A Suite of Tools for Simplifying and Understanding Python Tests

Josie Holmes

Alex Groce
School of Informatics, Computing, and Cyber Systems,
Northern Arizona University
agroce@gmail.com

ABSTRACT

Automated test generation tools (we hope) produce failing tests from time to time. In a world of fault-free code this would not be true, but in such a world we would not need automated test generation tools. Failing tests are generally speaking the most valuable products of the testing process, and users need tools that extract their full value. This paper describes the tools provided by the TSTL testing language for making use of tests (which are not limited to failing tests). In addition to the usual tools for simple delta-debugging and executing tests as regressions, TSTL provides tools for 1) minimizing tests by criteria other than failure, such as code coverage, 2) normalizing tests to achieve further reduction and canonicalization than provided by delta-debugging, 3) generalizing tests to describe the neighborhood of similar tests that fail in the same fashion, and 4) avoiding slippage, where delta-debugging causes a failing test to change underlying fault. These tools can be accessed both by easy-to-use command-line tools and via a powerful API that supports more complex custom test manipulations.

CCS CONCEPTS

•Software and its engineering →Software testing and debugging;

KEYWORDS

test reduction, semantic simplification, slippage, normalization, generalization

ACM Reference format:
Josie Holmes and Alex Groce. 2017. A Suite of Tools for Simplifying and Understanding Python Tests. In *Proceedings of International Symposium on Software Testing and Analysis, Santa Barbara, California, USA, July 2017 (ISSTA'17),* 7 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

It has long been understood that effective automated testing often requires test reduction [20, 34, 35, 45] to produce useful tests. In fact, test reduction is now standard practice in industrial testing tools such as Mozilla's `jsfunfuzz` [40–42].[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 [11, 30] 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* [38] 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 [21]? 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 [44] or `jsfunfuzz`

[1] Approaches that optimize for short tests [8, 14] may not require reduction, but random testing [6, 27], model-checking [15], and symbolic execution [46] can all benefit.

[2] No single component of a 1-minimal/delta-debugged [45] test can be removed without causing the test to pass.

ISSTA'17, Santa Barbara, California, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

Test #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 #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 #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)
```

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.

[41]), but it is often highly effective. Running 100,000 tests (of 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 **??**, in Section **??**, 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 [22–24, 29] domain-specific language (DSL) [13], 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.

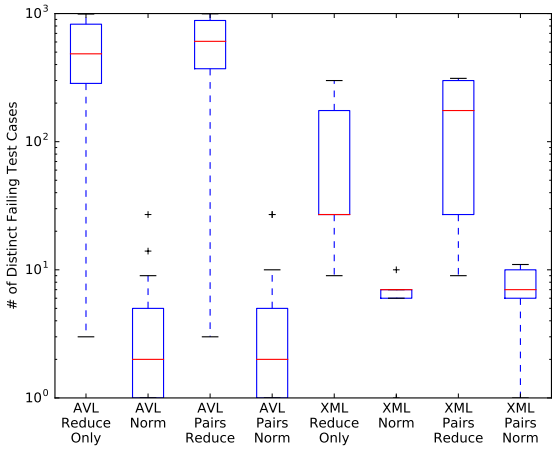

Figure 3: Effects of normalization on AVLTree and XML parser mutants.

2 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 [24]. We also tested the Python interface to Z3 [10], but did not find faults thus far; normalization did help produce more comprehensible and uniform Z3 quick tests [17].

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 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.

2.1 AVLTree

For basic experiments on **RQ1-3**, we used a simple Python AVL tree found on the web [43], with 225 lines of code[3].

Figure 3 shows the reduction in number of distinct failing tests produced by normalization and reduction (vs. reduction only) for a set of 82 mutants [4] of AVLTree [31] (**RQ1**). Of the 228 mutants produced by MutPy [26], 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.

[3] All sizes non-comment, non-blank lines, by cloc [9].

Differences were statistically significant by paired Wilcoxon test with $p \leq 1 \times 10^{-154}$.

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" [7, 28]. 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 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) [7]. For the AVLTree example, the slippage rate for reduction is almost 30%, 10% *worse* than that for normalization. Mitigation approaches for slippage during reduction [28] should also apply to normalization.

The reduction in tests produced by normalization for mutant pairs was, as with single mutants, very large (Figure 3). 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[5]. 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 [7]. 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.

2.2 XML Parser

We also investigated **RQ1-3** for a Python XML parser with about 260 lines of code [12], 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 3). 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.

2.3 TSTL

As noted in Section **??**, 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]. Generating and reducing 100 tests for it required 1,090 seconds and produced 90

[4] All p-values given below are for the same recommended [5] statistical test.
[5] It is sometimes possible to detect both faults with a single test case.

[6] 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.

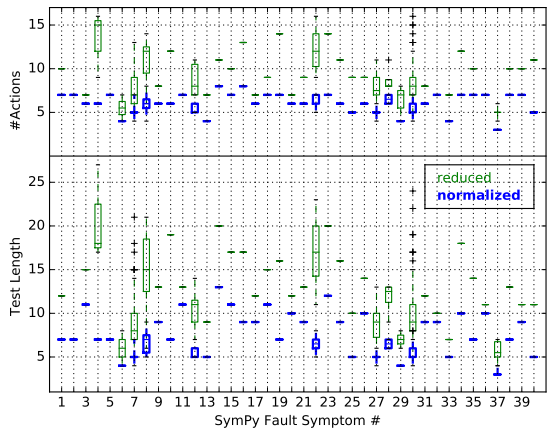

Figure 4: Effects of normalization on SymPy tests.

failures. Normalization and generalization increased total runtime to 3,690 seconds, and produced just 2 failures.

2.4 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, 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 4 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.

2.5 SortedContainers

SortedContainers [32] 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, $p \leq 1 \times 10^{-28}$).

2.6 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

```
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 5: Normalized and generalized NumPy test.

