeout of 30 seconds for the solver.
- `FFG-10 - 60`: 2 hour long run of the modified YinYang with fusion functions of size 10 and a timeout of 60 seconds for the solver.
- `FFG-20 - 30`: 1 hour long run of the modified YinYang with fusion functions of size 20 and a timeout of 30 seconds for the solver.
- `FFG-20 - 60`: 2 hour long run of the modified YinYang with fusion functions of size 20 and a timeout of 60 seconds for the solver.

Table 1. Code coverage evaluations (LIA, LRA). The numbers represent the percentage (%) coverage for the corresponding coverage metric. Column l, f, b represent line coverage, function coverage, and branch coverage respectively. Vanilla is the original version of YinYang. FFG indicates the modified version of YinYang and the usage of one of the fusion function files generated for benchmarking. The numbers represent the number of nodes per fusion function and the SMT solver timeout in seconds.

		LIA						**LRA**					
		SAT			UNSAT			SAT			UNSAT		
		l	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*
CVC5	*FFG-10 - 30*	*19.9*	*32.7*	*5.9*	*19.9*	*33.1*	*5.9*	*15.2*	*28.3*	*4.0*	*2.7*	*5.8*	*0.6*
	FFG-10 - 60	*18.7*	*31.8*	*5.5*	*20.6*	*33.5*	*6.2*	*18.4*	*31.9*	*5.3*	*2.7*	*5.8*	*0.6*
	FFG-20 - 30	*17.9*	*30.9*	*5.1*	*19.9*	*33.0*	*5.9*	*15.6*	*28.5*	*4.2*	*2.7*	*5.8*	*0.6*
	FFG-20 - 60	*17.9*	*30.8*	*5.2*	*19.5*	*32.7*	*5.8*	*19.0*	*31.9*	*5.5*	*2.7*	*5.8*	*0.6*
	FFG-25 - 30	*20.2*	*32.9*	*6.1*	*20.0*	*33.1*	*6.0*	*17.7*	*30.9*	*5.0*	*2.7*	*5.8*	*0.6*
	FFG-25 - 60	*17.8*	*30.6*	*5.1*	*20.1*	*33.2*	*6.0*	*19.1*	*32.1*	*5.6*	*2.7*	*5.8*	*0.6*
	FFG-30 - 30	*17.9*	*30.9*	*5.1*	*19.8*	*32.9*	*5.9*	*19.0*	*32.1*	*5.5*	*2.7*	*5.8*	*0.6*
	FFG-30 - 60	*19.5*	*32.6*	*5.7*	*20.1*	*33.2*	*6.0*	*18.0*	*31.6*	*5.1*	*2.7*	*5.8*	*0.6*
	FFG-5 - 30	*17.9*	*30.9*	*5.2*	*19.8*	*33.1*	*5.9*	*15.9*	*28.9*	*4.3*	*2.7*	*5.8*	*0.6*
	FFG-5 - 60	*19.6*	*32.7*	*5.8*	*20.1*	*33.3*	*6.1*	*17.8*	*31.3*	*5.0*	*2.7*	*5.8*	*0.6*
	FFG-50 - 30	*17.9*	*30.9*	*5.1*	*18.5*	*31.6*	*5.4*	*18.4*	*31.5*	*5.3*	*2.7*	*5.8*	*0.6*
	FFG-50 - 60	*18.3*	*31.1*	*5.3*	*18.8*	*32.1*	*5.5*	*18.7*	*32.0*	*5.4*	*2.7*	*5.8*	*0.6*
	Vanilla - 30	*19.6*	*33.0*	*5.8*	*19.6*	*32.9*	*5.8*	*19.1*	*32.4*	*5.5*	*2.7*	*5.8*	*0.6*
	Vanilla - 60	*19.1*	*32.7*	*5.6*	*19.8*	*33.0*	*5.9*	*17.5*	*30.9*	*4.9*	*2.7*	*5.8*	*0.6*
Z3	*FFG-10 - 30*	*15.3*	*20.1*	*4.5*	*16.7*	*21.6*	*4.9*	*12.8*	*17.5*	*3.3*	*14.2*	*18.2*	*3.9*
	FFG-10 - 60	*15.3*	*20.0*	*4.5*	*18.1*	*22.9*	*5.3*	*12.9*	*17.6*	*3.4*	*13.6*	*17.9*	*3.7*
	FFG-20 - 30	*15.1*	*19.9*	*4.4*	*19.2*	*23.9*	*5.7*	*13.3*	*17.8*	*3.5*	*13.5*	*17.9*	*3.7*
	FFG-20 - 60	*11.7*	*16.6*	*3.2*	*19.3*	*23.9*	*5.7*	*13.2*	*17.8*	*3.5*	*13.4*	*17.8*	*3.6*
	FFG-25 - 30	*15.2*	*20.0*	*4.5*	*19.9*	*24.3*	*6.0*	*12.8*	*17.5*	*3.4*	*13.9*	*18.0*	*3.8*
	FFG-25 - 60	*15.0*	*19.9*	*4.4*	*17.6*	*22.5*	*5.2*	*12.9*	*17.6*	*3.4*	*14.2*	*18.2*	*3.9*
	FFG-30 - 30	*15.1*	*20.0*	*4.4*	*19.8*	*24.2*	*5.9*	*12.8*	*17.5*	*3.4*	*13.6*	*17.9*	*3.7*
	FFG-30 - 60	*15.3*	*20.0*	*4.5*	*20.2*	*24.4*	*6.1*	*13.0*	*17.7*	*3.4*	*13.1*	*17.6*	*3.5*
	FFG-5 - 30	*15.2*	*20.0*	*4.5*	*17.3*	*22.2*	*5.1*	*12.9*	*17.6*	*3.4*	*13.2*	*17.7*	*3.5*
	FFG-5 - 60	*15.4*	*20.1*	*4.5*	*17.3*	*22.2*	*5.1*	*12.9*	*17.6*	*3.4*	*14.0*	*18.1*	*3.8*
	FFG-50 - 30	*11.5*	*16.4*	*3.1*	*14.5*	*19.6*	*4.1*	*13.1*	*17.6*	*3.5*	*14.2*	*18.2*	*3.8*
	FFG-50 - 60	*10.6*	*15.8*	*2.7*	*14.6*	*19.5*	*4.1*	*12.8*	*17.5*	*3.4*	*13.5*	*17.8*	*3.6*
	Vanilla - 30	*15.4*	*20.1*	*4.5*	*16.4*	*21.4*	*4.8*	*13.0*	*17.6*	*3.4*	*13.5*	*17.8*	*3.6*
	Vanilla - 60	*11.6*	*16.5*	*3.1*	*16.3*	*21.4*	*4.8*	*12.9*	*17.7*	*3.4*	*13.9*	*18.1*	*3.8*

