One Test to Rule Them All

Alex Groce
Josie Holmes
Kevin Kellar

ABSTRACT

Test reduction has long been seen as critical for automated testing. However, traditional test reduction simply reduces the *length* of a test, but does not attempt to reduce *semantic* complexity. This paper extends previous efforts with algorithms for *normalizing* and *generalizing* tests. Rewriting tests into a *normal form* can reduce semantic complexity and even remove steps from an already delta-debugged test. Moreover, normalization dramatically reduces the *number* of tests that a reader must examine, partially addressing the "fuzzer taming" problem of discovering distinct faults in a set of failing tests. Generalization, in contrast, takes a test and reports what aspects of the test could have been changed while preserving the property that the test fails. Normalization plus generalization aids understanding of tests, including tests for complex and widely used APIs such as the NumPy numeric computation library and the ArcPy GIS scripting package. Normalization frequently reduces the number of tests to be examined by *well over an order of magnitude*, and often to just one test per fault. Together, ideally, normalization and generalization allow a user to replace reading a large set of tests that vary in unimportant ways with reading *one annotated summary test.*

CCS CONCEPTS

•Software and its engineering →Software testing and debugging;

KEYWORDS

test case reduction, semantic simplification, fuzzer taming

ACM Reference format:
Alex Groce, Josie Holmes, and Kevin Kellar. 2017. One Test to Rule Them All. In *Proceedings of International Symposium on Software Testing and Analysis, Santa Barbara, California, USA, July 2017 (ISSTA'17),* 11 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

It has long been understood that effective automated testing often requires test reduction [24, 39, 40, 50] to produce useful tests. In fact, test reduction is now standard practice in industrial testing tools such as Mozilla's `jsfunfuzz` [45–47].[1] However, simply reducing the length of a test does not produce true semantic simplicity. There may be many 1-minimal[2] tests that present different variations of a single fault. Far too often, reading more than one of these tests provides no additional information on the fault.

Consider the three 1-minimal tests in Figure 1. These tests are very similar, and all result in an unbalanced AVL tree (due to a missing call to `rebalance` in `delete`). However, the tests are syntactically different, and a testing system that collects failing tests will present all three of these tests to a user. In this paper we propose to go beyond test reduction and convert all three of these tests into a single, *normalized* form that preserves failure while deemphasizing accidental aspects of each test, such as particular integer values and variable names, and ordering of steps.

Figure 2 shows the result of applying our *test normalization* algorithm to the three tests in Figure 1, and then applying our *test generalization* algorithm to the normalized test. *All three tests normalize to the same test.* Normalization is enabled by a term rewriting algorithm [14, 35] that operates on the level of test actions, and is thus language-agnostic: it works by successively rewriting tests into "simpler" versions that preserve failure and are likely to retain the underlying cause of the failure. Many different reduced tests therefore normalize to the same form. Unlike delta-debugging, normalization applies even to 1-minimal tests, and often provides further reduction beyond the level of 1-minimality.

The test also includes comments, produced by our *generalization* [43] algorithm, indicating what about the test can be changed while preserving the property that the test fails. Generalization uses automated experiments to discover a semantic neighborhood of failing tests. E.g., the value 1 assigned to `int0` in step 0 is not essential. It could be changed to any value in the range 5-20. Similarly, the exact ordering of many steps in the test is inessential. Finally, in step 9, instead of using the existing value of `int1` (4), a fresh assignment could be inserted before the `delete` call, setting `int1` to 3 instead. Changing any of these aspects of the test (one at a time) will preserve failure. Generalization shows the user how a failure represents a *family* of similar failing tests.

Combining normalization and generalization avoids common problems with understanding automatically generated tests. For instance, when a large integer appears in a test, the question arises — is this value important, or just a random number of no significance [26]? Large values in a normalized test are always essential, because normalization includes value minimization. Without generalization, however, it would be easy to assume all *small* numeric values in normalized tests are accidental. Generalization allows users to distinguish actually essential small values.

Normalization is not yet a complete solution to the problem of identifying distinct faults (e.g., our algorithms do not apply to complex custom test generators such as Csmith [49] or `jsfunfuzz` [46]), but it is often highly effective. Running 100,000 tests (of

[1]Approaches that optimize for short tests [9, 18] may not require reduction, but random testing [7, 32], model-checking [19], and symbolic execution [51] can all benefit.

[2]No single component of a 1-minimal/delta-debugged [50] test can be removed without causing the test to pass.

```
Test #1               Test #2               Test #3
avl0 = avl.AVLTree()  int0 = 14             avl1 = avl.AVLTree()
int0 = 4              avl0 = avl.AVLTree()  int3 = 18
int2 = 13             int2 = 13             avl1.insert(int3)
int3 = 7              int1 = 15             int0 = 5
avl0.insert(int2)     avl0.insert(int1)     int3 = 12
avl0.insert(int3)     int1 = 11             avl1.insert(int0)
int1 = 15             avl0.insert(int2)     int0 = 15
avl0.insert(int1)     avl0.insert(int0)     avl1.insert(int0)
avl0.insert(int0)     avl0.insert(int1)     avl1.insert(int3)
avl0.delete(int2)     avl0.delete(int0)     int1 = 15
                                            avl1.delete(int1)
```

Figure 1: Three random tests for one fault.

```
#[
int0 = 1                              # STEP 0
#  or int0 = 5
#   - int0 = 20
#  swaps with step 4
int1 = 3                              # STEP 1
#  or int1 = 5
#   - int1 = 20
#  swaps with step 6
avl0 = avl.AVLTree()                  # STEP 2
#] (steps in [] can be in any order)
avl0.insert(int0)                     # STEP 3
#[
int0 = 2                              # STEP 4
#  swaps with step 0
avl0.insert(int1)                     # STEP 5
#] (steps in [] can be in any order)
int1 = 4                              # STEP 6
#  or int1 = 5
#   - int1 = 20
#  swaps with step 1
avl0.insert(int1)                     # STEP 7
avl0.insert(int0)                     # STEP 8
avl0.delete(int1)                     # STEP 9
#  or (
#      int1 = 3  ;
#      avl0.delete(int1)
#     )
```

Figure 2: Normalization and generalization for all three tests. Lines beginning with # are comments in Python, used for annotations.

length 100) on the faulty AVL tree produces 860 failing tests with no duplicates. Normalizing these reduces the number of distinct failing tests to just 22. Ideally *all* failures due to the same fault in the SUT (Software Under Test) would normalize to a single, representative test. We aim to *approximate* such a canonical form for faults. Figure 5, in Section 2.1.2, shows an AVL tree test for this fault that normalizes differently. In experiments with 82 AVL tree faults, the mean number of distinct failures after normalization for 1,000 tests was just 3.1 (with median 2).

The contributions of this paper are 1) the idea of test normalization and generalization as key steps towards a goal of "one test to rule them all" (per fault), 2) algorithms for normalization and generalization that make use of the abstract interface for testing provided by the TSTL [27–29, 34] domain-specific language (DSL) [17], and 3) experimental results showing the value of these ideas. Normalization frequently provides significant additional test length reduction for complex SUTs, and can reduce the set of failures to be examined by more than an order of magnitude. Normalization and generalization have also been useful in understanding complicated tests for a variety of real-world software systems.

```
@import avl
pool: <int> 4 CONST
pool: <avl> 3
property: <avl>.check_balanced()
<int> := <[1..20]>
<avl> := avl.AVLTree()
<avl>.insert(<int>)
<avl>.delete(<int>)
<avl>.find(<int>)
<avl>.inorder()
```

Figure 3: Part of TSTL harness for AVL trees.

