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

CSCS CONCEPTS
• Software and its engineering → Software testing and debugging;

KEYWORDS
test case reduction, semantic simplification, fuzzer taming

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference format:

1 Approaches that optimize for short tests [9, 18] may not require reduction, but random testing [7, 33], model-checking [20], and symbolic execution [51] can all benefit.
2 No single component of a 1-minimal/delta-debugged [50] test can be removed without causing the test to pass.
balanced() of tests for an AVL tree class, in the usually API-call sequences, but also grammar-based tests, and a TSTL [28, 29, 35] is a language for defining the structure of tests. A test is just an ordered sequence of actions, which is equivalent to a finite set of numbered steps. Because our normalization is defined in terms of actions, steps, and pools, it is language agnostic. Terms of actions, steps, and pools, it is language agnostic. However, it would be easy to assume all small numeric values in normalized tests are accidental. Generalization allows users to distinguish actually essential small values.

Normalization is not yet a complete solution to the problem of identifying distinct faults (e.g., our algorithms do not apply to complex custom test generators such as Csmith [49] or jsfunfuzz [46]), but it is often highly effective. Running 100,000 tests (of length 100) on the faulty AVL tree produces 860 failing tests with no duplicates. Normalizing these reduces the number of distinct failing tests to just 22. Ideally all failures due to the same fault in the SUT (Software Under Test) would normalize to a single, representative test. We aim to approximate such a canonical form for faults. Figure 5, in Section 2.1.2, shows an AVL tree test for this fault that normalizes differently. In experiments with 82 AVL tree faults, the mean number of distinct failures after normalization for 1,000 tests was just 3.1 (with median 2).

The contributions of this paper are 1) the idea of test normalization and generalization as key steps towards a goal of "one test to rule them all" (per fault), 2) algorithms for normalization and generalization that make use of the abstract interface for testing provided by the TSTL [28–30, 35] domain-specific language (DSL) [17], and 3) experimental results showing the value of these ideas. Normalization frequently provides significant additional test length reduction for complex SUTs, and can reduce the set of failures to be examined by more than an order of magnitude. Normalization and generalization have also been useful in understanding complicated tests for a variety of real-world software systems.

2 FORMAL DEFINITIONS

TSTL [28, 29, 35] is a language for defining the structure of tests (usually API-call sequences, but also grammar-based tests), and a set of tools for use in generating, manipulating, and understanding those tests. Figure 3 shows a simplified portion of a TSTL definition (called a harness [23, 26]) of tests for an AVL tree class, in the latest syntax for TSTL. Given a harness like the one in Figure 3, TSTL compiles it into a class file defining an interface for testing that provides features such as querying the set of available testing actions, restarting a test, replaying a test, collecting code coverage data, and so forth. The TSTL release [30] provides testing tools that use the interface for testing and debugging.

The key point for our purposes is merely that a TSTL test harness defines a set of pools whose instances hold values produced and used during testing [4] (a common approach to defining API-testing sequences) and a finite set of actions that are possible during testing, typically API calls and assignments to pool instances. In this example, there are two pools, one named int and one named avl. There are four instances of the int pool, which means that a test in progress can store up to 4 ints 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 to any integer in the range 1-20 inclusive, setting the value of an avl to a newly constructed AVL tree, and calling insert, delete, find and inorder with chosen pool instances. Figure 1 in the introduction shows three valid tests produced by running a random test generator on the TSTL-compiled interface produced by this definition. TSTL handles ensuring that tests are well-formed. No pool instance (such as avl11) can appear in an action until it has been assigned a value. No pool instance that has been assigned a value can be assigned a different value until it has been used in an action, to avoid degenerate sequences such as int3 = 10 followed by int3 = 4. Each action in a test is called a “step” — the first step of the first test in Figure 1 is storing a new AVL tree in avl0, for example. A test is just an ordered sequence of actions, which is equivalent to a set of numbered steps. Because our normalization is defined in terms of actions, steps, and pools, it is language agnostic.

