One Test Case to Rule Them All

Alex Groce,* Josie Holmes† Kevin Kellar‡
*School of Electrical Engineering and Computer Science
Oregon State University
† Department of Geography
Pennsylvania State University ‡ Crescent Valley High School

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

I. Introduction

It has long been understood that effective automated testing often requires test case reduction [1], [2], [3], [4] to produce test cases without irrelevant operations[1]. In fact, test case reduction is now standard practice in industrial testing tools such as Mozilla's `jsfunfuzz` [11], [12], [13]. However, simply reducing the length of a test case does not produce any kind of semantic simplicity. There may be many (often far more than a thousand, in practice) 1-minimal test cases that present different variations of a single fault. Far too often, reading more than one of these test cases provides no useful additional information on the cause of failure.

Consider the three test cases shown in Figure 1. These test cases are obviously very similar, and in fact all lead to a violation of the property that an AVL tree must always be nearly balanced, due to a missing call to `rebalance` in `delete` in a Python implementation. However, the test cases are syntactically very different, and a testing system that collects failing test cases will present all three of these tests for a user to examine. While there are methods for attempting to determine which test cases represent distinct faults [14], the ideal solution is to convert all three of these test cases into a single, *canonical* form that preserves the structure of the failure while de-emphasizing such accidental aspects of each test case as the particular integer values and variables used, and the ordering of assignments and insertions. Unlike delta-debugging, this concept applies even to test generation methods that produce short or even guaranteed-minimal tests.

Figure 2 shows the result of applying our *test case normalization* algorithm to the three tests in Figure 1, and then applying our *test case generalization* algorithm to the normalized test case. *All three test cases normalize to the same test case*. The test case also includes comments, produced by our *generalization* [15] algorithm, indicating what about the test case can be changed while preserving the property that the test fails. For example, the value 1 assigned to `int0` in step 0 is not essential. It could be changed to any value in the range 5-20 (1-20 is the range of values allowed by the test generator) without changing the final result. The same is true of the assignment of 3 to `int1`. Similarly, the exact ordering of many steps in the test case is not important. Finally, step 9 is annotated to show that instead of using the existing value of `int1` (4), a fresh assignment could be inserted before the `delete` call, setting `int1` to 3 instead. These possible changes are not meant to be combined — the annotation claims only that changing these aspects of the test case one at a time will preserve failure. This annotation provides information allowing a user to understand that a single test case is representative of a *family* of test cases that cause the same kind of failure.

Combining normalization and generalization avoids some common problems with understanding automatically generated test cases. For instance, when a large integer appears in such a test case, the question always arises — is this unusual value important, or just a random number of no significance [16]? After normalization, any large values in a test case are essential, rather than accidental, because normalization includes value minimization. Without the additional step of generalization it would be easy to assume all small numeric values in normalized tests are accidental. Generalization informs a user when a small value is required to reproduce a failure, and when it is simply an artifact of normalization.

Normalization is not 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 [17] or `jsfunfuzz` [11]), but it is often highly effective. Running 100,000 tests (of length 100) on the faulty AVL tree produces 860 failing test cases with no duplicates. Normalizing these

[1]Algorithms such as bounded-exhaustive testing [5] or some search-based approaches that optimize for short test cases [6] may not require reduction, but random testing [7], [8], model-checking [9], and other approaches, including symbolic execution [10] can benefit from test case/counterexample reduction.

reduces the number of distinct failing test cases to just 22. Ideally *all* failures due to the same fault in the SUT (Software Under Test) would normalize to a single, representative test case. We aim to *approximate* such a canonical form for faults. Figure 4, in Section II-C, shows an AVL tree test case that normalizes differently. The graph structure required for the `delete` fault results in an unusually poor effectiveness for normalization. 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). This fault, in contrast, produced 18 distinct failures. Nonetheless, determining that the failure in Figure 4 (or one of the similarly structured other 20 failures) is due to the same fault as the failure in Figure 2 should be much less difficult than performing the same task over many hundreds of failures with much more extreme variation.