2 FORMAL DEFINITIONS

TSTL [27, 28, 34] is a language for defining the structure of tests (usually API-call sequences, but also grammar-based tests), and a set of tools for use in generating, manipulating, and understanding those tests. Figure 3 shows a simplified portion of a TSTL definition (called a harness [22, 25]) of tests for an AVL tree class, in the latest syntax for TSTL. Given a harness like the one in Figure 3, TSTL compiles it into a class file defining an interface for testing that provides features such as querying the set of available testing actions, restarting a test, replaying a test, collecting code coverage data, and so forth. The TSTL release [29] provides testing tools that use the interface for testing and debugging.

The key point for our purposes is merely that a TSTL test harness defines a set of *pools* that hold values produced and used during testing [4] (a common approach to defining API-testing sequences) and a *finite* set of actions that are possible during testing, typically API calls and assignments to pool values. In this example, there are two pools, one named `int` and one named `avl`. There are four instances of the `int` pool, which means that a test in progress can store up to 4 `int`s at one time (in variables named `int0`, `int1`, `int2`, and `int3`), and three instances of the `avl` pool. The actions defined are: setting the value of an `int` pool to any integer in the range 1-20 inclusive, setting the value of an `avl` pool to a newly constructed AVL tree, and calling `insert`, `delete`, `find` and `inorder` with chosen pools. Figure 1 in the introduction shows three valid tests produced by running a random test generator on the TSTL-compiled interface produced by this definition. TSTL handles ensuring that tests are well-formed. No pool instance (such as `avl1`) can appear in an action until it has been assigned a value. No pool instance that has been assigned a value can be assigned a different value until it has been used in an action, to avoid degenerate sequences such as `int3 = 10` followed by `int3 = 4`. Each action in a test is called a "step" — the first step of the first test in Figure 1 is storing a new AVL tree in `avl0`, for example. A test is just an ordered sequence of actions, which is equivalent to a set of numbered steps. Because our normalization is defined in terms of actions, steps, and pools, it is language agnostic.

The definition of pools and actions in TSTL defines a *total order* on all actions. First, actions are ordered by their position in the harness definition file. All `insert` actions therefore precede all `delete` actions, and all `delete` actions precede `find` actions. One line of TSTL typically defines more than one action. For example, the line `<avl>.insert(<int>)` defines 12 actions, one for each choice of `avl` and `int` pool instance, e.g. `avl0.insert(int0)`, `avl1.insert(int0)`, and `avl0.insert(int1)`. These are ordered lexically, with the first pool appearing in the text taking precedence.

Value ranges, such as in the `int` initialization, are also ordered in the natural way, with lower values first. Given this total order, each action can be assigned a unique index, from 0 up to 1 less than the total number of actions. Initially, this kind of ordering (and numbering) for each action was intended to allow for a kind of Gödel-numbering of tests, for use in proofs about properties of test-generation algorithms [4].

The ordering of actions also allows us to concisely define a practical method for normalizing tests, by simple syntactic means, by considering an action "simpler" than another action if it has a lower index. One intuitive concern with normalization based on action ordering is that a user may not place actions in a "good" order. What if normalization rewrites actions into "more complex" actions? The exact ordering usually *does not matter*, it only matters that there exists *some ordering* to guide normalization and generalization. Actions will only be replaced or swapped if, causally, the predicate (failure) is *indifferent* to the choice. The normal form's *existence* matters much more than the precise contents, in the context of that semantic restriction. In cases where there is a clear notion of simplicity (e.g., more vs. fewer optional parameters to a call, fewer array dimensions), simplest-first is often also most natural in TSTL.

2.1 Normalization

A test normalization algorithm has a simple goal: we ideally aim to produce a function $f : t \rightarrow t$ (a function that takes a test and returns a test) such that: (1) if t fails, $f(t)$ fails[3]; (2) if t_1 and t_2 fail due to the same fault, $f(t_1) = f(t_2)$; (3) if t_1 and t_2 fail due to different faults, $f(t_1) \neq f(t_2)$.

Such a function would define a true *canonical form* for tests, where each underlying fault is uniquely represented by a single test. In general, it seems clear that defining such a function f is (at least) as difficult as automatic fault localization and repair, and thus undecidable. Therefore, we aim at approximating the goal, by providing a set of simple transformations such that: (1) f changes many tests to the same test, (2) f has low probability of changing two tests failing for different reasons into the same test, and (3) f is not unreasonably expensive to compute. The implementation for f (in fact, for a family of f-approximating functions, with different tradeoffs in runtime and level of normalization) involves defining a set of rewrite rules such that for a test t, the rules define a finite set of candidate tests $C(t)$ where $t' \in C(t)$ if $t \Rightarrow t'$, possible simplifications of t, where each t' is the result of applying some rewrite rule to t. The notion of simplicity is defined by a restriction on the rewriting rules. For any rewrite $t \Rightarrow t'$, we require that $|C(t')| < |C(t)|$. Such a rewrite system is necessarily *strongly normalizing*: any sequence of rewrites chosen will eventually end in a term (test) that cannot be further rewritten, since the total length of a rewrite sequence is bounded by the initial $|C(t)|$ [14].

In the setting of TSTL, where test actions have a defined total order, two simple principles can be applied to produce useful rewrite rules. All rewrites reduce the sum of the indices of the actions in the test, make the test's actions more ordered by index, or reduce test length. This guarantees that the rewrites are strongly normalizing. The second principle that determines the rewrite rules is that each rule ought to be unlikely to change the underlying cause for test failure. To that end, the rules for normalization always either change at most one action (possibly in multiple steps, but in a uniform way) or make *no* changes to the actions performed, only to the pools used or the positions of actions in the test. We cannot guarantee normalization does not change the underlying fault in a test; however, the limited scope of rewrites should minimize the chance of fault change (known as "slippage" [8, 33]).

[3]Normalization can be generalized to apply to any predicate over tests, not just failure [21]; a passing test can be normalized, though without some more interesting predicate, such as preservation of coverage, this is not clearly useful.

2.1.1 *Rewrite Rules.*

Figure 4 shows the rewrite rules used in TSTL normalization. The notation in the rules is relatively simple. A *step* is an action paired with an index indicating its position in a test, where the first action is step 0, etc.; e.g., $(2 : a)$ indicates the third step of the test is action a (indexing is from 0). $\Delta(t, t')$ is the set of all steps in t such that $t(i) \neq t'(i)$.

For ordering, we say that $a < b$ iff the index of action a is lower than that of action b. We compare steps with $<$ by comparing their actions — $(i : a) < (j : b)$ iff $a < b$. For a set or sequence of actions or steps, we define the *min* of the set to be the lowest indexed action in the set, and use these to compare sets: $s_1 < s_2$ iff $min(s_1) < min(s_2)$. For pools, $p < p'$ if and only if p's index is lower than the index for p' *and* p and p' are from the same pool.

The term $t[x \mapsto y]$ denotes the test t with all instances of x replaced by y. Here, x and y can be actions, steps, or pools. $t[x \leftrightarrow y]$ is similar, except that x and y are swapped. Term $t(i, j)[x \mapsto y]$ is the same as $t[x \mapsto y]$, except that the replacement is only applied between steps i and j, inclusive. Finally, $t_{\rightarrow i}(x)$ denotes t with all steps containing x that are before step i moved to step i, preserving their previous order, shifting steps at i and after i to make room for the moved steps, again preserving order.

2.1.2 *Normalization Algorithm.*

These rules alone do not determine a complete normalization method; it is also necessary to determine the order in which they are applied. The order in our default implementation is the order above, with the modification that in practice the **ReplacePool** and **ReplaceMovePool** rewrites are checked in the same loop (e.g., for every possible replacement of a pool, both rules are checked, in the order given above). The core algorithm, assuming a set of ordered rewrite rules defines $C(t)$, is given as Algorithm 1. Here `pred` is an arbitrary predicate indicating that the candidate test still satisfies the property of interest that held for the original test t. In most cases, this predicate will be "the test fails" but we also have preserved code coverage for regression suites [21]. Notice that after applying each rewrite rule, we perform delta-debugging on the new base test, since often a rewrite makes other steps irrelevant.