The definition of pools and actions in TSTL defines a total order on all actions. First, actions are ordered by their position in the
where each underlying fault is uniquely represented by a single test. A test normalization algorithm has a simple goal: we ideally aim at approximating the goal, by normalizing each rewrite rule to rewrite f... as difficult as automatic fault localization and repair, and such are preserved. This is not clearly useful. A passing test can be normalized, though without some more interesting predicate, such as preservation of coverage, this is not clearly useful.

normalizing each rule ought to be unlikely to change the underlying cause for test failure. To that end, the rules for normalization always either change at most one action (possibly in multiple steps, but in a uniform way) or make no changes to the actions performed, only to the pool instances used or the positions of actions in the test. We cannot guarantee normalization does not change the underlying fault in a test; however, the limited scope of rewrites should minimize the chance of fault change (known as "slippage" [8, 34]).

2.1 Normalization

A test normalization algorithm has a simple goal: we ideally aim to produce a function f : t → t (a function that takes a test and returns a test) such that: (1) if t fails, f(t) fails; (2) if t1 and t2 fail due to the same fault, f(t1) = f(t2); (3) if t1 and t2 fail due to different faults, f(t1) ≠ f(t2).

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

In the setting of TSTL, where test actions have a defined total order, two simple principles can be applied to produce useful rewrite rules. All rewrites reduce the sum of the indices of the actions in the test, make the test’s actions more ordered by index, or reduce test length. This guarantees that the rewrites are strongly normalizing. The second principle that determines the rewrite rules is that each rule ought to be unlikely to change the underlying cause for test failure. To that end, the rules for normalization always either change at most one action (possibly in multiple steps, but in a uniform way) or make no changes to the actions performed, only to the pool instances used or the positions of actions in the test. We cannot guarantee normalization does not change the underlying fault in a test; however, the limited scope of rewrites should minimize the chance of fault change (known as "slippage" [8, 34]).

2.1.1 Rewrite Rules. Figure 4 shows the rewrite rules used in TSTL normalization. The notation in the rules is relatively simple. A step is an action paired with an number indicating its position in a test, where the first action is step 0, etc.; e.g., (2 : a) indicates the third step of the test is action a (indexing is from 0). Δ(t, t′) is the set of all steps in t such that t(i) ≠ t′(i).

For ordering, we say that a < b iff the index of action a is lower than that of action b. We compare steps with < by comparing their actions — (i : a) < (j : b) iff a < b. For a set or sequence of actions or steps, we define the min of the set to be the lowest indexed action in the set, and use these to compare sets: s1 < s2 iff min(s1) < min(s2). For pool instances, p < p′ if and only if p’s pool index is lower than that for p′ and p and p′ are from the same pool. For example, int3 < int4, but avl0 < int4.

The term 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 pool instances. t[x \leftrightarrow y] is similar, except that x and y are swapped. Term t[i,j][x \rightarrow y] is the same as t[x \leftrightarrow y], except that the replacement is only applied between steps i and j, inclusive. Finally, t[i\rightarrow k](x) denotes t with all steps containing x that are before step i moved to step i, preserving their previous order, shifting steps at i and after i to make room for the moved steps, again preserving order.

2.1.2 Normalization Algorithm. These rules alone do not determine a complete normalization method; it is also necessary to determine the order in which they are applied. The order in our default implementation is the order above, with the modification that in practice the ReplacePool and ReplaceMovePool rewrites are checked in the same loop (e.g., for every possible replacement of a pool instance, both rules are checked, in the order given above). The order was determined after considerable, but not exhaustive, trial and error, and aims to apply more general, but less expensive, rules first. 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 [22]. Notice that after applying each rewrite rule, we perform delete-debugging on the new base test, since often a rewrite makes other steps irrelevant.

Normalization can be generalized to apply to any predicate over tests, not just failure [22]; a passing test can be normalized, though without some more interesting predicate, such as preservation of coverage, this is not clearly useful.
Algorithm 1

1: modified = True
2: while modified do
3:    modified = False
4:    for i ∈ C(t) do
5:      if pred([i]) then
6:        modified = True
7:        t += ddimmin([i])
8:      break (exit for loop)
9:    end if
10: end for
11: end while
12: return t

Original test:

0: int0 = 10
1: int2 = 7
2: avl0 = avl.AVLTree()
3: avl0.insert(int2)
4: avl0.insert(int0)
5: int1 = 1
6: int3 = 1
7: int3 = 15
8: avl0.insert(int3)
9: avl0.delete(int3)