The contributions of this paper are 1) normalization and generalization for API-based test cases presented as key steps towards a goal of "one test case to rule them all" (per fault), 2) algorithms for normalizing and generalizing test cases that make use of the abstract graph interface for testing provided by the TSTL [18], [19], [20] domain-specific language (DSL) [21], and 3) initial experimental results showing the value of normalization and generalization. Normalization preserves fault detection approximately as well as test case reduction via delta-debugging, and often reduces the set of test cases to be examined by more than an order of magnitude. In many cases, the normalization is perfect (one test case per fault). Normalization and generalization have also been useful in understanding complicated test cases for a variety of software systems, including TSTL itself, the NumPy library for numeric computation, and ArcPy, a widely-used, highly complex package for Geographic Information System (GIS) automation.

II. Formal Definitions of Normalization and Generalization

A. A Brief Introduction to TSTL

TSTL [19], [20] is a language for defining the structure of test cases (usually API-call sequences, but also grammar-based tests using string construction), and a set of tools for use in generating, manipulating, and understanding those test cases. Figure 3 shows a simplified portion of a TSTL definition (called a harness) of tests for an AVL tree class, in the latest syntax for TSTL (which differs slightly from syntax in the cited papers introducing TSTL). TSTL provides numerous features not shown in this small example, including automatic differential testing, complex logging, support for complex guards, and use of pre- and post- values. 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 [18] provides testing tools that use the interface for test generation 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 [22] (a common approach to defining API-testing sequences) and a set of actions that are possible during testing, typically API calls and assignments to pool values. In

Test case #1

```
avl0 = avl.AVLTree()
int0 = 4
int2 = 13
int3 = 7
avl0.insert(int2)
avl0.insert(int3)
int1 = 15
avl0.insert(int1)
avl0.insert(int0)
avl0.delete(int2)
```

Test case #2

```
int0 = 14
avl0 = avl.AVLTree()
int2 = 13
int1 = 15
avl0.insert(int1)
int1 = 11
avl0.insert(int2)
avl0.insert(int0)
avl0.insert(int1)
avl0.delete(int0)
```

Test case #3

```
avl1 = avl.AVLTree()
int3 = 18
avl1.insert(int3)
int0 = 5
int3 = 12
avl1.insert(int0)
int0 = 15
avl1.insert(int0)
avl1.insert(int3)
int1 = 15
avl1.delete(int1)
```

Fig. 1. Three randomly generated test cases for the same 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)
#     )
```

Fig. 2. Normalization and generalization for all three test cases. Lines beginning with # are comments in Python, used for annotations.