Figure 5 shows the sequence of rewrites to normalize an AVL tree failure. The numbering of steps appears inconsistent in the first **SwapAction** because a successful delta-debugging removes a no-longer-needed step after a rewrite. The pattern in this example, where successful rewrites are roughly equal to the original number of steps, is frequently seen across SUTs.

The worst-case complexity of normalization can be given an upper bound by recalling our rule that each rewrite must lower the number of possible rewrites by at least one. This means if there are n possible rewrites of a test, there can be at most n predicate checks for the current test, then at most $n - 1$ checks for the rewritten test, and so on, for a total of $\frac{n(n+1)}{2}$ predicate checks, where n

- **SimplifyAll:** $t \Rightarrow t[a \mapsto a']$
  where $a' < a$
  Covers the case where all appearances of an action can be replaced with a lower-indexed action.
- **ReplacePool:** $t \Rightarrow t(i,j)[p \mapsto p']$
  where $p < p'$ and $0 \leq i < j < |t|$
  Covers the case when all appearances of an instance of a pool can be replaced with a lower-indexed instance of that pool (possibly only within a range of steps).
- **ReplaceMovePool:** $t \Rightarrow t_{\rightarrow i}(p')[p \mapsto p']$
  where $p < p'$ and $0 \leq i < |t|$
  Covers the case when all appearances of an instance of a pool can be replaced with a lower-indexed instance of that pool, if all steps containing the new instance before a certain step are then moved to that step.
- **SimplifySingle:** $t \Rightarrow t[(i : a) \mapsto (i : a')]$
  where $a' < a$
  Covers the case where one action can be replaced with a simpler (lower-indexed) action.
- **SwapPool:** $t \Rightarrow t(i,j)[p \Leftrightarrow p']$
  where $\Delta(t', t) < \Delta(t, t')$
  and $0 \leq i < j < |t|$
  and $p < p'$
  Covers the case where swapping two pool instances (within a range of steps) reduces the minimum action index of the steps.
- **SwapAction:** $t \Rightarrow t[(i : a) \mapsto (i : b), (j : b) \mapsto (j : a)]$
  where $i < j$ and $b < a$
  Covers the case where two actions can be swapped in the test, with the lower-indexed action appearing first.
- **ReduceAction:** $t \Rightarrow t[(i : a) \mapsto (i : a')]$
  where $|ddmin(t')| < |t|$
  Covers the case where an action can be replaced by any action, enabling further delta-debugging.

Figure 4: Rewrite rules for normalization.

Algorithm 1 Basic algorithm for normalization

```
modified = True
while modified do
    modified = False
    for $t' \in C(t)$ do
        if pred($t'$) then
            modified = True
            $t = ddmin(t')$
            break (exit for loop)
        end if
    end for
end while
return $t$
```

is the number of possible rewrites of the test t being normalized: $n = |C(t)|$. Test execution to check the predicate can be assumed to have a constant cost since the length of the test does not usually change by more than a few steps during normalization.

Original test:

```
 0: int0 = 10
 1: int2 = 7
 2: avl1 = avl.AVLTree()
 3: avl1.insert(int2)
 4: avl1.insert(int0)
 5: int1 = 1
 6: int3 = 1
 7: avl1.insert(int3)
 8: int3 = 15
 9: avl1.insert(int3)
10: avl1.delete(int1)
```

Normalized:

```
0: int0 = 1
1: int1 = 2
2: avl0 = avl.AVLTree()
3: avl0.insert(int0)
4: avl0.insert(int1)
5: int1 = 3
6: avl0.insert(int1)
7: int1 = 4
8: avl0.insert(int1)
9: avl0.delete(int0)
```

Normalization Steps:

SimplifyAll: `int0 = 10` ↦ `int0 = 2`
SimplifyAll: `int2 = 7` ↦ `int2 = 3`
SimplifyAll: `int3 = 15` ↦ `int3 = 4`
ReplacePool: `int2` ↦ `int1`
ReplacePool: `avl1` ↦ `avl0`
ReplacePool: `int3` ↦ `int0`
SwapAction: `(0: int0 = 2)` ↔ `(6: int0 = 1)`
SwapPool: `int0` ↔ `int1 (between steps 2 and 10)`
SwapAction: `(1: int1 = 3)` ↔ `(5: int1 = 2)`

Figure 5: An example of normalization steps.

How many rewrites can a test t have? Assume there are k steps in the test and α possible actions. The action replacement rewrites (**SimplifyAll**, **SimplifySingle**, and **ReduceAction**) allow for at most $k(\alpha - 1)$ rewrites each. There are $\leq k^2$ possible **SwapAction** rewrites, and $\leq k^2(\alpha - 1)$ possible rewrites for each of **ReplacePool**, **ReplaceMovePool**, and **SwapPool**[4]. There are therefore at most $n = k^2 + 3k^2(\alpha - 1) + 3k(\alpha - 1)$ rewrites, over-approximating (since an action cannot be rewritten to itself). Each rewrite also requires a *ddmin* call, which is quadratic in k [50]. Substituting this expression into the expression $\frac{n(n+1)}{2}$ above, we see that the worst-case cost for normalization is $O(k^4\alpha^2)$. While this is worse than the quadratic cost of delta-debugging, this algorithm is applied to already 1-minimal tests, unlike delta-debugging. The k in our quartic complexity is therefore, for random tests, typically an order of magnitude smaller than the k in delta-debugging [21, 24, 39], which can partly balance the additional cost for typical tests whose unreduced length is much larger than reduced length.

We believe that if normalization can decrease human effort in examining large numbers of redundant tests, this price is more than reasonable. In practice, most actions are not enabled at most steps, and the rules are applied in an order that quickly converges on a normal form for many tests. In our experiments, normalization was only very expensive when delta-debugging was also costly, and appeared to be worse than delta-debugging by a constant factor, somewhere in the range of 2-100x, usually close to 10x. With multiple faults, normalization provides a quantifiable value in shorter time until all faults have been examined [8].

The simplest optimization is to improve on the constant ordering of rewrite rules. Once a rule fails to produce a candidate that satisfies `pred`, that rule should be moved to the end of the ordering of rewrites, since once a rule fails once to produce any valid rewrites,

[4]The details of how these bounds are determined, in that pool changes are also action changes, are not critical, and a more detailed analysis is somewhat involved, and beyond the scope of this paper; we note that like delta-debugging, the worst-case complexity is seldom observed, and offer some further optimizations.

it frequently produces no further reductions. This simple change typically halves the time required for normalization. When test execution is very expensive, the set of candidate tests can be further restricted: limiting action replacements to cases where Levenshtein [41] distance (text edit distance) between the code for actions is bounded to a small value was effective in reducing runtime, and often had little impact on final results. A further useful optimization when normalizing large numbers of tests is to cache results across tests (since the algorithm is deterministic for a given `pred`). For very large numbers of tests, this is the most important optimization. For systems with expensive replay, delta-debugging of each new base test can be omitted: the **ReduceAction** rewrite will eventually remove extraneous steps. It is also trivial to parallelize normalization by checking the predicate over multiple candidates at once. As soon as a candidate satisfies the predicate, a parallel implementation can proceed with that candidate as the new base, making the algorithm nondeterministic (in practice, we suspect the same final result will usually appear), or the algorithm can wait for all earlier-in-sequence candidates to be checked, and only proceed when no candidate that would be checked earlier is in the queue.