- `FFG-25 - 30`: 1 hour long run of the modified YinYang with fusion functions of size 25 and a timeout of 30 seconds for the solver.
- `FFG-25 - 60`: 2 hour long run of the modified YinYang with fusion functions of size 25 and a timeout of 60 seconds for the solver.
- `FFG-30 - 30`: 1 hour long run of the modified YinYang with fusion functions of size 30 and a timeout of 30 seconds for the solver.
- `FFG-30 - 60`: 2 hour long run of the modified YinYang with fusion functions of size 30 and a timeout of 60 seconds for the solver.
- `FFG-50 - 30`: 1 hour long run of the modified YinYang with fusion functions of size 50 and a timeout of 30 seconds for the solver.
- `FFG-50 - 60`: 2 hour long run of the modified YinYang with fusion functions of size 50 and a timeout of 60 seconds for the solver.

Table 2. Code coverage evaluations (NRA, QF_LIA).

		NRA						QF_LIA					
		SAT			UNSAT			SAT			UNSAT		
		l	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*
CVC5	*FFG-10 - 30*	*0.6*	*2.0*	*0.1*	*2.7*	*5.8*	*0.6*	*17.1*	*29.9*	*4.8*	*14.3*	*27.1*	*3.9*
	FFG-10 - 60	*0.6*	*2.0*	*0.1*	*2.7*	*5.8*	*0.6*	*17.6*	*30.5*	*5.1*	*14.5*	*27.3*	*4.0*
	FFG-20 - 30	*0.6*	*2.0*	*0.1*	*2.7*	*5.8*	*0.6*	*17.5*	*30.4*	*5.0*	*13.7*	*26.2*	*3.7*
	FFG-20 - 60	*0.6*	*2.0*	*0.1*	*2.7*	*5.8*	*0.6*	*17.3*	*30.2*	*4.9*	*14.4*	*27.3*	*3.9*
	FFG-25 - 30	*0.6*	*2.0*	*0.1*	*2.7*	*5.8*	*0.6*	*17.0*	*30.2*	*4.8*	*14.3*	*27.2*	*3.9*
	FFG-25 - 60	*0.6*	*2.0*	*0.1*	*2.7*	*5.8*	*0.6*	*17.4*	*30.3*	*5.0*	*14.3*	*27.1*	*3.9*
	FFG-30 - 30	*0.6*	*2.0*	*0.1*	*2.7*	*5.8*	*0.6*	*17.1*	*30.2*	*4.8*	*14.5*	*27.2*	*3.9*
	FFG-30 - 60	*0.6*	*2.0*	*0.1*	*2.7*	*5.8*	*0.6*	*17.6*	*30.5*	*5.0*	*13.3*	*25.6*	*3.5*
	FFG-5 - 30	*0.6*	*2.0*	*0.1*	*2.7*	*5.8*	*0.6*	*16.9*	*30.1*	*4.7*	*14.5*	*27.3*	*4.0*