Normalized:

0: int0 = 1
1: int2 = 2
2: avl0 = avl.AVLTree()
3: avl0.insert(int0)
4: avl0.insert(int2)
5: int2 = 3
6: int0 = 4
7: int0 = 3
8: avl0.insert(int0)
9: avl0.delete(int0)

Normalization Steps:

- SimplifyAll: t ::= ([a] ⇒ [a]) where a′ < a
- ReplacePool: t ::= ([i], p) ⇒ [p'] where p < p′ and 0 ≤ i < |t|
- ReplaceMovePool: t ::= ([i], p) ⇒ [p'] where p < p′ and 0 ≤ i < |t|
- SimplifySingle: t ::= ([a], [a']) where a′ < a
- SwapPool: t ::= ([i], [p]) ⇒ [p'] where p < p′ and 0 ≤ i < |t|
- ReduceAction: t ::= ([a]) ⇒ ([a']) where ddimmin([a′]) < |t|
- Reducible: t ::= ([a]), p ⇒ ([b]) where p < p′ and b < a

The worst-case complexity of normalization can be given an upper bound by recalling our rule that each rewrite must lower the number of possible rewrites by at least one. This means if there are n possible rewrites of a test, there can be at most n predicate checks for the current test, then at most n – 1 checks for the rewritten test, and so on, for a total of n times predicate checks, where n is the number of possible rewrites of the test t being normalized: n = |C(t)|. Test execution to check the predicate can be assumed to have a constant cost since the length of the test does not usually change by more than a few steps during normalization.

Figure 4: Rewrite rules for normalization.

Figure 5: An example of normalization steps.

The simplest optimization is to improve on the constant ordering of rewrite rules. Once a rule fails to produce a candidate that satisfies pred, that rule should be moved to the end of the ordering of rewrites, since once a rule fails once to produce any valid rewrites, it frequently produces no further reductions. This simple change typically halves the time required for normalization. When test execution is very expensive, the set of candidate tests can be further restricted: limiting action replacements to cases where Levenshtein [41] distance (text edit distance) between the code for actions is bounded to a small value was effective in reducing runtime, and often had little impact on final results. A further useful optimization when normalizing large numbers of tests is to cache results across tests (since the algorithm is deterministic for a given pred).

\[^{[1]}\text{The details of how these bounds are determined, in that pool changes are also action changes, are not critical, and a more detailed analysis is somewhat involved, and beyond the scope of this paper; we note that like delta-debugging, the worst-case complexity is seldom observed, and offer some further optimizations.}\]
For very large numbers of tests, this is the most important optimization. For systems with expensive replay, delta-debugging of each new base test can be omitted: the ReduceAction rewrite will eventually remove extraneous steps. It is also trivial to parallelize normalization by checking the predicate over multiple candidates at once. As soon as a candidate satisfies the predicate, a parallel implementation can proceed with that candidate as the new base, making the algorithm nondeterministic (in practice, we suspect the same final result will usually appear), or the algorithm can wait for all earlier-in-sequence candidates to be checked, and only proceed when no candidate that would be checked earlier is in the queue.

2.2 Generalization

The core idea of generalization is to use methods similar to those involved in normalization to provide a user with information about changeable aspects of a test. Some values and orderings of steps in a test are essential to the failure: when changed, they cause the test to no longer fail. Many others, however, are accidental — any concrete test has to choose some worst case, no swaps are possible. The complexity of checking for when no candidate that would be checked earlier is in the queue.

Reducing Action

\begin{algorithm}
\begin{algorithmic}[1]
\caption{Basic algorithm for generalization}
\State \textbf{swap} = 0
\State \textbf{replace} = 0
  \For {\textbf{(i, a)} \in t}
    \If {\textbf{a} \neq \textbf{a}'}
      \State \textbf{replace} = \textbf{replace} + 1
    \EndIf
  \EndFor
  \For {i < j < |t| - 1 \land (j : b) > (i : a)}
    \If {\text{pred}((i : a) \rightarrow ((i, a') \rightarrow ((j, b) \rightarrow (i : a)) \rightarrow \text{true})}
      \State \text{swap} = \text{swap} + 1
    \EndIf
  \EndFor
\State \text{return} (\text{swap}, \text{replace})
\end{algorithmic}
\end{algorithm}

This algorithm collects all steps that can be replaced with other actions or swapped with other steps, and returns the set to be reported to the user. This version assumes the test has already been normalized, but can be extended to any test by removing the restrictions that \textbf{a} > \textbf{a}' and \textbf{j} > \textbf{i}.