2.2 Generalization

The core idea of generalization is to use methods similar to those involved in normalization to provide a user with information about changeable aspects of a test. Some values and orderings of steps in a test are *essential* to the failure: when changed, they cause the test to no longer fail. Many others, however, are *accidental* — any concrete test has to choose *some* values and step ordering (enforced by the normalization process) but many such choices are arbitrary, or at least allow variance, with respect to the cause of failure. Generalization automatically performs experiments to distinguish essential and accidental aspects of a test, and summarizes the results. We can define a finite set of simple variations on a test, and summarize which variations maintain a predicate of interest. In the typical case of failure, generalization simply presents the user with *all* "very similar" tests that also fail. Formally speaking, a generalization algorithm produces as output *a set of rewrites* of a test t satisfying `pred` such that each rewrite also satisfies `pred`. Presenting a generalization involves concisely expressing these (small) rewrites.

2.2.1 Generalization Algorithm. The core algorithm (Algorithm 2) is simple, using only swaps and single-action rewrites:

Algorithm 2 Basic algorithm for generalization

```
swap = $\emptyset$
replace = $\emptyset$
for $(i, a) \in t$ do
    for $a' : a' > a$ do:
        if pred($t[(i, a) \mapsto (i, a')]$) then
            replace = replace $\cup((i, a), (i, a'))$
        end if
    end for
    for $j : i < j < |t| - 1 \wedge (j : b) > (i : a)$ do
        if pred($t[(i : a) \mapsto (i : b), (j : b) \mapsto (j : a)]$) then
            swap = swap $\cup((i, a), (j, b))$
        end if
    end for
end for
return (swap, replace)
```

This algorithm collects all steps that can be replaced with other actions or swapped with other steps, and returns the set to be reported to the user. This version assumes the test has already been normalized, but can be extended to any test by removing the restrictions that $a > a'$ and $(j : b) > (i : a)$. The complexity of generalization is simpler to determine than that of normalization. If we assume all actions are enabled at each step, and there are α actions and k steps, checking for replacements requires $k(\alpha - 1)$ test executions, when every action is the lowest-indexed action. In that worst case, no swaps are possible. The complexity of checking for swaps in the worst case is quadratic in k. In practice, most actions are not enabled at most steps, and most actions in a test are not the lowest-indexed action. Basic generalization is trivial to parallelize.

```
#[
test0 = []                             # STEP 0
#  or test0 = sut0.test()
actionlist0 = sut0.actions()           # STEP 1
#  or actionlist0 = sut0.enabled()
#] (steps in [] can be in any order)
action0 = actionlist0[0]               # STEP 2
#[
test0.append(action0)                  # STEP 3
pred0 = sut0.fails                     # STEP 4
#] (steps in [] can be in any order)
sut0.normalize(test0,pred0)            # STEP 5
sut0.normalize(test0,pred0)            # STEP 6
#  or (
#      test0 = []  ;
#      sut0.normalize(test0,pred0)
#     )
```

Figure 6: Generalized test for TSTL itself, showing fresh value generalization.

2.2.2 Fresh Values and Misleading Tests. A side-effect of delta-debugging and normalization is reduction of the number of variables in a test. While usually helpful, this can sometimes result in misleading tests. In a stateful system, putting the system into a bad state may require building a complex object. Once system state is corrupted, however, the complex object is irrelevant, and its appearance in the call leading to failure can be misleading. In previous work at NASA, we observed that sometimes a delta-debugged file system test [23, 24] would use an open file descriptor in a call, leading to the suspicion that the file had been corrupted, when in fact the file system's state was damaged, and the same operation on *any* file would have failed. We therefore propose a more aggressive generalization: replacing a pool use with a *fresh value*.

Consider the test in Figure 6, produced by a TSTL harness for TSTL itself[5]. The problem involves an invalid cache, produced by normalizing a test with only one action. Without fresh value generalization, it appears that the failure is due to normalizing `test0` again. The annotation after step 6 lets us see that the failure will take place even for a fresh test. Without this generalization, the state of `test` may appear to be important, not the system state.

Formalizing this generalization requires additional notation. $U(a)$ is the set of pools *used* in the action a — pools that in the action, but

[5]Since Python TSTL provides a Python API, that API can be used as the SUT in testing.

not on the left-hand side of an assignment. $I(a, p)$ is a predicate that is true iff action a stores a new value in pool p. Finally, $t[+(a : i)]$ denotes test t with the action a inserted at step i and each step from i onwards moved to a position one higher.

Algorithm 3 Basic algorithm for fresh object generalization

```
1:  fresh = $\emptyset$
2:  for $(i, a) \in t$ do
3:      for $p \in U(a)$ do
4:          for $a' : I(a', p)$ do
5:              if pred($t[+(a', i)]$) then
6:                  fresh = fresh $\cup (i, a')$
7:              end if
8:          end for
9:      end for
10: end for
11: return fresh
```

In practice, the `fresh` set returned should be pruned to avoid redundant actions. It is not useful information that `int0 = 1 ; int1 = 2; int0 = 1; f(int0)` fails if `int0 = 1; int1 = 2; f(int0)` fails. Redundancy elimination also needs to take into account the potential assignments to a pool from the `replace` generalization, which are also redundant. Furthermore, it is useful to distinguish between pools that are never modified, only assigned to, and pools that are modified without appearing on a left-hand side (LHS). As an example, if an integer is used as an argument to a function, the pool value's last assignment is still valid and should be omitted from "fresh" values, as redundant. However, calling a function on an AVL tree may modify it, making an assignment non-redundant, even if it is the last appearance of that pool on the LHS. We use the `CONST` tag (see Figure 3) to mark values than cannot be modified on the RHS. Further extensions of the fresh value generalization could be considered. For example, if a fresh value for some object requires use of a complex constructor, values required to call the constructor can also be produced, if needed, recursively. In our experiments so far, simple fresh value generation sufficed, as inputs to constructors were usually available in pools. Knowing all possible fresh values is likely unimportant.

3 EXPERIMENTAL RESULTS

This section presents some initial results of applying normalization and generalization. All tests were generated using pure random testing, based on TSTL harnesses developed previously, all included in the TSTL release [29]. We also tested the Python interface to Z3 [13], but did not find faults thus far; normalization did help produce more comprehensible and uniform Z3 quick tests [21].

These experiments are not intended to provide a full evaluation of normalization and generalization, but to establish the basic potential value of the techniques, and some initial data on core research questions, over seven Python libraries ranging from small to large and complex. These are: **RQ1:** How effectively does normalization reduce the number of failures reported? **RQ2:** How often does normalization lose faults? **RQ3:** What is the cost of normalization and generalization? **RQ4:** How much additional reduction over delta-debugging can normalization provide? and **RQ5:** Does

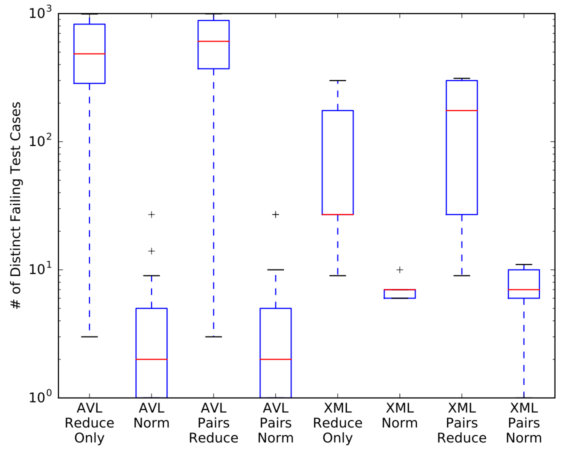

Figure 7: Effects of normalization on AVLTree and XML parser mutants.

normalization and generalization provide substantial benefits in understanding complex tests? The primary threat to validity is that we have only applied our methods to tests produced using random testing for seven subjects, written in Python (some small, some large). Note that **RQ5** would need a human study to fully evaluate.