```
@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()
```

Fig. 3. Part of a TSTL definition of AVL tree tests.

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 an AVL tree's `insert`, `delete`, `find` and `inorder` methods. Figure 1 in the introduction shows three valid test cases 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 case is called a "step" — the first step of the first test case in Figure 1 is storing a new AVL tree in `avl0`, for example. A test case is just an ordered sequence of actions, which is equivalent to a set of numbered steps.

The definition of pools and actions in TSTL defines a *total order* on all actions. First, actions are ordered by their position in the 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 of TSTL code `<avl>.insert(<int>)` defines 12 actions, one for each choice of `avl` and `int` pool instance: the action set includes `avl0.insert(int0)`, `avl1.insert(int0)`, `avl0.insert(int1)`, and so forth. These are ordered lexically, with the first pool appearing in the text taking precedence (`avl0.insert(int2)` precedes `avl1.insert(int0)`, etc.). 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 [22]. However, it also allows us to concisely define a practical method for normalizing and generalizing test cases.

In normal TSTL semantics, assigning a value to the same pool twice in a row is not allowed; until a pool value is used, it cannot be assigned to again. However, during normalization and generalization, this restriction is removed. The restriction exists only to prevent the generation of uninteresting test sequences. The normalization and generalization algorithms avoid such sequences by other means, and allowing overwriting assignments makes normalization much more effective. Such sequences only appear in intermediate steps; after normalization, test cases are guaranteed to not have any remaining sequences invalid in normal TSTL semantics.

B. Normalization

A test-case normalization algorithm has a simple goal: we ideally aim to produce a function $f : t \rightarrow t$ (a function that takes a test case and returns a test case) such that:

1) If t fails, $f(t)$ fails.
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 test cases, where each underlying fault is uniquely represented by a single test case. In general, it seems clear that defining such a function f is (at least) as difficult as automatic fault localization and repair. Therefore, we aim at approximating the goal, by providing a set of simple transformations such that f changes many tests to the same test, f has low probability of changing two tests failing for different reasons into the same test, and 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 $t' \in C(t)$, possible simplifications of t, where each t' is the result of applying some rewrite, r_i to t. The notion of simplicity is defined by a restriction on the rewriting rules. For any test case t and rewrite $r_i : r_i(t) \in C(t)$, we require that $|C(r_i(t))| < |C(t)|$. Such a rewrite system is necessarily *strongly normalizing*: any sequence of rewrites chosen will eventually end in a term (test case) that cannot be further rewritten, and the total length of a rewrite sequence in this case is bounded by the initial $|C(t)|$.

In the setting of TSTL, where test actions have a defined total order, two simple principles can be applied to produce useful rewrite rules. First, rewrites must reduce the sum of the indices of the actions in the test case, make the test case'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 case 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 case. We cannot guarantee normalization does not change the underlying fault in a test; however, the limited scope of rewrites should at least make the likelihood of fault change (known as "slippage" [14]) no worse than that of delta-debugging, which is widely accepted as a reasonable tradeoff.

1) Definitions and Notation: In order to formally define normalization, some additional notation is required. A *step* is an action paired with an index indicating its position in a test case, 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 $t(i) \neq t'(i)$.

We use the $<$ operator over various types: $a < b$ 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 rewrite $t[x \Rightarrow y]$ 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. Rewrite $t(i, j)[x \Rightarrow y]$ is the same as $t[x \Rightarrow y]$, 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) Rewrite Rules:

1) **SimplifyAll:** $t' = t[a \Rightarrow a']$
   where $a' < a$
   Covers the case where all appearances of an action can be replaced with a lower-indexed action.
2) **ReplacePool:** $t' = t(i, j)[p \Rightarrow p']$
   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 (with possible restriction to a range of steps).
3) **ReplaceMovePool:** $t' = t_{\rightarrow i}(p')[p \Rightarrow p']$
   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 assignments to the new instance before a certain step are moved to that step.
4) **SimplifySingle:** $t' = t[(i : a) \Rightarrow (i : a')]$
   where $a' < a$
   Covers the case where one action can be replaced with a simpler (lower-indexed) action.
5) **SwapPool:** $t' = t(i, j)[p \Leftrightarrow p']$
   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 minimal action index of the modified steps.
6) **SwapAction:** $t' = t[(i : a) \Rightarrow (i : b), (j : b) \Rightarrow (j : a)]$
   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.
7) **ReduceAction:** $t' = t[(i : a) \Rightarrow (i : a')]$
   where $|ddmin(t')| < |t|$
   Covers the case where an action can be replaced by any action (not just lower-indexed actions) and this enables further delta-debugging.

3) 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 [23]. Notice that after applying each rewrite rule, we perform delta-debugging on the new base test case, since often a rewrite makes other steps irrelevant.

Algorithm 1 Basic algorithm for normalization