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

\begin{algorithm}
\caption{Generalization Algorithm.}
\begin{algorithmic}[1]
\State \textbf{swap} = 0
\State \textbf{replace} = 0
\For {\textbf{(i, a)} \in t}
  \If {\textbf{a} \neq \textbf{a}'}
    \State \textbf{replace} = \textbf{replace} + 1
  \EndIf
\EndFor
\For {i < j < |t| - 1 \land (j : b) > (i : a)}
  \If {\text{pred}((i : a) \rightarrow ((i, a') \rightarrow ((j, b) \rightarrow (i : a)) \rightarrow \text{true})}
    \State \textbf{swap} = \textbf{swap} + 1
  \EndIf
\EndFor
\State \textbf{return} (\textbf{swap}, \textbf{replace})
\end{algorithmic}
\end{algorithm}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6.png}
\caption{Generalized test for TSTL itself, showing fresh value generalization.}
\end{figure}

swaps in the worst case is quadratic in \textbf{k}. In practice, most actions are not enabled at most steps, and most actions in a test are not the lowest-indexed action. Basic generalization is trivial to parallelize.

2.2.2 Fresh Values and Misleading Tests. A side-effect of delta-debugging and normalization is reduction of the number of variables in a test. While usually helpful, this can sometimes result in misleading tests. In a stateful system, putting the system into a bad state may require building a complex object. Once system state is corrupted, however, the complex object is irrelevant, and its appearance in the call leading to failure can be misleading. In previous work at NASA, we observed that sometimes a delta-debugged file system test [24, 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 have failed. We therefore propose a more aggressive generalization: replacing a pool instance use with a fresh value.

Consider the test in Figure 6, produced by a TSTL harness for TSTL itself\footnote{Since Python TSTL provides a Python API, that API can be used as the SUT in testing.}. The problem involves an invalid cache, produced by normalizing a test with only one action. Without fresh value generalization, it appears that the failure is due to normalizing test0 again. The annotation after step 6 lets us see that the failure will take place even for a fresh test. Without this generalization, the state of test may appear to be important, not the system state. Formalizing this generalization requires additional notation. \text{U}(a) is the set of pool instances used in the action \textbf{a} — pool instances that appear in the action, but not on the left-hand side of an assignment. \text{I}(a, p) is a predicate that is true iff action \textbf{a} stores a new value in pool instance \textbf{p}. Finally, \text{I}(a, p) \oplus \text{I}(a, p) denotes test \textbf{a} with the action \textbf{a} inserted at step \textbf{i} and each step from \textbf{i} onwards moved to a position one higher.

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

3 EXPERIMENTAL RESULTS

This section presents some initial results of applying normalization and generalization, and comparing the results to delta-debugging (which we refer to as reduction) alone. All tests were generated using pure random testing, based on TSTL harnesses developed previously, all included in the TSTL release [30]. We also tested the Python interface to Z3 [13], but did not find faults thus far; normalization did help produce more comprehensible and uniform Z3 quick tests [22].

These experiments are intended to establish the basic potential value of the techniques, and provide, for seven Python libraries ranging from small to large and complex, some initial data on research questions. 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? We also examined the question of whether normalization and generalization provide substantial benefits in understanding complex tests, in a qualitative way, by examining tests for some of the larger SUTs studied. 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). The benefits for human understanding would require a human study to fully evaluate.

Our experimental subjects and results can be divided into three parts. First, we studied simple programs with small failing tests, in order to use mutation analysis to thoroughly investigate RQ1-3, especially in the context of tests without a good fault signature, where normalization is most needed to reduce failures to examine. Second, we studied larger and more realistic programs with a large number of lengthy, complex failures, largely to provide more information on RQ4 (additional reduction beyond that provided by delta-debugging). Finally, we examined a much smaller number of failures for subjects where each reduction or normalization required an extremely long time to complete, and understanding individual tests is a difficult task.