3.1 AVLTree

For basic experiments on **RQ1-3**, we used a simple Python AVL tree found on the web [48], with 225 lines of code[6].

Figure 7 shows the reduction in number of distinct failing tests produced by normalization and reduction (vs. reduction only) for a set of 82 mutants [5] of AVLTree [36] (**RQ1**). Of the 228 mutants produced by MutPy [31], only these 82 produced at least 1 failure in 1,000 tests (other mutants were equivalent or caused failure at a lower rate). Using reduction only, the mean number of distinct failures for each mutant (obviously a single fault) was 498.4 (median 485). Using normalization, the mean was 3.1 failures (median 2). For 38 of the 82 mutants, normalization produced only a single failure. Differences were statistically significant by paired Wilcoxon test with $p \leq 1 \times 10^{-157}$.

AVLTree mutants also provided a way to evaluate the danger of normalization losing faults, **RQ2**. Faults can be lost when normalization changes a test failing due to one fault into a test failing due to a different fault, a problem known as "slippage" [8, 33]. AVLTree provides a slippage challenge, as there are very few API calls, and all calls take the same inputs, so tests for faults are likely to be very close to each other in the combinatorial space. To test the slippage rates for normalization, we randomly selected 364 mutant pairs drawn from the 82 detectable mutants, and produced higher-order-mutants for each of these by applying both mutants to the source (using only mutants that modified different source lines). Of these, the set of reduced tests included at least one test capable of exposing each of the two faults for only 238 pairs. In almost

[6]All sizes non-comment, non-blank lines, by cloc [12].
[7]All p-values given below are for the same recommended [6] statistical test.

all cases this was due to reduction slippage, but in a few cases it was due to one fault completely masking another: e.g., if `insert` always fails, it is not possible to produce a test that exposes a fault in `delete` on a non-empty tree.

Out of these 238 mutant pairs, normalization produced tests exposing both faults for 80.7% of pairs (19.3% slippage at the suite level). Interestingly, in 4 cases normalization took a set of reduced tests not capable of exposing both faults, and produced a smaller set of tests that was capable of detecting both faults. Slippage due to reduction is very rare for some SUTs but common for other SUTs (up to 23% for Mozilla's JavaScript engine) [8]. For the AVLTree example, the slippage rate for reduction is almost 30%, 10% *worse* than that for normalization. Mitigation approaches for slippage during reduction [33] should also apply to normalization.

The reduction in tests produced by normalization for mutant pairs was, as with single mutants, very large (Figure 7). For mutant pairs, the mean/median number of distinct failures was 554.4/607 for reduction alone, but only 2.8/2 with normalization. The runtime for normalization was almost unchanged from the single-mutant times. For reduction alone, using pairs increased the number of distinct failures, while normalized failure counts decreased. For 48.3% (115 of 238) of mutant pairs where reduction did not lose a fault, normalization worked as well as possible — it preserved both faults and produced only 1 or 2 test cases[8]. The improvement due to normalization was statistically significant with $p \leq 1 \times 10^{-80}$. A second way to examine reduction in a multi-fault setting is to consider the mean number of tests a user must examine before seeing all faults [8]. For the AVL mutant pairs, a user must examine almost 20 tests (on average) before encountering at least one instance of both faults. With normalization, this drops to a mean of just 2 tests. The difference is significant with $p \leq 1.4 \times 10^{-6}$.

AVLTree also provided basic results for **RQ3**. The mean cost to reduce a test was 0.05 seconds, with a median of 0.03 seconds. The mean cost for normalization was 0.38 seconds, with a median of 0.1 seconds. The minimum runtime for both algorithms was negligible (less than 1 millisecond), but the maximums were 1.3 seconds for reduction and 17.6 seconds for normalization. Note that in all our results the cost of normalization is given on an already-reduced test, so the inputs for normalization are smaller than those for reduction; however, this is the expected use-case for normalization. Comparing on equal-sized tests would simply involve adding the costs for reduction to those for normalization, as an additional step of normalization. Finally, the criticality of caching for normalizing large numbers of tests is evident. Out of 60,226 normalizations performed in our full AVLTree mutant experiments, 59,972 (99.6%) resulted in a cache hit (most of these after a small number of normalization steps). In fact, the total number of rewrites performed during the experiments was only 145,780, for a mean of only 2.4 non-cache-hit rewrites of each test.

3.2 XML Parser

We also investigated **RQ1-3** for a Python XML parser with about 260 lines of code [15], using one real fault, triggered by the empty tag (<>), and one seeded fault triggered when adding two nodes with the same name. A comment in the code indicates the seeded fault is realistic, and possibly existed in an earlier version of the code. Running 1,000 tests produced 848 failing tests. Without normalization, it took only 37.45 seconds to execute and delta-debug all 1,000 tests. The output was 717 distinct failing test cases. Normalization increased the runtime to 354.7 seconds, but output only 5 failures (3 for the original fault and 2 for the seeded fault). The XML parser also shows that normalization and generalization work for programs with string inputs defined by a grammar in TSTL, as well as for pure-API testing.

In addition to these realistic faults, we performed the same mutation-based analysis as with AVLTree (Figure 7). In this case, a weaker specification (no reference implementation) resulted in fault detection for only 5 of 357 mutants. For these mutants, reduction alone produced a mean of 107.6 failures (median 27); normalization reduced that to only 6.2 failures (median 7) (**RQ1**). The difference was significant at the 95% confidence level, ($p = 0.042$). With mutant pairs, these numbers increased to 181.9 mean faults (median 175) with reduction only, compared to 7.7 mean faults (median 7) when using normalization (**RQ1**). This difference was statistically significant, $p = 0.007$. Normalization was perfect in only one case for the XML parser mutants, and perfect for no mutant pairs. The difference in number of tests before encountering both faults was not statistically significant (for these 5 faults, failure rates are very similar, so hitting both is trivial). For combined mutants, the slippage rate was only 12.5% for the XML parser (**RQ2**). That is, of the 9 viable mutant pairs, only 1 lost a fault, and in that case the faults were both semantically and syntactically very similar (within 1 line and with similar effects). Reduction alone caused no slippage. Adding normalization to reduction increased the runtime from a mean of 12.6 seconds to 120.8 seconds (**RQ3**); there were 3,829 cache hits over a total of 3,929 normalizations.

3.3 TSTL

As noted in Section 2.2.2, TSTL is used to test TSTL's own API interface (the code is about 2,700 LOC; a compiled SUT is often > 30KLOC). We discovered one fault while testing the latest version of TSTL, the cache-related problem shown in Figure 6[9]. Generating and reducing 100 tests for it required 1,090 seconds and produced 90 failures. Normalization and generalization increased total runtime to 3,690 seconds, and produced just 2 failures.

3.4 SymPy

SymPy [2] is a widely used open source pure Python library for symbolic mathematics. SymPy is used by several other projects, has over 400 contributors, over 25,000 commits to date, and over 225KLOC. The TSTL tester for SymPy focuses on core expressions and algebraic simplification, and covers about 15KLOC and 21,000 branches of the system. Testing this core resulted in discovery of a number of faults in SymPy, detected by assertion violations or uncaught (and not expected) exceptions. Some of these have been reported to the project; however, since SymPy currently has 2,128 open issues, with one opened approximately each day, only one has (at this point) been fixed. If we assume that each different assertion violation or exception message indicates a different underlying fault,

[8]It is sometimes possible to detect both faults with a single test case.

[9]We note that normalizing a test with respect to the predicate that it does not normalize (by a different predicate) may produce a headache in the TSTL user.