	FFG-5 - 60	*0.6*	*2.0*	*0.1*	*2.7*	*5.8*	*0.6*	*17.6*	*30.5*	*5.1*	*14.6*	*27.4*	*4.0*
	FFG-50 - 30	*0.6*	*2.0*	*0.1*	*2.7*	*5.8*	*0.6*	*13.8*	*26.5*	*3.5*	*14.0*	*26.7*	*3.8*
	FFG-50 - 60	*0.6*	*2.0*	*0.1*	*2.7*	*5.8*	*0.6*	*14.4*	*27.4*	*3.8*	*13.9*	*26.8*	*3.7*
	Vanilla - 30	*0.6*	*2.0*	*0.1*	*2.7*	*5.8*	*0.6*	*16.6*	*29.8*	*4.6*	*14.5*	*27.3*	*3.9*
	Vanilla - 60	*0.6*	*2.0*	*0.1*	*2.7*	*5.8*	*0.6*	*17.1*	*30.3*	*4.8*	*14.6*	*27.4*	*4.0*
Z3	*FFG-10 - 30*	*9.9*	*12.8*	*2.6*	*9.9*	*12.5*	*2.6*	*16.8*	*21.0*	*4.8*	*12.9*	*17.3*	*3.5*
	FFG-10 - 60	*9.7*	*12.7*	*2.5*	*10.1*	*12.9*	*2.7*	*16.0*	*20.4*	*4.6*	*12.7*	*17.2*	*3.4*
	FFG-20 - 30	*8.9*	*12.3*	*2.2*	*9.7*	*12.4*	*2.6*	*15.1*	*19.7*	*4.2*	*13.0*	*17.3*	*3.5*
	FFG-20 - 60	*0.4*	*0.4*	*0.1*	*9.6*	*12.4*	*2.5*	*16.0*	*20.3*	*4.6*	*12.9*	*17.3*	*3.5*
	FFG-25 - 30	*9.2*	*12.4*	*2.4*	*9.7*	*12.5*	*2.6*	*15.8*	*20.3*	*4.5*	*12.8*	*17.3*	*3.5*
	FFG-25 - 60	*10.0*	*12.8*	*2.6*	*9.6*	*12.4*	*2.5*	*17.0*	*21.0*	*4.9*	*13.6*	*17.6*	*3.7*
	FFG-30 - 30	*9.7*	*12.7*	*2.5*	*9.6*	*12.4*	*2.5*	*8.8*	*13.6*	*2.2*	*12.8*	*17.2*	*3.4*
	FFG-30 - 60	*9.7*	*12.7*	*2.5*	*9.9*	*12.5*	*2.6*	*16.4*	*20.6*	*4.7*	*13.2*	*17.4*	*3.6*
	FFG-5 - 30	*9.9*	*12.8*	*2.6*	*10.5*	*13.4*	*2.8*	*17.0*	*21.0*	*4.9*	*12.7*	*17.2*	*3.4*
	FFG-5 - 60	*9.6*	*12.6*	*2.5*	*10.1*	*12.9*	*2.7*	*16.4*	*20.5*	*4.7*	*12.8*	*17.2*	*3.4*
	FFG-50 - 30	*9.8*	*12.8*	*2.6*	*9.5*	*12.3*	*2.5*	*10.5*	*15.6*	*2.8*	*9.2*	*13.4*	*2.4*
	FFG-50 - 60	*9.6*	*12.6*	*2.5*	*9.6*	*12.3*	*2.5*	*10.5*	*15.6*	*2.7*	*9.4*	*13.6*	*2.5*
	Vanilla - 30	*10.0*	*12.8*	*2.6*	*10.5*	*13.2*	*2.8*	*16.8*	*21.1*	*4.8*	*13.0*	*17.4*	*3.5*
	Vanilla - 60	*10.0*	*12.9*	*2.6*	*9.9*	*12.5*	*2.6*	*17.2*	*21.4*	*5.0*	*12.6*	*17.1*	*3.3*