3.1 Mutation-Based Experiments

Our small subjects are a simple Python AVL tree found on the web [48], with 225 lines of code6 and a simple XML parser with about 260 lines of code [15]. For both subjects, we produced mutants using the MutPy tool [32], then filtered the set to contain only mutants that produced at least 1 failing test in 1,000 tests. MutPy generates mutants using both the standard “core” operators common to many tools [5] and a few Python-specific operators. We then used the filtered mutants to generate higher-order mutants (“pairs”) composed of two mutants, such that each of the mutants could be detected in isolation: that is, there existed at least one failing test such that fixing the other mutant left the test still failing. For AVL, there were 82 failing mutants (out of 228 total), from which we sampled 364 pairs, restricted to mutants modifying different source lines; 364 random samples were required to sample each mutant at least twice. Of these, 238 pairs had independently detectable faults. For XML, only 5 of 357 mutants generated were detectable due to a weak specification. There were thus only 9 pairs with independently detectable faults, and we evaluated all of these.

3.1.1 RQ1: Reduction in Number of Failures. Figure 7 shows the reduction in number of distinct failing tests produced by reduction alone vs. reduction and normalization, for AVLTree and XML mutants; note the log-scale y-axis. All differences are statistically significant by paired Wilcoxon test [6], at \(p < 0.05\). For 38 of the 82 AVL single mutants, normalization was perfect (1 failure); for XML, only one mutant was perfectly normalized. Normalization was perfect (1 or 2 failures) for 115 of the 238 AVLTree pairs, but

\[\text{Algorithm 3: Basic algorithm for fresh object generalization}\]

1. fresh = ∅
2. for \((i, a) \in t\) do
3. for \(p \in U(a)\) do
4. for \(a' \in I(a', p)\) do
5. if pred(\(+a', i\)) then
6. fresh = fresh ∪ \((i, a')\)
7. end if
8. end for
9. end for
10. end for
11. return fresh


![Figure 7: Effects of normalization on AVLTree and XML parser mutants.](image-url)
none of the XML pairs. Even when the result was not perfect, improvement was usually quite dramatic, from about 500 failures on average per mutant or mutant pair for AVL and about 100 for XML to less than 5 for AVL and less than 10 for XML.

A second way to examine reduction in a multi-fault setting is to consider the mean number of tests a user must examine before seeing all faults [8]. For the AVL mutant pairs, a user must examine almost 20 tests (on average) before encountering at least one instance of both faults. With normalization, this drops to a mean of just 2 tests. The difference is significant with \( p \leq 1.4 \times 10^{-6} \). The difference in number of tests before encountering both faults in XML pairs was not statistically significant (for these 5 faults, failure rates are very similar, so hitting both is easy no matter how many failures there are; however, satisfying oneself that all faults have been seen is much harder with a very large number of tests to examine).

3.1.2 RQ2: Slippage via Normalization. Mutants also provided a way to evaluate the danger of normalization losing faults, RQ2. Faults can be lost when normalization changes a test failing due to one fault into a test failing due to a different fault; a problem known as “slippage” [8, 34]. The AVL and XML testers pose an interesting slippage challenge, as the failure signatures are not useful, and the tests for faults are likely to be very close to each other in the combinatorial space, due to the simplicity of the tested interfaces. Out of the 238 AVL mutant pairs, normalization produced tests exposing both faults for 80.7% of pairs (19.3% slippage at the suite level). Interestingly, for 4 AVLTree pairs 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. For XML pairs, the slippage rate was 12.5% (of the 9 pairs, only 1 lost a fault, and in that case the faults were both syntactically and semantically very similar: within 1 line and with similar effects).

Recall that of our 364 AVL mutant pairs, only 238 could be independently detected; in most cases this was due to reduction itself introducing slippage (in a few cases, one fault completely masks another; e.g., if the constructor always fails, insert failures cannot be detected). Slippage due to reduction itself is very rare for some SUTs but common for others (up to 23% for Mozilla’s JavaScript engine) [8]. For the AVLTree example, the slippage rate for reduction is almost 30%, 10% worse than that for normalization; for XML, however, reduction alone never caused slippage. Mitigation approaches for slippage during reduction [34] should also apply to normalization.