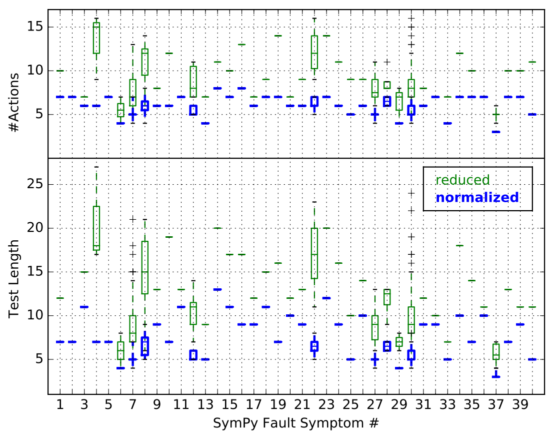

Figure 8: Effects of normalization on SymPy tests.

SymPy provides us with a set of 40 complex, hard-to-understand real faults for evaluating normalization. While we expect that this is an over-approximation of the actual number of distinct faults, inspection of the tests and the covered code suggests it is not far from the actual number. Because we used a (we believe) non-lossy fault identification method (the exact failure symptom), our SymPy results are relatively useless for **RQ2**, but they answer **RQ1**, **RQ3**, and **RQ4** for a large, realistic system and real faults.

We generated, normalized, and reduced tests until we had 500 tests, exhibiting all 40 fault signatures. Some SymPy faults (not included in our count of 40 tests) cause infinite loops, stopping reduction or normalization. Of 570 failures, 549 reduced and 500 both reduced and normalized.

RQ1: Reduction alone did not reduce the number of distinct failing tests at all. Normalization reduced the total number of distinct failing tests to 114. There were 12.5 mean different failing tests, per fault, for both unreduced and reduced tests, and only 3.15 mean failing tests per fault for normalized tests. This difference was significant, with $p = 0.003$. Normalization reduced the number of tests to examine for 11 of the 13 faults with more than one failure; in 2 cases, the normalization was perfect (one failure).

RQ3: The mean time for reduction was 104.45 seconds, with a median of 19.70 seconds. The mean time for normalization was 594 additional seconds, with a median of 260.214 seconds. The difference was significant, with $p \leq 1 \times 10^{-80}$.

RQ4: The mean length of unreduced tests was 44.664 steps, with a median of 40.5 steps. For reduced tests, this shrank to a mean of 9.984 and a median of 9.0 steps. Normalized tests had mean length of 5.48 steps and median of 5.0 steps. SymPy failures show that normalization reduces not only the length of tests, but the number of actions (roughly speaking, different API/method calls/functions) that must be considered for debugging: reduced tests included 8.116 mean different actions, but normalized tests only 5.282 mean different actions. These differences were all statistically significant with $p \leq 1 \times 10^{-76}$. Normalization made it possible to completely ignore a large number of SymPy functions for debugging purposes. The unreduced and reduced tests included all 50 SymPy functions tested. The normalized tests, however, included only 32 of these, and enabled us to ignore such complex code as trigonometric expansion and simplification, power expansion, logarithmic combination, and even generalized expansion. Figure 8 graphically shows the impact of normalization on length and number of functions covered by reduced tests. The lower part of the figure shows changes in test length, and the upper part shows, for the same fault, the change in number of different actions. The green boxplots show reduced tests, while the emphasized blue boxplots show normalized tests. It is clear that normalization not only has a significant effect on average, but has a large benefit for most individual faults. The reduction in tests to examine is also shown, indirectly, by the fact that the "boxes" for normalized data are often simply lines because the tests are similar or identical.

3.5 SortedContainers

SortedContainers [37] is a popular Python library of about 2KLOC, that provides pure Python sorted containers that are as fast as C extension containers. We have reported 3 bugs in SortedContainers (all quickly corrected). One of these bugs causes an infinite loop, making it difficult to reduce or normalize. We therefore only present results for the other two faults reported. We generated 168 failing tests, all distinct, exhibiting both reported faults, over 15 hours of testing. All failures reduced and normalized. **RQ1-RQ3:** Reduction did not reduce the number of failures, but did reduce mean test length from 80.8 to 13.2 steps (median from 85.0 to 12.5), and mean number of different actions per test from 14.4 to 7.4 (median from 14.0 to 8.0). Normalization was perfect: all tests normalized to two canonical tests, one per fault, both with only 6 steps and 5 distinct actions — a $> 50\%$ reduction in size beyond delta-debugging. Changes were significant with $p \leq 1.0 \times 10^{-24}$. There were 95 total different actions in tests before reduction, 27 in tests after reduction, and only 7 between the two normalized tests. **RQ4:** Reduction took a mean of 0.09 seconds, normalization a mean of 134.1 seconds (median 0.06 vs. 57.0) (significant, $p \leq 1 \times 10^{-28}$).

3.6 NumPy

Our final two case studies provide little information on **RQ1** and **RQ2**; for these SUTs, failure rates are low enough or test reduction runtimes high enough that each failure is usually dealt with one-by-one. However, the value of normalization and generalization for further reduction (**RQ4**) and aiding in understanding tests (**RQ5**) is effectively shown by these complex programs. They also provide results for **RQ3** when even test reduction is expensive. NumPy [1] is a widely used Python library that supports large, multi-dimensional matrices and provides a huge library of mathematical functions. The SciPy library for scientific computing builds on NumPy. Developing tests for NumPy is challenging, because none of the authors are experts in numeric computation, and the specification of correct behavior is often somewhat subtle. As an example, consider the test in Figure 9. Prior to normalization, understanding why the test leads to a violation of self-equality for an array is difficult: the reduced-only test has 42 steps and includes not only array multiplication and addition, but subtraction, array copying, reshaping, flattening, filtering by unique elements, and raveling. After normalization, it is much clearer what is happening:

```
dim0 = 1                            # STEP 0
#  or dim0 = 10
shape0 = (dim0)                     # STEP 1
#  or shape0 = (dim0, dim0)
#  or shape0 = (dim0, dim0, dim0)
array0 = np.ones(shape0)            # STEP 2
array0 = array0 + array0            # STEP 3
array0 = array0 + array0            # STEP 4
#  or array0 = array0 * array0
array0 = array0 * array0            # STEP 5
array0 = array0 * array0            # STEP 6
array0 = array0 * array0            # STEP 7
array0 = array0 * array0            # STEP 8
array0 = array0 * array0            # STEP 9
array0 = array0 * array0            # STEP 10
array0 = array0 * array0            # STEP 11
array0 = array0 * array0            # STEP 12
array0 = array0 * array0            # STEP 13
array0 = array0 - array0            # STEP 14
assert (np.array_equal(array0,array0))
```

Figure 9: Normalized and generalized NumPy test.

1) `array0` contains `NaN` and 2) this is correct behavior (the array *should* contain `NaN`). The greater length and much larger number of operations involved in the original reduced test obscures this critical point. In NumPy, array equality does not hold for objects containing `NaN`, so the assertion must be modified. As far as we know, normalization transforms all instances of this fault into this canonical test, but our data is insufficient to make a definite claim.

Other, more complex, failures have also made it clear that normalization is useful for additional test length reduction for NumPy, and that generalization makes any surprising restrictions on test values clear. For NumPy tests, normalization takes much longer than reduction, in part due to the expense of operations on large arrays. For almost all tests, the mean time to reduce tests is about 4-5 seconds, and the time for normalization is between 712 and 774 seconds. Generalization takes between 52 and 59 seconds in these cases. The exception was a test of 45,206 steps (!) leading to a memory exhaustion error and crash. This was reduced (over nearly a day) to a test with 10 steps, which then normalized (in only 2 hours) to a test with 8 steps. The normalized test involved no operations other than array initialization, array flattening, and array addition. The reduced test involved larger array dimensions, array multiplication, and array subtraction, as well.