We can observe that the generator produces fusion functions which, compared to manually designed ones, achieves greater or equal code coverage on average. These improvements are mostly in the order of 1%, which is still a considerable improvement given the large code bases of the two solvers (Z3 consists of more than 487K LOC and CVC5 has over 369K LOC). We notice that in the case of CVC5 over QF_SLIA unsat seeds, the line coverage of the best run is improved by 2.6% line coverage, 3.8% for function coverage and 1% for branch coverage compared to the run of vanilla YinaYang with the same time and timeout parameters. This suggest that our Expression Rewriting of non-trivial operators allows for good flexibility and that there is a lot of room for improvement, especially for theories where the technique can be used extensively to include more and more operators. We also notice that, to the best of our knowledge, all the functions that have been designed for YinYang up until now are eventually derivable from our generator by selecting the right amount of nodes to be used in the formula. This can also be seen by the similarity of results achieved on average by FFG-5 and vanilla tests. Indeed, our generator is eventually capable of producing all fusion functions of a given size over a fixed set of operators. More generally, we can observe that when the vanilla case achieves better coverage it is almost always limited to 0.1-0.2% compared to the best run of the generated functions.

Table 3. Code coverage evaluations (QF_LRA, QF_NRA).

		QF_LRA						QF_NRA					
		SAT			UNSAT			SAT			UNSAT		
		l	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*
CVC5	*FFG-10 - 30*	*16.8*	*29.5*	*4.7*	*13.3*	*25.8*	*3.5*	*16.7*	*29.1*	*4.7*	*16.1*	*28.7*	*4.4*
	FFG-10 - 60	*16.8*	*29.5*	*4.7*	*13.3*	*25.8*	*3.5*	*16.1*	*28.6*	*4.4*	*16.6*	*29.4*	*4.6*
	FFG-20 - 30	*16.8*	*29.5*	*4.7*	*13.3*	*25.8*	*3.5*	*16.7*	*29.4*	*4.6*	*16.2*	*28.9*	*4.5*
	FFG-20 - 60	*16.8*	*29.5*	*4.7*	*13.3*	*25.8*	*3.5*	*16.8*	*29.5*	*4.7*	*16.6*	*29.4*	*4.6*
	FFG-25 - 30	*16.9*	*29.6*	*4.7*	*13.2*	*25.8*	*3.5*	*16.4*	*28.7*	*4.6*	*16.2*	*28.9*	*4.5*