```
1: modified = True
2: while modified do
3:     modified = False
4:     for t' ∈ C(t) do
5:         if pred(t') then
6:             modified = True
7:             t = ddmin(t')
8:             break (exit for loop)
9:         end if
10:    end for
11: end while
12: return t
```

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 case, there can be at most n predicate checks for the current test case, then at most $n - 1$ checks for the rewritten test case, and so on, for a total of $\frac{n(n+1)}{2}$ predicate checks, where n is the number of possible rewrites of the test case t being normalized, e.g., $n = |C(t)|$. Test case execution to check the predicate can be assumed to have a constant cost since the length of the test case does not usually change by more than a few steps during normalization.

How many rewrites can a test case t have? Assume there are k steps in the test case 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**[2]. 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 [1]. 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 test cases, unlike delta-debugging. The k in our quartic complexity is therefore, for random test cases, typically an order of magnitude smaller than the k in delta-debugging [4], [23], [2], which can partly balance the additional cost for test cases whose unreduced length is less than 100.

We believe that if normalization can decrease human effort in examining large numbers of redundant test cases, 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 test cases. 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 2-10x, usually closer to 10x. Measuring the payoff

[2]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 below.

from improved understanding of test cases is difficult; however, in multiple fault settings, normalization can often provide a quantifiable value in shorter time until all faults have been examined by a human, for bug triage [14].

4) Normalization Optimizations: The simplest and most important optimization is to improve on the constant ordering of rewrite rules. In our implementation, once a rule fails to produce a candidate that satisfies `pred`, that rule is moved to the end of the ordering of rewrites. This optimization followed the observation that once a rule fails once to produce any valid rewrites for a test case, it frequently produces no further reductions. This simple change typically halves the time required for normalization.

For test cases with a very short runtime, this algorithm (with reordering) is usually practical: while more expensive than delta-debugging it still requires less than a minute to run. However, when test case execution is expensive, the set of candidate test cases must be further restricted. In our experiments, we found that restricting action replacements to cases where the Levenshtein [24] distance (text edit distance) between the code for the actions was bounded to some small value (5 to 25 characters) was effective in reducing runtime, and often had no impact on the final result. In practice, most test actions can only be replaced by syntactically similar actions, without completely changing test semantics.

A further useful optimization when normalizing large numbers of tests of the same SUT is to cache results across tests: as soon as the rewrite sequence produces a previously seen test, the final result can be returned (since the algorithm is deterministic). For very large numbers of tests, this is the most important optimization. Interestingly, although $ddmin$ is also deterministic, caching delta-debugging results across tests is far less useful, since the chance of seeing an exact match is extremely small. This is true for the *initial* input to normalization, but a small number of pool and value changes often result in a cache hit. For systems with very expensive replay, the delta-debugging of each new base test case can also be omitted: the **ReduceAction** rewrite will eventually remove the extraneous steps. However, the reduced calls to *ddmin* are offset by attempts to rewrite steps that could be removed.

It is also trivial to parallelize the normalization of a test case by checking the predicate over multiple candidates at once. As soon as a candidate satisfies the predicate, a parallel implementation faces two possible choices. Either the normalization 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 first in the sequential version of the algorithm is in the work queue.

C. Normalization Example

Figure 4 shows the sequence of successful rewrites in normalizing a simple AVL tree test failure. Note that 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.

Original test case:

```
 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)
```

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)`

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)
```

Fig. 4. An example of normalization steps.

D. Generalization

1) Generalization Algorithm: 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 case. Some values and orderings of steps in a test case are *essential* to the failure: when changed, they cause the test case to no longer fail. Many others, however, are *accidental* — any concrete test case 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 case, and summarizes the results. The core algorithm (Algorithm 2) is simple.

Algorithm 2 Basic algorithm for generalization

```
1: swap = ∅
2: replace = ∅
3: for (i, a) ∈ t do
4:     for a′ : a′ > a do:
5:         if pred(t[(i, a) ⇒ (i, a′)]) then
6:             replace = replace ∪((i, a), (i, a′))
7:         end if
8:     end for
9:     for j : i < j < |t| − 1 ∧ (j : b) > (i : a) do
10:        if pred(t[(i : a) ⇒ (i : b), (j : b) ⇒ (j : a)]) then
11:            swap = swap ∪((i, a), (j, b))
12:        end if
13:    end for
14: end for
15: return (swap, replace)
```

```
#[
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)
#     )
```

Fig. 5. Generalized test for TSTL itself, showing fresh value generalization.

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 case 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 case 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 case are not the lowest-indexed action. Basic generalization is trivial to parallelize, as all checks are independent.

2) Fresh Values and Misleading Test Cases: A side-effect of delta-debugging and normalization is reduction of the number of variables in a test case. While usually helpful, this can sometimes result in misleading test cases. 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 case [4], [25] 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 produce the problem. We therefore present a more aggressive generalization than replacement or swap: replacing a pool use with a *fresh value*.

Consider the test case in Figure 5, produced by a TSTL harness for TSTL itself[3]. 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 this test again. The annotation after step 6 lets us see that the failure will take place even for a new empty test. Without this generalization, the state of the `test` may appear to be important, not the state of TSTL itself.

Formalizing this generalization requires some additional notation. $U(a)$ provides a list of pools *used* in the action a — pools that appear in the action, 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 = ∅
2: for (i, a) ∈ t do
3:     for p ∈ U(a) do
4:         for a' : I(a', p) do
5:             if pred(t[+(a', i)]) then
6:                 fresh = fresh ∪(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 the sequence `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. 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 as 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 needed inputs to constructors were usually available in pools. It is not clear how important it is to know *all* values that could result in a valid fresh value.