3.7 Esri ArcPy

Esri is the single largest Geographic Information System (GIS) software vendor. Esri's ArcGIS tools are widely used for GIS analysis. Automation is essential for complex GIS analysis and data management, and Esri has long provided tools for programming GIS software systems. One such tool is a Python site-package, ArcPy [3]. ArcPy is a complex library, with dozens of classes and hundreds of functions. Most of the code involved in ArcPy functionality is the C++ source for ArcGIS itself (which is not available), but the released Python interface code alone is over 50KLOC. We have discovered and reported six crash-inducing faults in ArcPy/ArcGIS.

Figures 10 and 11 show one crash-inducing test, after initial delta-debugging (from over 2,000 test steps) (Figure 10) and after normalization and generalization (Figure 11). In this setting normalization has contributed a significant amount of additional reduction over delta-debugging. For the crash fault shown in this paper, normalization reduced the length from 19 steps to 11 steps. For three

```
shapefile2 = "C:\arctmp\new3.shp"                   # STEP 0
shapefile1 = "C:\arctmp\new3.shp"                   # STEP 1
featureclass2 = shapefile2                          # STEP 2
featureclass0 = shapefile1                          # STEP 3
shapefilelist2 =
   glob.glob("C:\Arctmp\*.shp")                     # STEP 4
fieldname0 = "newf3"                                # STEP 5
shapefile1 = shapefilelist2 [0]                     # STEP 6
featureclass1 = shapefile1                          # STEP 7
arcpy.CopyFeatures_management
   (featureclass1,featureclass2)                    # STEP 8
op1 = ">"                                           # STEP 9
newlayer2 = "l2"                                    # STEP 10
val1 = "100"                                        # STEP 11
selectiontype2 = "SWITCH_SELECTION"                 # STEP 12
fieldname1 = "newf1"                                # STEP 13
arcpy.MakeFeatureLayer_management
   (featureclass0, newlayer2)                       # STEP 14
arcpy.SelectLayerByAttribute_management
   (newlayer2,selectiontype2,
   ' "'+fieldname0+'" '+op1+val1)                   # STEP 15
op0 = ">"                                           # STEP 16
arcpy.Delete_management(featureclass2)              # STEP 17
arcpy.SelectLayerByAttribute_management
   (newlayer2,selectiontype2,
   ' "'+fieldname1+'" '+op0+val1)                   # STEP 18
```

Figure 10: Test with reduction-only for ArcPy.

other crashes, normalization reduced the tests from 18 to 14 steps, from 27 to 20 steps, and from 20 to 16 steps. One crash fault only reduced from 10 steps to 9 steps, but the omission was informative. None of the ArcPy faults experienced slippage — the normalized test was always clearly the same fault as the reduced test. The cost of normalization is high — in our runs, it has taken from 17,340 seconds up to 24,769 seconds. However, in this setting even delta-debugging is extremely expensive — the cost of reduction alone has ranged from 7,930 seconds to 8,688 seconds. Generalization has taken between 3,203 and 11,149 seconds. These high costs are due to the need to run tests in a sandbox environment to avoid killing the testing process, and the runtime of complex GIS analyses. Even under these circumstances, reducing, normalizing, and generalizing tests has been a more effective use of human time than trying to understand the faults without help. For example, in the test shown in this paper, it was important to understand that the SQL query and selection type are not essential, but using a freshly created layer will not result in a crash: the problem appears to be that ArcGIS (or ArcPy) does not invalidate layers built from a feature class when that feature class is deleted. In this instance, a generalization (the fresh values generalization in particular) is informative by its absence: we know that it was attempted, but prevented failure. The reduced, non-normalized test (Figure 10) makes this far less clear, as the use of `CopyFeatures` and the multiplicity of shapefiles involved disguises the essence of the problem.

We are also preparing a test suite that covers as much as possible of the Python source in the latest version of ArcPy and records the values returned. For future versions of ArcPy, a "semantic diff" based on these calls can be produced, allowing developers to see how API usage changes with new releases. The tests in the suite are normalized and generalized (based on code coverage and output, not failure — these tests all pass) to make them easy to understand, and show which parameter combinations do not change results.

```
shapefilelist0 =
   glob.glob("C:\Arctmp\*.shp")        # STEP 0
#[
shapefile0 = shapefilelist0 [0]        # STEP 1
newlayer0 = "l1"                       # STEP 2
#  or newlayer0 = "l2"
#  or newlayer0 = "l3"
#  swaps with steps 3 4 5 6 7
#] (steps in [] can be in any order)
#[
featureclass0 = shapefile0             # STEP 3
#  swaps with step 2
fieldname0 = "newf1"                   # STEP 4
#  or fieldname0 = "newf2"
#  or fieldname0 = "newf3"
#  swaps with steps 2 8
selectiontype0 = "SWITCH_SELECTION"    # STEP 5
#  or selectiontype0 = "NEW_SELECTION"
#  or selectiontype0 = "ADD_TO_SELECTION"
#  or selectiontype0 = "REMOVE_FROM_SELECTION"
#  or selectiontype0 = "SUBSET_SELECTION"
#  or selectiontype0 = "CLEAR_SELECTION"
#  swaps with steps 2 8
op0 = ">"                              # STEP 6
#  or op0 = "<"
#  swaps with steps 2 8
val0 = "100"                           # STEP 7
#  or val0 = "1000"
#  swaps with steps 2 8
#] (steps in [] can be in any order)
arcpy.MakeFeatureLayer_management
   (featureclass0, newlayer0)          # STEP 8
#  swaps with steps 4 5 6 7
arcpy.SelectLayerByAttribute_management
   (newlayer0,selectiontype0,
   ' "'+fieldname0+'" '+op0+val0)      # STEP 9
arcpy.Delete_management(featureclass0) # STEP 10
arcpy.SelectLayerByAttribute_management
   (newlayer0,selectiontype0,
   ' "'+ fieldname0+'" '+op0+val0)     # STEP 11
```

Figure 11: Normalized and generalized ArcPy test.

4 RELATED WORK

This work builds on the idea behind delta-debugging [50]: tests should not contain extraneous information that is not needed to reproduce failure (or some other behavior [20, 21]). Delta-debugging and slicing [40] are limited, generally, to producing subsets of the original test, not modifying parts of the test to obtain further simplicity. We extend this concept by also allowing modification or re-ordering, which also allows further length reduction.

Normalization is in part motivated by the fuzzer taming [8] problem: determining how many distinct faults are present in a large set of failing tests. This is a key problem in practical application of automated testing. Previous work on fuzzer taming [8] used delta-debugging to reduce some tests to syntactic duplicates.

Zhang [52] proposed an alternative approach to semantic test simplification that, like our approach, is able to modify, rather than simply remove, portions of a test. However, because Zhang operates directly over a fragment of the Java language, rather than using an abstraction of test actions allowed, the set of rewrite operations performed is highly restricted: no new methods can be invoked, statements cannot be re-ordered, and no new values are used. These restrictions limit the approach's ability to simplify tests and make it inappropriate for normalization, as opposed to simplification. The approach also performs little syntactic normalization: e.g., it does not even force a test to use fixed variable names when variable name is irrelevant. CReduce [44] performs some simple normalization as part of a complex test reduction scheme for C code, and the peephole-rewrite scheme used in CReduce is also an inspiration for the approach taken by our normalizer.

Work on automatically producing readable tests [10, 11] is also related, in that it aims to "simplify" tests. Readable tests are intended to assist debugging by humans, while our normalization and generalization aims to increase the information density of a test, further reduce length, and address the fuzzer taming problem. The approaches are orthogonal and could likely be profitably combined: users might be best served by normalized, generalized tests modified to improve readability.

The most closely related work to our generalization efforts is Pike's SmartCheck [43]. SmartCheck targets algebraic data in Haskell, and offers an interesting alternative approach to reduction and generalization. Test generalization is also akin to dynamic invariant generation, in that it informs the user of invariants over a series of test executions [16]. The only other work we are aware of that is similar to generalization concerns essential and accidental aspects of model checking counterexamples [26, 30, 38].