	FFG-25 - 60	*16.8*	*29.5*	*4.7*	*13.3*	*25.8*	*3.5*	*16.7*	*29.1*	*4.7*	*16.6*	*29.4*	*4.6*
	FFG-30 - 30	*16.8*	*29.5*	*4.7*	*13.2*	*25.8*	*3.5*	*16.3*	*28.5*	*4.5*	*16.6*	*29.0*	*4.6*
	FFG-30 - 60	*16.8*	*29.5*	*4.7*	*13.2*	*25.8*	*3.5*	*16.3*	*29.2*	*4.5*	*16.4*	*29.3*	*4.5*
	FFG-5 - 30	*16.8*	*29.5*	*4.7*	*13.3*	*25.8*	*3.5*	*16.6*	*29.0*	*4.6*	*16.3*	*29.0*	*4.5*
	FFG-5 - 60	*16.8*	*29.5*	*4.7*	*13.4*	*25.9*	*3.5*	*16.7*	*29.5*	*4.7*	*16.6*	*29.5*	*4.6*
	FFG-50 - 30	*16.6*	*29.0*	*4.6*	*13.0*	*25.6*	*3.4*	*16.2*	*28.6*	*4.5*	*16.5*	*29.3*	*4.6*
	FFG-50 - 60	*16.7*	*29.1*	*4.7*	*13.0*	*25.7*	*3.4*	*16.6*	*29.0*	*4.6*	*16.6*	*29.4*	*4.6*
	Vanilla - 30	*16.7*	*29.6*	*4.7*	*13.4*	*25.9*	*3.6*	*16.5*	*28.9*	*4.6*	*16.2*	*28.9*	*4.5*
	Vanilla - 60	*16.7*	*29.5*	*4.7*	*13.5*	*26.0*	*3.6*	*16.6*	*29.1*	*4.6*	*16.5*	*29.4*	*4.6*
Z3	*FFG-10 - 30*	*11.7*	*15.8*	*3.0*	*12.4*	*17.1*	*3.3*	*16.3*	*20.3*	*4.6*	*15.3*	*19.1*	*4.3*
	FFG-10 - 60	*11.5*	*15.7*	*2.9*	*12.3*	*16.9*	*3.3*	*16.2*	*20.2*	*4.6*	*15.5*	*19.1*	*4.3*
	FFG-20 - 30	*10.9*	*15.5*	*2.8*	*12.1*	*16.5*	*3.2*	*15.9*	*20.0*	*4.5*	*15.2*	*19.0*	*4.2*
	FFG-20 - 60	*11.8*	*15.8*	*3.0*	*12.5*	*17.2*	*3.3*	*16.2*	*20.2*	*4.6*	*15.6*	*19.2*	*4.4*
	FFG-25 - 30	*11.6*	*15.8*	*3.0*	*12.1*	*16.8*	*3.2*	*16.2*	*20.3*	*4.6*	*15.1*	*18.9*	*4.2*
	FFG-25 - 60	*11.5*	*15.7*	*3.0*	*12.4*	*17.1*	*3.3*	*16.3*	*20.4*	*4.6*	*16.0*	*20.0*	*4.5*
	FFG-30 - 30	*11.6*	*15.8*	*2.9*	*12.2*	*16.8*	*3.2*	*16.0*	*20.1*	*4.5*	*9.7*	*12.3*	*2.6*
	FFG-30 - 60	*10.9*	*15.4*	*2.7*	*12.3*	*16.9*	*3.3*	*16.1*	*20.2*	*4.6*	*15.6*	*19.2*	*4.4*
	FFG-5 - 30	*10.8*	*15.3*	*2.7*	*12.3*	*16.9*	*3.2*	*16.2*	*20.3*	*4.6*	*14.8*	*18.1*	*4.2*
	FFG-5 - 60	*10.8*	*15.3*	*2.7*	*12.5*	*17.2*	*3.3*	*16.3*	*20.3*	*4.6*	*15.5*	*19.2*	*4.4*
	FFG-50 - 30	*11.1*	*15.4*	*2.8*	*11.9*	*16.4*	*3.1*	*15.5*	*19.8*	*4.3*	*12.9*	*16.9*	*3.4*
	FFG-50 - 60	*11.8*	*15.8*	*3.0*	*12.1*	*16.7*	*3.2*	*16.2*	*20.3*	*4.6*	*15.6*	*19.2*	*4.4*
	Vanilla - 30	*11.1*	*15.5*	*2.8*	*12.5*	*17.3*	*3.3*	*16.1*	*20.2*	*4.5*	*15.1*	*18.9*	*4.2*
	Vanilla - 60	*11.2*	*15.6*	*2.8*	*12.6*	*17.3*	*3.3*	*16.3*	*20.2*	*4.6*	*15.9*	*19.4*	*4.5*