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 5. 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: 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.

2.7 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 6 and 7 show one crash-inducing test, after initial delta-debugging (from over 2,000 test steps) (Figure 6) and after normalization and generalization (Figure 7). 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 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 6) 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.

3 RELATED WORK

The tools described here are obviously inspired by delta-debugging [45] and the idea that tests should not contain extraneous parts not needed to cause test failure (or other behavior of interest [16, 17]). Delta-debugging and slicing [35] produce subsets of the original

```
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 6: Test with reduction-only for ArcPy.

```
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 7: Normalized and generalized ArcPy test.

test, but do not modify parts of the test to obtain further simplicity. Our work on normalization [19] extends this idea to rewrite tests into a "canonical" form. Normalization to a canonical form is in part motivated by the fuzzer taming [7] problem.

Zhang [47] proposed an approach to semantic test simplification that, like our approach, is able to modify, rather than simply remove, portions of a test. However, because the simplification operates directly over a fragment of the Java language, rather than using an abstraction of test actions, the set of rewrite operations is very simple: no new methods can be invoked, statements cannot be reordered, and no new values are used. 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 [39] performs some simple normalization as part of its test reduction for C code, and the peephole-rewrite scheme used in CReduce is also an inspiration for the approach taken by our normalizer. By writing a TSTL harness that is in the form of constructor calls to create an AST, TSTL can normalize/reduce hierarchically structured input data in ways similar to CReduce and Hierarchical Delta Debugging [36].

The most closely related work to test generalization is Pike's SmartCheck [38]. SmartCheck works with algebraic data in Haskell, and is an alternative approach to reduction and generalization. The only other work we are aware of that is similar to generalization concerns causality in model checking counterexamples [21, 25, 33].

4 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 [37] in real-world debugging, few practicing testers doubt the value of being provided with a minimal number of minimally-sized failing tests [18, 34, 40].

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 [8, 14]. TSTL-based normalization and generalization are currently available in a well-tested Python version [23, 24]; 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 [41] or C [44] tests are non-trivial, but not obviously overwhelming [39]. Less effective methods than ours might still aid debugging and assist fuzzer taming [7]. 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 [24] 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. 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.
[5] 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.
[6] 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.
[7] 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.
[8] 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.
[9] A. Danial. CLOC: Count lines of code. https://github.com/AlDanial/cloc.
[10] 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.
[11] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computing*, 3(1):69–115, 1987.
[12] erezbibi. https://pypi.python.org/pypi/my_xml/0.1.1.
[13] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.
[14] 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.
[15] 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.
[16] 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.
[17] 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.
[18] 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.
[19] A. Groce, J. Holmes, and K. Kellar. One test to rule them all. In *International Symposium on Software Testing and Analysis*, page accepted for publication, 2017.
[20] 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.
[21] A. Groce and D. Kroening. Making the most of BMC counterexamples. *Electron. Notes Theor. Comput. Sci.*, 119(2):67–81, Mar. 2005.
[22] A. Groce and J. Pinto. A little language for testing. In *NASA Formal Methods Symposium*, pages 204–218, 2015.
[23] 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.
[24] A. Groce, J. Pinto, P. Azimi, P. Mittal, J. Holmes, and K. Kellar. TSTL: the template scripting testing language. https://github.com/agroce/tstl.
[25] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.
[26] K. Halas. MutPy 0.4.0. https://pypi.python.org/pypi/MutPy/0.4.0.
[27] R. Hamlet. When only random testing will do. In *International Workshop on Random Testing*, pages 1–9, 2006.
[28] J. Holmes, A. Groce, and A. Alipour. Mitigating (and exploiting) test reduction slippage. In *Workshop on Automated Software Testing*, 2016.
[29] 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.
[30] G. Huet and D. C. Oppen. Equations and rewrite rules: A survey. Technical report, Stanford, CA, USA, 1980.
[31] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.
[32] G. Jenks. SortedContainers introduction. http://www.grantjenks.com/docs/sortedcontainers/introduction.html.
[33] 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.
[34] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *International Symposium on Software Reliability Engineering*, pages 267–276, 2005.
[35] 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.
[36] G. Misherghi and Z. Su. Hdd: hierarchical delta debugging. In *International Conference on Software engineering*, pages 142–151, 2006.
[37] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis*, pages 199–209, 2011.
[38] L. Pike. SmartCheck: automatic and efficient counterexample reduction and generalization. In *ACM SIGPLAN Symposium on Haskell*, pages 53–64, 2014.
[39] 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.
[40] J. Ruderman. Bug 329066 - Lithium, a testcase reduction tool (delta debugger). https://bugzilla.mozilla.org/show_bug.cgi?id=329066, 2006.
[41] J. Ruderman. Introducing jsfunfuzz, 2007. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.
[42] J. Ruderman. Releasing jsfunfuzz and DOMFuzz. https://www.squarefree.com/2015/07/28/releasing-jsfunfuzz-and-domfuzz/, 2015.
[43] user1689822. python AVL tree insertion. http://stackoverflow.com/questions/12537986/python-avl-tree-insertion.
[44] 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.
[45] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.
[46] 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.
[47] S. Zhang. Practical semantic test simplification. In *International Conference on Software Engineering*, pages 1173–1176, 2013.