3.1.3 RQ3: Normalization Runtime. The mean cost to reduce an AVLTree test was 0.05 seconds; the mean cost for normalization was 0.38 seconds. For XML, the mean cost for reduction was 12.6 seconds vs. 120.8 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. 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 AVL mutant experiments was only 145,780, for a mean of only 2.4 non-cache-hit rewrites of each test. For XML, there were 3,829 cache hits over a total of 3,929 normalizations.

3.2 Experiments Using Real Faults

3.2.1 XML Parser. We also investigated one real fault, triggered by the empty tag (\(<>\)), and one seeded fault, triggered when adding two nodes with the same name, for the XML Parser. A comment in the code indicates the seeded fault is realistic, and probably 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.

3.2.2 TSTL. As noted in Section 2.2.2, TSTL is used to test TSTL’s own API interface (the code is about 2,700 LOC; a compiled SUT is often > 30KLOC). We discovered one fault while testing the latest version of TSTL, the cache-related problem shown in Figure 8. Generating and reducing 100 tests for it required 1,090 seconds and produced 90 failures. Normalization and generalization increased total runtime to 3,690 seconds, but only 2 failures.

3.2.3 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, has over 25,000 commits to date, and consists of more than 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 faults detected from 500 to less than 5 for AVL and less than 10 for XML.

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.
Table 1: Summary of results for SymPy and SortedContainers.

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Reduced</th>
<th>Normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Tests</td>
<td>Length</td>
<td>#Acts</td>
</tr>
<tr>
<td>SymPy</td>
<td>300</td>
<td>44.7</td>
<td>50</td>
</tr>
<tr>
<td>SortedContainers</td>
<td>168</td>
<td>86.8</td>
<td>95</td>
</tr>
</tbody>
</table>

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.

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 \( p \leq 1 \times 10^{-75} \). Normalization made it possible to completely ignore a large number of SymPy functions for debugging purposes.

The unreduced and reduced tests included all 50 SymPy functions tested. The normalized tests, however, included only 32 of these, and enabled us to ignore such complex code as trigonometric expansion and simplification, power expansion, logarithmic combination, and even generalized expansion. Figure 8 graphically shows the impact of normalization on length and number of functions covered by reduced tests. The lower part of the figure shows changes in test length, and the upper part shows, for the same fault, the change in number of different actions. The green boxplots show reduced tests, while the emphasized blue boxplots show normalized tests. It is clear that normalization not only has a significant effect on average, but has a large benefit for most individual faults. The reduction in tests to examine is also shown, indirectly, by the fact that the “boxes” for normalized data are often simply lines because the tests are similar or identical.

3.2.4 SortedContainers. SortedContainers [37] is a popular 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 two other 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, \( p \leq 1 \times 10^{-28} \)).

Table 1 summarizes mean results for SymPy and SortedContainers, our primary results for large numbers of failing tests for real faults of complex programs where normalization is perhaps as useful for additional reduction beyond delta-debugging as it is for reducing the number of tests to examine.

3.2.5 NumPy. Our final two case studies provide little information on RQ1 and RQ2: for these SUTs, failure rates are low enough or test reduction runtimes high enough that each failure is usually dealt with one-by-one. However, the value of normalization and generalization for further reduction (RQ4) and aiding in understanding tests is effectively shown by these complex programs. They also provide results for RQ3 when even test reduction is expensive. NumPy [1] is a widely used Python library that supports large, multi-dimensional matrices and provides a huge library of mathematical functions. The SciPy library for scientific computing builds on NumPy. Developing tests for NumPy is challenging, because none of the authors are experts in numeric computation, and the specification of correct behavior is often somewhat subtle. As an example, consider the test in Figure 9. Prior to normalization, understanding why the test leads to a violation of self-equality for an array is difficult: the reduced-only test has 42 steps and includes not only array multiplication and addition, but subtraction, array copying, reshaping, flattening, filtering by unique elements, and raveling. After normalization, it is much clearer what is happening: 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 3 seconds, and the time for normalization is between 712 and 774 seconds. Generalization takes between 52 and 59 seconds in these cases. The exception was a test of 45,206 steps (!) leading to a memory exhaustion error and crash. This was reduced (over nearly a day) to a test with 10 steps, which then normalized (in only 2 hours) to a test with 8 steps. The normalized test involved no operations other than array initialization, array flattening, and array addition. The reduced test involved larger array dimensions, array multiplication, and array subtraction, as well.