3.2 Bugs

During a short period of 2 weeks we have been running YinYang with fusion functions generated with our generator. Since it was still under development, we couldn't use all theories during the whole time frame (String theory has been added only later). Nevertheless, in this short period we were able to find 4 confirmed bugs in Z3. One of them was rejected by the maintainer, even though it is a segmention fault regression introduced in the last 3 months. The others have been accepted but still not fixed. One of them is a memory leak affecting string solver `z3str3`. The other two are soundness issues. We also found a lot of possible duplicates. We avoid posting possible instances of the same problem and wait for the developer to fix the first bug and only post a new one if the input is still triggering an error with the updated code. We also compare the behavior of the version under test against previous versions to identify regressions. We de-duplicate segmentation faults by comparing their ASAN traces. We have noticed that with bigger fusion functions it's easier to trigger crashes and memory faults than with shorter ones (around 10-20% more such inputs are generated). However, solve time also increases significantly in relation to the fusion function size, subsequently reducing the efficiency of YinYang.

Reduction. Before submitting bugs to the developers, we use delta debugging tools to reduce the input size. ddSMT [9] has shown to be particularly efficient in reducing most of the bugs. In addition, we use C-Reduce [13] mainly to reduce

Table 4. Code coverage evaluations (QF_S, QF_SLIA).

		QF_S						**QF_SLIA**					
		SAT			UNSAT			SAT			UNSAT		
		l	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*	*l*	*f*	*b*
CVC5	*FFG-10 - 30*	*21.1*	*33.4*	*6.8*	*13.7*	*25.8*	*3.8*	*22.2*	*34.4*	*7.3*	*21.1*	*33.5*	*6.9*
	FFG-10 - 60	*21.0*	*33.0*	*6.8*	*13.8*	*25.9*	*3.8*	*22.5*	*35.0*	*7.5*	*19.5*	*30.9*	*6.3*
	FFG-20 - 30	*20.9*	*33.3*	*6.7*	*13.9*	*26.0*	*3.9*	*21.7*	*34.1*	*7.2*	*17.7*	*29.4*	*5.5*
	FFG-20 - 60	*21.0*	*33.3*	*6.7*	*13.8*	*25.9*	*3.9*	*19.1*	*30.6*	*6.1*	*18.1*	*29.6*	*5.8*