5 CONCLUSIONS AND FUTURE WORK

This paper introduces test normalization and generalization. The methods presented are significant steps towards a difficult goal: providing users of automated testing with a *single test, as short and simple as possible, for each underlying fault in the SUT*, and *annotations describing the general conditions under which the fault manifests as failure*. Normalization approaches this ideal by rewriting numerous distinct failing tests into a smaller, often minimal, set of simpler tests. Generalization uses automated experiments to distinguish essential and accidental elements of a test. In our experiments, normalization reduced the number of failures to examine by well over an order of magnitude, often to the ideal of one per fault, and reduced the length of tests beyond what is possible with delta-debugging alone. While there is doubt about the utility of automatic fault localization [42] in real-world debugging, few practicing testers doubt the value of being provided with a minimal number of minimally-sized failing tests [23, 39, 45].

The algorithms for normalization and generalization depend only on a (possibly somewhat arbitrary) total order over test actions and an abstract form for tests, suitable for term rewriting. Our approach is therefore likely applicable to any source language and many different test generation methods, including those that already produce short tests [9, 18]. TSTL-based normalization and generalization are currently available in a well-tested Python version [28, 29]; there is also a beta version, with more limited normalization, for Java. The goal of normalization and generalization can also be pursued in settings other than API sequence or string grammar testing. The difficulties of defining a normal form for JavaScript [46] or C [49] tests are non-trivial, but not obviously overwhelming [44]. Less effective methods than ours might still aid debugging and assist fuzzer taming [8]. Simple generalization (e.g., is this numeric constant essential, can these two statements be swapped?) and a limited form of fresh value generalization should be easy to apply, even for complex programming language tool tests.

The working version of TSTL [29] supports normalization and generalization. Although derived via lengthy experiment-driven evolution, our rules are likely not yet ideal (though many "obvious" optimizations such as applying alpha-conversion to lower pool indices before normalization turn out to be surprisingly harmful). Further experimental evaluation of normalization and generalization over more SUTs is important to quantify effectiveness and motivate new rewrites and generalizations. The TSTL implementations are designed to allow these to be easily added, in order to bring testing closer to the goal of "one test to rule them all."

REFERENCES

[1] NumPy. http://www.numpy.org.

[2] SymPy. http://www.sympy.org/en/index.html.

[3] What is ArcPy? http://resources.arcgis.com/EN/HELP/MAIN/10.1/index.html#//000v000000v7000000.

[4] J. Andrews, Y. R. Zhang, and A. Groce. Comparing automated unit testing strategies. Technical Report 736, Department of Computer Science, University of Western Ontario, December 2010.

[5] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering*, pages 402–411, 2005.

[6] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[7] A. Arcuri, M. Z. Z. Iqbal, and L. C. Briand. Formal analysis of the effectiveness and predictability of random testing. In *International Symposium on Software Testing and Analysis*, pages 219–230, 2010.

[8] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–208, 2013.

[9] D. Coppit, J. Yang, S. Khurshid, W. Le, and K. Sullivan. Software assurance by bounded exhaustive testing. *IEEE Transactions on Software Engineering*, 31(4):328–339, Apr. 2005.

[10] E. Daka, J. Campos, J. Dorn, G. Fraser, and W. Weimer. Generating readable unit tests for Guava. In *Search-Based Software Engineering - 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings*, pages 235–241, 2015.

[11] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer. Modeling readability to improve unit tests. In *Foundations of Software Engineering, ESEC/FSE*, pages 107–118, 2015.

[12] A. Danial. CLOC: Count lines of code. https://github.com/AlDanial/cloc.

[13] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[14] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computing*, 3(1):69–115, 1987.

[15] erezbibi. https://pypi.python.org/pypi/my_xml/0.1.1.

[16] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[17] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[18] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419. ACM, 2011.

[19] P. Gastin, P. Moro, and M. Zeitoun. Minimization of counterexamples in SPIN. In *SPIN Workshop on Model Checking of Software*, pages 92–108. Springer-Verlag, 2004.

[20] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr. Cause reduction: Delta-debugging, even without bugs. *Journal of Software Testing, Verification, and Reliability*. accepted for publication.

[21] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr. Cause reduction for quick testing. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 243–252. IEEE, 2014.

[22] A. Groce and M. Erwig. Finding common ground: Choose, assert, and assume. In *International Workshop on Dynamic Analysis*, pages 12–17, 2012.

[23] A. Groce, K. Havelund, G. Holzmann, R. Joshi, and R.-G. Xu. Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence*, 70(4):315–349, 2014.

[24] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.

[25] A. Groce and R. Joshi. Random testing and model checking: Building a common framework for nondeterministic exploration. In *Workshop on Dynamic Analysis*, pages 22–28, 2008.

[26] A. Groce and D. Kroening. Making the most of BMC counterexamples. *Electron. Notes Theor. Comput. Sci.*, 119(2):67–81, Mar. 2005.

[27] A. Groce and J. Pinto. A little language for testing. In *NASA Formal Methods Symposium*, pages 204–218, 2015.

[28] A. Groce, J. Pinto, P. Azimi, and P. Mittal. TSTL: a language and tool for testing (demo). In *ACM International Symposium on Software Testing and Analysis*, pages 414–417, 2015.

[29] A. Groce, J. Pinto, P. Azimi, P. Mittal, J. Holmes, and K. Kellar. TSTL: the template scripting testing language. https://github.com/agroce/tstl.

[30] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.

[31] K. Halas. MutPy 0.4.0. https://pypi.python.org/pypi/MutPy/0.4.0.

[32] R. Hamlet. When only random testing will do. In *International Workshop on Random Testing*, pages 1–9, 2006.

[33] J. Holmes, A. Groce, and A. Alipour. Mitigating (and exploiting) test reduction slippage. In *Workshop on Automated Software Testing*, 2016.

[34] J. Holmes, A. Groce, J. Pinto, P. Mittal, P. Azimi, K. Kellar, and J. O'Brien. TSTL: the template scripting testing language. *International Journal on Software Tools for Technology Transfer*, 2017. Accepted for publication.

[35] G. Huet and D. C. Oppen. Equations and rewrite rules: A survey. Technical report, Stanford, CA, USA, 1980.

[36] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[37] G. Jenks. SortedContainers introduction. http://www.grantjenks.com/docs/sortedcontainers/introduction.html.

[38] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 445–458, 2002.

[39] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *International Symposium on Software Reliability Engineering*, pages 267–276, 2005.

[40] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *International Conference on Automated Software Engineering*, pages 417–420, 2007.

[41] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[42] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis*, pages 199–209, 2011.

[43] L. Pike. SmartCheck: automatic and efficient counterexample reduction and generalization. In *ACM SIGPLAN Symposium on Haskell*, pages 53–64, 2014.

[44] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 335–346, 2012.

[45] J. Ruderman. Bug 329066 - Lithium, a testcase reduction tool (delta debugger). https://bugzilla.mozilla.org/show_bug.cgi?id=329066, 2006.

[46] J. Ruderman. Introducing jsfunfuzz, 2007. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.

[47] J. Ruderman. Releasing jsfunfuzz and DOMFuzz. https://www.squarefree.com/2015/07/28/releasing-jsfunfuzz-and-domfuzz/, 2015.

[48] user1689822. python AVL tree insertion. http://stackoverflow.com/questions/12537986/python-avl-tree-insertion.

[49] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294, 2011.

[50] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.

[51] C. Zhang, A. Groce, and M. A. Alipour. Using test case reduction and prioritization to improve symbolic execution. In *International Symposium on Software Testing and Analysis*, pages 160–170, 2014.

[52] S. Zhang. Practical semantic test simplification. In *International Conference on Software Engineering*, pages 1173–1176, 2013.