3.2.6 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 [35].

Figures 10 and 11 show one crash-inducing test, after initial delta-debugging (from over 2,000 test steps) (Figure 10) and after normalization and generalization (Figure 11). In this setting normalization has contributed a significant amount of additional reduction over delta-debugging. For the crash fault shown in this paper, normalization reduced the length from 19 steps to 11 steps. For three other crashes, normalization reduced the tests from 18 to 14 steps, from 27 to 20 steps, and from 20 to 16 steps. One crash fault only reduced from 10 steps to 9 steps, but the omission was informative.

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

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

4 RELATED WORK

This work builds on the idea behind delta-debugging [50]: tests should not contain extraneous information that is not needed to reproduce failure (or some other behavior [21, 22]). Delta-debugging and slicing [40] are limited, generally, to producing subsets of the original test, not modifying parts of the test to obtain further simplicity. We extend this concept by also allowing modification or re-ordering, which also allows further length reduction.
Normalization is in part motivated by the fuzzer taming [8] problem: determining how many distinct faults are present in a large set of failing tests. This is a key problem in practical application of automated testing. Previous work on fuzzer taming [8] used delta-debugging to reduce some tests to syntactic duplicates.

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

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

The most closely related work to our generalization efforts is Pike’s SmartCheck [43]. SmartCheck targets algebraic data in Haskell, and offers an interesting alternative approach to reduction and generalization. Test generalization is also akin to dynamic invariant generation, in that it informs the user of invariants over a series of test executions [16]. Fraser and Zeller proposed generalizing unit tests [19], but without the goal of preserving a predicate such as test failure. The only other work we are aware of that is similar to generalization concerns essential and accidental aspects of model checking counterexamples [27, 31, 38].

5 CONCLUSIONS AND FUTURE WORK

This paper introduces test normalization and generalization. The methods presented are significant steps towards a difficult goal: providing users of automated testing with a single test, as short and simple as possible, for each underlying fault in the SUT, and annotations describing the general conditions under which the fault manifests as failure. Normalization approaches this ideal by rewriting numerous distinct failing tests into a smaller, often minimal, set of simpler tests. Generalization uses automated experiments to distinguish essential and accidental elements of a test. In our experiments, normalization reduced the number of failures to examine by well over an order of magnitude, often to the ideal of one per fault, and reduced the length of tests beyond what is possible with delta-debugging alone. While there is doubt about the utility of automatic fault localization [42] in real-world debugging, few practicing testers doubt the value of being provided with a minimal number of minimally-sized failing tests [24, 39, 45].

The algorithms for normalization and generalization depend only on a (possibly somewhat arbitrary) total order over test actions and an abstract form for tests, suitable for term rewriting. Our approach is therefore likely applicable to any source language and many different test generation methods, including those that already produce short tests [9, 18]. TSTL-based normalization and generalization are currently available in a well-tested Python version [29, 30]; there is also a beta version, with more limited normalization, for Java. The goal of normalization and generalization can also be pursued in settings other than API sequence or string grammar testing. The difficulties of defining a normal form for JavaScript [46] or C [49] tests are non-trivial, but not obviously overwhelming [44]. Less effective methods than ours might still aid debugging and assist fuzzer taming [8]. Simple generalization (e.g., is this numeric constant essential, can these two statements be swapped?) and a limited form of fresh value generalization should be easy to apply, even for complex programming language tool tests.

The working version of TSTL [30] 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.”

Acknowledgements: The authors would like to thank John Reghr, and Lee Pike for their comments. The current implementation of normalization is considerably faster than reported in this paper thanks to a suggestion of David R. MacIver with respect to the ReduceAction rule.

Figure 11: Normalized and generalized ArcPy test.
REFERENCES

[24] K. Halas. MurPy 0.4.0. h/t_p://pypi.python.org/pypi/MurPy/0.4.0.