	FFG-25 - 30	*20.9*	*33.3*	*6.7*	*13.8*	*26.0*	*3.9*	*22.5*	*34.8*	*7.5*	*17.8*	*29.4*	*5.6*
	FFG-25 - 60	*20.9*	*33.3*	*6.6*	*13.9*	*26.0*	*3.9*	*21.8*	*33.9*	*7.2*	*17.7*	*29.5*	*5.6*
	FFG-30 - 30	*20.9*	*33.2*	*6.7*	*13.8*	*25.9*	*3.8*	*18.8*	*30.2*	*5.9*	*17.7*	*29.5*	*5.5*
	FFG-30 - 60	*21.0*	*33.3*	*6.7*	*13.8*	*26.0*	*3.9*	*21.0*	*32.6*	*6.8*	*21.2*	*33.8*	*6.9*
	FFG-5 - 30	*21.1*	*33.5*	*6.8*	*13.7*	*25.9*	*3.8*	*23.4*	*35.4*	*7.8*	*21.8*	*34.0*	*7.2*
	FFG-5 - 60	*21.2*	*33.4*	*6.9*	*13.8*	*26.0*	*3.8*	*23.4*	*35.4*	*7.9*	*18.9*	*29.9*	*6.1*
	FFG-50 - 30	*20.7*	*32.9*	*6.6*	*13.7*	*25.9*	*3.8*	*0.6*	*2.0*	*0.1*	*17.0*	*28.9*	*5.2*
	FFG-50 - 60	*20.9*	*33.0*	*6.7*	*13.7*	*25.8*	*3.8*	*0.6*	*2.0*	*0.1*	*16.9*	*28.7*	*5.2*
	Vanilla - 30	*21.1*	*33.4*	*6.8*	*13.6*	*25.9*	*3.7*	*22.8*	*34.5*	*7.7*	*19.2*	*30.2*	*6.2*
	Vanilla - 60	*21.2*	*33.4*	*6.8*	*13.6*	*25.9*	*3.7*	*23.2*	*34.9*	*7.8*	*18.8*	*29.8*	*6.1*
Z3	*FFG-10 - 30*	*11.9*	*16.5*	*3.3*	*11.8*	*15.4*	*3.4*	*12.0*	*16.2*	*3.4*	*11.9*	*15.6*	*3.5*
	FFG-10 - 60	*12.5*	*16.7*	*3.6*	*11.9*	*15.6*	*3.5*	*11.7*	*16.1*	*3.3*	*12.1*	*15.8*	*3.6*
	FFG-20 - 30	*12.0*	*16.5*	*3.4*	*11.9*	*15.6*	*3.5*	*12.5*	*16.7*	*3.6*	*11.8*	*15.6*	*3.5*
	FFG-20 - 60	*12.1*	*16.6*	*3.5*	*11.9*	*15.6*	*3.5*	*12.7*	*16.9*	*3.7*	*11.6*	*15.5*	*3.4*
	FFG-25 - 30	*11.6*	*16.4*	*3.2*	*11.5*	*15.4*	*3.2*	*11.8*	*16.1*	*3.3*	*11.4*	*15.0*	*3.3*
	FFG-25 - 60	*12.2*	*16.6*	*3.5*	*11.8*	*15.5*	*3.4*	*12.5*	*16.7*	*3.6*	*12.6*	*16.2*	*3.8*
	FFG-30 - 30	*12.1*	*16.6*	*3.4*	*11.9*	*15.6*	*3.5*	*12.3*	*16.6*	*3.5*	*11.7*	*15.5*	*3.4*
	FFG-30 - 60	*12.5*	*16.8*	*3.6*	*12.1*	*15.8*	*3.6*	*11.7*	*16.1*	*3.3*	*11.4*	*15.0*	*3.3*
	FFG-5 - 30	*11.9*	*16.5*	*3.3*	*12.3*	*16.4*	*3.6*	*11.9*	*16.2*	*3.3*	*12.6*	*16.4*	*3.8*
	FFG-5 - 60	*11.5*	*16.4*	*3.2*	*12.0*	*15.7*	*3.5*	*12.5*	*17.1*	*3.5*	*12.2*	*15.9*	*3.6*
	FFG-50 - 30	*11.9*	*16.4*	*3.3*	*10.0*	*14.5*	*2.6*	*0.4*	*0.4*	*0.1*	*11.1*	*14.8*	*3.1*
	FFG-50 - 60	*11.9*	*16.4*	*3.4*	*9.2*	*13.9*	*2.3*	*0.4*	*0.4*	*0.1*	*11.3*	*14.9*	*3.3*
	Vanilla - 30	*12.4*	*16.7*	*3.6*	*10.9*	*15.3*	*3.0*	*12.6*	*16.7*	*3.6*	*12.7*	*16.6*	*3.8*
	Vanilla - 60	*12.5*	*16.8*	*3.6*	*11.1*	*15.3*	*3.0*	*11.9*	*16.1*	*3.4*	*12.2*	*16.1*	*3.6*