E. Discussion

TSTL's conversion of testing into a graph exploration problem [19], where a test generation algorithm can be agnostic as to the underlying SUT, or even the language of the system under test, enables us to produce semantic normalization and generalization by simple syntactic means. The current TSTL implementation is in Python[4], but the normalization and generalization algorithms do not depend on the underlying language, only on the abstract notions of actions and pools. One intuitive concern with this approach is that a user may not place actions in a "good" order in the TSTL definition. What if normalization rewrites actions into "more complex" rather than simpler actions? In fact, 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 is *indifferent*

[3]Since TSTL provides a Python API, that API can be used as the SUT in testing; we have discovered several important TSTL bugs this way.

[4]There is also a beta Java version, with a simpler version of normalization.

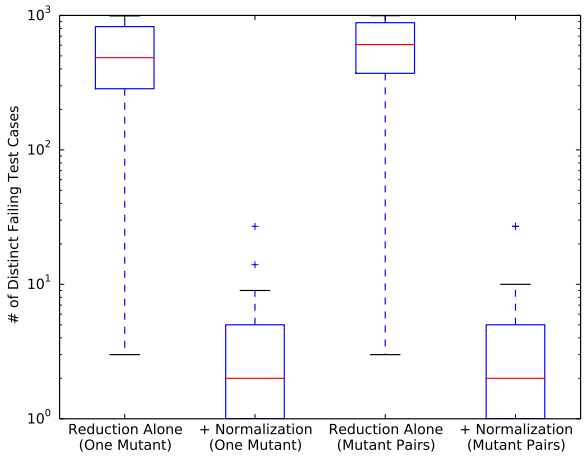

Fig. 6. Effects of normalization on 82 AVL Tree mutants.

to the choice. The normal form's *existence* matters. not the precise contents, in the context of that semantic restriction.

III. Case Studies

This section presents some initial results of applying normalization and generalization. All tests were generated using random testing. We also tested the Python interface to Z3 [26], but did not find faults thus far; normalization did help produce more comprehensible and uniform Z3 quick tests [23].

A. AVLTree

For basic experiments on the ability of normalization to reduce the number of failing test cases and preserve fault detection, we used a simple Python implementation of AVL trees found on the web [27], with 225 lines of code[5].

Figure 6 shows the reduction in number of distinct failing test cases produced by normalization (vs. reduction only) for a set of 82 mutants [29] of the AVLTree source code [30]. Of the 228 mutants produced by MutPy [31], only these produced at least 1 failure in 1,000 test cases. Using only delta-debugging-based reduction, the mean number of distinct failures for each mutant (which is, by definition, a single fault) was 498.4, with a median of 485. Using normalization, the mean was 3.1 distinct failures, with a median of just 2 failures. For 38 of the 82 mutants, normalization produced only a single representative failure. For every mutant, normalization reduced the number of distinct failures, and no mutant produced more than 27 distinct failures, after normalization.

The mean cost to reduce a test case 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 case, 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 an average of only 2.4 non-cache-hit rewrites of each test. Note that this is with the cache starting empty for each of the 82 mutants.

AVLTree mutants also provided a way to evaluate the danger of normalization losing faults. Faults can be lost when normalization changes a test case failing due to one fault into a test case failing due to a different fault, a problem known as "slippage" [14]. AVLTree provides a slippage challenge, as there are very few API calls, and all calls take the same inputs, so test cases 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 test cases included at least one test capable of exposing each of the two faults for only 238 pairs. In almost 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 case that exposes a fault in `delete` on a non-empty tree.

Out of these 238 mutant pairs, normalization produced test cases 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 test cases not capable of exposing both faults, and produced a smaller set of test cases that was capable of detecting both faults. Arguably this is "slippage" in that a test case not capable of exposing fault F was modified to expose fault F (classic slippage) but since the new test suite exposed both faults, normalization actually restored fault detection.

Data on slippage is not extensive, in part because it is only detected if the original test cases before reduction are stored and re-executed after bugs have been fixed. In our own previous work [14], slippage due to reduction was extremely rare for some SUTs (GCC 4.3.0) and very common for other SUTs (up to 23% for Mozilla's JavaScript engine). For our own AVLTree example, the slippage rate for reduction is almost 30%, considerably worse than that for normalization. As a simple mitigation, we propose storing at least reduced test cases, and ideally the unreduced test cases, as a slippage check when all faults detected by normalized test cases are fixed.

The reduction in test cases produced by normalization for mutant pairs was, as with single mutants, very large (Figure 6). For mutant pairs, the mean/median number of distinct failures was 554.4/607 for reduction alone, but only 2.8/2 for reduction + normalization. The runtime for normalization was essentially unchanged from the single-mutant numbers given above. 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

[5]All code sizes non-comment, non-blank lines, measured by cloc [28].

fault, normalization worked as well as possible — it preserved both faults and produced only 1 or 2 test cases[6].