crashes. Since C-Reduce is not intended for SMT-LIB format, however, it may generate input that is not valid SMT-LIB or type inconsistent, even if it still triggers the problem. Even if not a real issue, we prefer to have report inputs that are well-formed, especially in the case of regressions, where we can then show that the bug is only triggered in recent versions of the solver. A technique we found particularly effective and also seems to deal well with this issue, is to run ddSMT and C-Reduce interleaved and then always finish by running ddSMT, which can understand and remove malformed parts from the input most of the time. A problem related to the usage of bigger formulas is that the mutated test input is bigger, thus requiring more time and effort for reduction before being able to report.

4 Related Work

Previous research from Mansur et al. (STORM) [10] has explored the possibility of generating new satisfiable formulas by recombining sub-expressions from an initial pool of seeds. While their approach is applicable to formulas containing multiple theories, individual sub-expressions can only be combined at the Boolean operator level. Comparing the generation procedure involved in our approach and theirs, we notice that we are starting from scratch, while their mutations rely on the initial pool of available expressions. Furthermore, our approach can also instantiate constants without giving them values, which is not always true when dealing with sub-expressions. The idea itself of generating new formulas by combining expressions is fundamental in their approach and also a key point of extensibility in ours, allowing Expression Rewriting.

Other methods such as Type-Aware Operator Mutation [14] replace operators by randomly chosen ones of the same (sub)type. This idea of compatibility of types can be found in our operator selection Algorithm 4. Generative Type-Aware Mutation [12] expands on Type-Aware Operator Mutation by replacing a sub-expression of type T with the application of a randomly chosen operator returning the same type T over sub-expressions of matching argument types. However, these techniques do not provide any satisfiability guarantees and therefore rely on differential testing. Despite this, operator mutation has proved to be highly effective at discovering critical bugs. As already seen above, the idea of using expressions in place of operators and vice versa is necessary to support non-invertible operators in our generator.

In the field of automatic generation of SMT formulas, an important contribution has been made by Blotsky et a. with StringFuzz [4], which is both a fuzzer and a generator for string constraints. The tool ships with 7 different built-in generators for string formulas. Those generators are configurable with many options. It also provides transformers to generate new instances by mutating seeds. Some of them ensure satisfiability-preservation. A key similarity of the approaches is the focus on extensibility as StringFuzz implements its components as Unix filters. However, the tool is specifically designed for string theory manipulation only, whereas our tool is also extensible at the theory level.

5 Conclusion

We have introduced an effective and easily extensible approach to automatically generate fusion functions to be used in Semantic Fusion for Satisfiability Modulo Theory (SMT) solver validation. The key idea is to treat the problem at an operator level by encoding rewriting rules for each operator. Our approach ensures the invertibility of the generated functions. Fundamentally, we make use of syntactic rewriting rules and Expression Rewriting, which enables the treatment of multiple nested STM solver level operators as atomic operators during rewriting, allowing for great flexibility and overcoming the limitation of non-invertible operators. Our tool produces configuration files that achieve equal or better coverage metric results when used to test state-of-the-art SMT solvers such as Z3 and CVC5 with YinYang. We also discovered and reported 4 confirmed bugs in Z3 in a time frame of only 2 weeks. Finally, we also modified YinYang to support a wider range of fusion functions.

In our future work, we want to extend the tool by providing new rewriting rules for already supported theories as well as introduce new theories such as Boolean Vectors and Arrays, which could allow to greatly enhance the coverage achieved by YinYang. It would also be interesting to further modify YinYang to accept more complex fusion functions, e.g. by considering more than two variables during Semantic Fusion. These types of functions are already obtainable from our tool. An exploration in this direction can possibly give more insights on Semantic Fusion itself and possibly achieve better bug finding results. Finally, it could be interesting to extend the tool to generate fusion functions with mixed variable types, which would allow to better test the interaction of multiple theories at the level of the SMT solver.

References

[1] 2022. *Semantic Fusion Seeds*. Retrieved May 24, 2022 from https://github.com/testsmt/semantic-fusion-seeds

[2] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: a versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.

[3] 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`.

[4] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A Fuzzer for String Solvers. In *CAV*.

[5] Nicola Dardanis and Lucas Weitzendorf. 2022. *fusion-function-generator*. Retrieved June 5, 2022 from https://github.com/nicdard/fusion-function-generator/

[6] GitHub. 2022. *Fusion functions specification*. Retrieved May 24, 2022 from https://yinyang.readthedocs.io/en/latest/fusion.html#fusion-functions

[7] GitHub. 2022. *YinYang*. Retrieved May 24, 2022 from https://github.com/testsmt/yinyang

[8] GNU. 2022. *Using the GNU Compiler Collection (GCC): Gcov*. Retrieved May 24, 2022 from https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[9] Gereon Kremer, Aina Niemetz, and Mathias Preiner. 2021. ddSMT 2.0: Better Delta Debugging for the SMT-LIBv2 Language and Friends. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 231–242. https://doi.org/10.1007/978-3-030-81688-9_11

[10] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. *CoRR* abs/2004.05934 (2020). arXiv:2004.05934 https://arxiv.org/abs/2004.05934

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

[12] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative Type-Aware Mutation for Testing SMT Solvers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 152 (oct 2021), 19 pages. https://doi.org/10.1145/3485529

[13] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. *SIGPLAN Not.* 47, 6 (jun 2012), 335–346. https://doi.org/10.1145/2345156.2254104

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