B. XML Parser

We also examined how normalization combined with multiple faults for a simple XML parser with about 260 lines of code [32], with 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. Running 1,000 tests produced 848 failing test cases. 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 reduced the number of failures to just 5: 3 for the original fault and 2 for the seeded fault.

C. TSTL

As noted in Section II-D2, TSTL is used to test TSTL's own API interface (the TSTL compiler is about 1,600 LOC; a compiled SUT is often 30KLOC or larger). We only discovered one fault while testing the latest version of TSTL, the cache-related problem shown in Figure 5. Generating and reducing 100 test cases 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.

D. NumPy

Our final two case studies provide little information on the ability of normalization to reduce the number of failing tests; for these SUTs, failure rates are low enough or test case reduction runtimes high enough that each failure is dealt with one-by-one. However, normalization and generalization are also useful for understanding individual test cases.

NumPy [33] 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 a simple example, consider the test case in Figure 7, normalized and generalized in Figure 8. Prior to normalization, understanding why the test case leads to a violation of self-equality for an array is difficult. After normalization, it is much clearer what is happening: 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 case 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 case, but our data (for fewer than 30 failures) is insufficient to make a definite claim, as the failure rate is slightly less than 3 failures/100,000 tests on average.

Other, more complex, failures have also made it clear that normalization is useful for additional test case length reduction

[6]It is possible to detect both faults with a single test case, if that test case fails when *either* fault is present.

```
dim1 = 1
shape2 = (dim1, dim1, dim1)
array1 = np.ones(shape2)
array0 = array1 * array1
array1 = array1 + array1
array4 = array0 + array1
array0 = np.reshape(array4,shape2)
array3 = array1 * array4
array2 = np.ravel(array4)
array5 = array2 - array3
array4 = array5 * array2
array1 = np.unique(array0)
array5 = array5 * array3
array0 = array1 * array5
array5 = np.unique(array0)
array1 = array4 - array2
array2 = array0.flatten()
array0 = array5 + array5
array5 = array5 + array2
array2 = array0 * array2
np.copyto(array5,array2)
array2 = array2 * array5
array3 = array0 * array2
array0 = array3 - array1
array4 = array3 * array0
array1 = array5 + array4
array5 = array0 * array1
array0 = array5 - array1
array4 = array0 * array3
array3 = array4 * array0
array1 = array5 + array3
array0 = array2 + array1
array5 = array5 - array0
array5 = array3 * array5
array0 = array1 + array5
array2 = array3 - array0
array4 = array2 * array1
array3 = array4 * array2
array2 = array0 - array0
np.copyto(array1,array3)
array4 = array2.flatten()
array1 = array1 * array4
assert (np.array_equal(array1,array1))
```

Fig. 7. Original "failing" test case for numpy (42 steps).

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 test cases, the average time to reduce tests is about 4-5 seconds, and the time for normalization is between 712 and 774 seconds. The average time to discover a failing test case, for comparison, is just over 400 seconds. Generalization takes between 52 and 59 seconds in these cases. The exception was a test case of 45,206 steps (!) leading to a memory exhaustion error and crash. This was reduced (over a period of nearly a day) to a test case with 10 steps, which then normalized (in only 2 hours) to a test case with 8 steps. The normalized test case involved no operations other than array initialization, array flattening, and array addition. The reduced test case required larger dimensions, array multiplication, and array subtraction, as well. This is the only case in which we have seen normalization time lower than reduction time, without assistance from the cache.

E. Esri ArcPy

Esri is the single largest Geographic Information System (GIS) software vendor, with about 40% of global market share. Esri's ArcGIS tools are extremely widely used for GIS analysis, in government, scientific research, commercial enterprises, and education. Automation is essential for complex

```
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))
```

Fig. 8. Normalized and generalized test case for numpy (15 steps).

GIS analysis and data management, and Esri has long provided tools for programming their GIS software systems. One such tool (Esri's newest) is a Python site-package, ArcPy [34]. ArcPy is a complex library, with dozens of classes and hundreds of functions distributed over a variety of of toolboxes. 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.

In order to improve the reliability of ArcPy, we are developing a TSTL-based framework for testing ArcPy itself, as well as libraries based on ArcPy. The ArcPy TSTL is already more than twice as large as the next-largest TSTL harness previously studied, even though it only includes a small portion of ArcGIS functionality so far. The first stage of testing has resulted in discovery of multiple faults in ArcPy/ArcGIS, six of which not only cause incorrect behavior but cause a crash that also ends testing. It is critical to understand these faults, the set of behaviors that trigger them, and modify the test definition to avoid triggering these faults while imposing minimal limitations on thorough testing for other faults.

There is no space in this paper to elaborate on the details of this large test effort (which has introduced numerous additional features and modes to TSTL), but normalization and generalization have been very useful in this process. Figures 9 and 10 show one crash-inducing test case, after initial delta-debugging (from over 2,000 test steps) (Figure 9) and after normalization and generalization (Figure 10). 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 other crashes, normalization reduced the test cases 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. 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 Python testing process, and the runtime of complex GIS analyses, with individual actions sometimes requiring minutes to execute. Even under these circumstances,

```
shapefile2 = "C:\arctmp\new3.shp"
shapefile1 = "C:\arctmp\new3.shp"
featureclass2 = shapefile2
featureclass0 = shapefile1
shapefilelist2 =
   glob.glob("C:\Arctmp\*.shp")
fieldname0 = "newf3"
shapefile1 = shapefilelist2 [0]
featureclass1 = shapefile1
arcpy.CopyFeatures_management
   (featureclass1,featureclass2)
op1 = ">"
newlayer2 = "l2"
val1 = "100"
selectiontype2 = "SWITCH_SELECTION"
fieldname1 = "newf1"
arcpy.MakeFeatureLayer_management
   (featureclass0, newlayer2)
arcpy.SelectLayerByAttribute_management
   (newlayer2,selectiontype2,
   ' "'+fieldname0+'" '+op1+val1)
op0 = ">"
arcpy.Delete_management(featureclass2)
arcpy.SelectLayerByAttribute_management
   (newlayer2,selectiontype2,
   ' "'+fieldname1+'" '+op0+val1)
```

Fig. 9. Original test case for ArcPy library (19 steps).

reducing, normalizing, and generalizing test cases has been a more effective use of human time than trying to understand the faults without these aids. For example, in the test case 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[7]. The original, non-normalized test case makes this far less clear, as the use of `CopyFeatures` and the multiplicity of shapefiles involved disguises the essence of the problem.

Normalization and generalization are also being used to prepare an API-behavior regression suite for ArcPy. One of the challenges of using a large API like ArcPy is that behavior of the system can change from version to version. In some cases this is due to new faults, or fixed faults, but in other cases there is an undocumented change, especially for unusual input combinations. In order to assist ArcPy developers, we are preparing a test suite that covers as much as possible of the Python source in the latest version of ArcPy, 10.3, and records the values returned. For future versions of ArcPy (or older versions), a "semantic diff" with 10.3, based on these calls, can be produced by running this suite. The tests in the suite are normalized and generalized to help users understand the set of conditions behind an API usage's behavior. This "coverage regression" quick test [23] will also be useful for understanding the API, since the set of online examples of usage from Esri is usually limited to a small set of parameter combinations.

IV. Related Work

This work builds on the idea behind delta-debugging [1]: test cases should not contain extraneous information that is not needed to reproduce failure (or some other behavior, as in more recent work [23], [35]). Delta-debugging and slicing [3]

[7]In this instance, a generalization (the fresh values generalization in particular) is informative even though it does not allow any generalization; knowing that it was attempted, but prevented the failure, is also valuable.

```
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
```

Fig. 10. Normalized and generalized test case for ArcPy library (12 steps).

are limited, generally, to producing subsets of the original test case, not modifying parts of the test to obtain further simplicity. We extend this concept by also allowing modification or reordering of actions in a test. In some cases these changes allows further reduction of the length of the test case as well. Some earlier work in bounded model checking modified counterexamples to use numerically smaller values [16] but otherwise did not aim to simplify or normalize failures.

Normalization is in part motivated by the fuzzer taming [14] problem: determining how many distinct faults are present in a large set of failing test cases. This is a key problem in practical application of automated testing. Previous work on fuzzer taming [14] used delta-debugging to reduce some test cases to syntactic duplicates.

Zhang [36] 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 [37] performs some simple normalization as part of a complex test-case 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 producing readable tests [38], [39] is also related, in that it aims to "simplify" tests in a way that goes beyond measuring test length. Work on readable tests is primarily intended to assist debugging in terms of human-factors, while our normalization and generalization aims to increase the information density of a test (potentially increasing readability as well), and to address the fuzzer taming problem. The approaches are completely orthogonal: our normalized tests could be altered to improve readability by these methods.

The most closely related work to our generalization efforts is Pike's SmartCheck [15]. SmartCheck targets algebraic data in Haskell, and offers an interesting alternative approach to reduction and generalization. Test case generalization is also akin to dynamic invariant generation, in that it informs the user of invariants over a series of test executions [40]. The only other work we are aware of that is similar to generalization concerns essential and accidental aspects of model checking counterexamples [41], [16], [42].

V. Conclusions and Future Work

This paper introduces test case normalization and generalization. The methods presented are significant steps towards a difficult goal: providing users of automated testing with a *single test case, 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 test cases into a smaller, often minimal, set of simpler test cases. Generalization uses automated experiments to identify essential and accidental aspects of a test case. In our experiments, normalization reduced the number of failures a user must examine by well over an order of magnitude compared to reduction alone, often to the ideal of one per fault. The algorithms for normalization and generalization rely on the ability of TSTL [19], [20] to define a total order over test actions, based on an abstract form for test cases, suitable for term rewriting. TSTL-based normalization is thus applicable to any SUT, any source language, and any test generation method, including methods that already produce short tests [6], [5].

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 [11] or C [37] test cases are formidable. Less effective methods than ours might still aid debugging and assist fuzzer taming [14]. 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 test cases.

TSTL is available in a working version for Python [18] that supports normalization and generalization. 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 case to rule them all."

REFERENCES

[1] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *Software Engineering, IEEE Transactions on*, vol. 28, no. 2, pp. 183–200, 2002.

[2] Y. Lei and J. H. Andrews, "Minimization of randomized unit test cases," in *International Symposium on Software Reliability Engineering*, 2005, pp. 267–276.

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

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

[5] D. Coppit, J. Yang, S. Khurshid, W. Le, and K. Sullivan, "Software assurance by bounded exhaustive testing," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 328–339, Apr. 2005.

[6] 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*, ser. ESEC/FSE '11. ACM, 2011, pp. 416–419.

[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*, 2010, pp. 219–230.

[8] R. Hamlet, "When only random testing will do," in *International Workshop on Random Testing*, 2006, pp. 1–9.

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

[10] 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*, 2014, pp. 160–170.

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

[12] ——, "Releasing jsfunfuzz and DOMFuzz," https://www.squarefree.com/2015/07/28/releasing-jsfunfuzz-and-domfuzz/, 2015.

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

[14] 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*, 2013, pp. 197–208.

[15] L. Pike, "SmartCheck: automatic and efficient counterexample reduction and generalization," in *ACM SIGPLAN Symposium on Haskell*, 2014, pp. 53–64.

[16] A. Groce and D. Kroening, "Making the most of BMC counterexamples," *Electron. Notes Theor. Comput. Sci.*, vol. 119, no. 2, pp. 67–81, Mar. 2005.

[17] 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*, 2011, pp. 283–294.

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

[19] A. Groce and J. Pinto, "A little language for testing," in *NASA Formal Methods Symposium*, 2015, pp. 204–218.

[20] 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*, 2015, pp. 414–417.

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

[22] J. Andrews, Y. R. Zhang, and A. Groce, "Comparing automated unit testing strategies," Department of Computer Science, University of Western Ontario, Tech. Rep. 736, December 2010.

[23] 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*. IEEE, 2014, pp. 243–252.

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

[25] 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*, vol. 70, no. 4, pp. 315–349, 2014.

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

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

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

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

[30] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

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

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

[33] "NumPy," http://www.numpy.org.

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

[35] 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.

[36] S. Zhang, "Practical semantic test simplification," in *International Conference on Software Engineering*, 2013, pp. 1173–1176.

[37] 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*, 2012, pp. 335–346.

[38] 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*, 2015, pp. 235–241.

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

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

[41] 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*, 2002, pp. 445–458.

[42] A. Groce and W. Visser, "What went wrong: Explaining counterexamples," in *SPIN Workshop on Model Checking of Software*, 2003, pp. 121–135.