Normalizing and Generalizing Test Cases

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

***Abstract*—Test case reduction has long been seen as essential to effective automated testing. However, test case reduction simply reduces the length of a test case. It does not attempt to produce *semantic* simplicity. In this paper, we present two algorithms. The first provides simplification for test-cases by attempting to rewrite them into a normal form. Test cases converted into normal form are sometimes shorter than the test cases produced by delta-debugging, but the primary feature of normalization is that it often converts many test cases that expose the same fault into a single test, reducing the number of test cases that a reader must examine, and partially addressing the "fuzzer taming" problem of determining how many faults are present in a set of failing test cases. 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. Together, normalization plus generalization allows a user to replace reading a set of test cases that vary in unimportant ways with reading one annotated test case. The algorithms for normalization and generalization make use of the features of a recently introduced DSL for testing, TSTL. Experimental data shows that normalization can reduce (by over an order of magnitude) the number of different test cases to be examined, without loss of fault detection. We show how normalization plus generalization aids understanding of test cases for a set of example programs, including ESRI's widely used arcpy GIS (Geographic Information System) library.**

I. Introduction

It has long been understood that effective automated testing requires test case reduction [1], [2], [3] to produce test cases that remove irrelevant operations. In fact, test case reduction is now standard practice in industrial testing tools such as Mozilla's `jsfunfuzz`. However, simply reducing the length of a test case does not produce true semantic minimality. There may be many 1-minimal test cases that present different variations of a single fault. In many cases, reading more than one of these test cases provides no useful additional information on the cause of failure.

Consider the three test cases shown in Figure 1. These test cases are obviously very similar, and in fact all lead to a violation of the property that an AVL tree must always be nearly balanced, due to a missing call to `rebalance` in `delete` in a Python implementation of AVL trees. However, the test cases are syntactically very different, and a testing system that collects failing test cases will produce three tests for a user to examine. While there are methods for attempting to determine which test cases represent distinct faults [4], the ideal solution is arguably to rewrite all three of these test cases into a single, normal form that preserves the structure of the failure while removing such accidental aspects of each test case as the particular integer values and variables used, and the ordering of assignments and insertions.

Figure 2 shows the result of applying our *test case normalization* method to these three tests, and then applying our *test case generalization* approach to the normalized test case. First, we note that all three test cases normalize to the same test case. This test case, in addition to the 10 steps required to produce the failure, includes comments indicating what about the test case can be changed while still failing in the same way. For instance, the value 1 assigned to `int0` in step 0 is not essential. It could be changed to any value in the range 5-20 (the total set of values allowed by the test generator) without changing the final result. The same is true of the assignment of 3 to `int1`. Similarly, the ordering of many steps in the test case is not important. Finally, step 9 is annotated to show that instead of using the existing value of `int1` (4), a fresh assignment could be inserted before the `delete` call, setting `int1` to 3 instead. These possible changes are not meant to be combined — the annotation claims only that changing these aspects of the test case one at a time will preserve failure.

II. A Brief Introduction to TSTL

TSTL [5], [6] is a language for defining the structure of test cases, and a set of tools for use in generating, manipulating, and understanding those test cases. Figure 3 shows a simplified portion of a TSTL definition of tests of an AVL tree class, in the latest syntax for TSTL (which differs slightly from that in the cited papers introducing TSTL). TSTL provides numerous features not shown in this small example, including automatic differential testing, complex logging, support for complex guards, and use of pre- and post- values. Given a harness like the one in Figure 3, TSTL compiles it into a class file defining an interface for testing that provides features such as querying the set of available testing actions, restarting a test, replaying a test, collecting code coverage data, and so forth. The TSTL release (https://github.com/agroce/tstl) also provides some testing tools that make use of this interface to generate tests and provide other debugging capabilities.

The key point for our purposes is merely that a TSTL test harness defines a set of *pools* that hold values produced and used during testing [7] (a common approach to defining API-testing sequences) and a set of actions that are possible during testing, typically API calls and assignments to pool values. In this example, there are two pools, one named `int` and one named `avl`. There are four instances of the `int` pool, which means that a test in progress can store up to 4 `int`s at one time (in variables named `int0`, `int1`, `int2`, and `int3`), and

Test case #1

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

Test case #2

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

Test case #3

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

Fig. 1. Three randomly generated test cases for the same fault.

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

Fig. 2. Normalization and generalization for all three test cases. Lines beginning with # are comments in Python, here used to annotate the test case.

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

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

```
avl1 = avl.AVLTree()
int3 = 10
int1 = 11
avl1.insert(int1)
int1 = 1
avl1.insert(int3)
avl1.insert(int1)
int3 = 9
avl1.insert(int3)
int2 = 11
avl1.delete(int2)
```

Fig. 4. An example TSTL-produced test

three instances of the `avl` pool. The actions defined here are setting the value of an `int` pool to any integer in the range 1-20 inclusive, setting the value of an `avl` pool to a newly constructed AVL tree, and calling an AVL tree's `insert`, `delete`, `find` and `inorder` methods. Figure 4 shows a valid test case produced by running a random test generator on the TSTL-compiled interface produced by this definition. TSTL handles ensuring that tests are well-formed: for example, no pool instance (such as `avl1` can appear in an action until it has been assigned a value), and no pool instance that has been assigned a value can be assigned a different value until it has been used in an action, to avoid degenerate sequences such as `int3 = 10` followed by `int3 = 4`. Each action in a test case is called a "step" — the first step of the example test is storing a new AVL tree in `avl1`, for example.

The definition of pools and actions in TSTL defines a *total order* on all actions. First, actions are ordered by their position in the definition file. All `insert` actions are therefore before all `delete` actions, and all `delete` actions are before find actions. Of course, one line of TSTL defines many actions, because of the choices for pools. For example, the line `<avl>.insert(<int>)` defines 12 actions, one for each choice of `avl` and `int` pool instance. These are ordered lexically, in the obvious way (`avl0` preceeds `avl1`, etc.). Value ranges such as in the `int` initialization, are also ordered in the natural way, with lower values first. Given this total order, each action can be assigned a unique index, from 0 up to 1 less than the total number of actions. Initially, this ordering (and numbering) for each action was intended to allow for a kind of Goedel-numbering of tests, for proving certain mathematical properties [7]. However, it also allows us to concisely define a practical method for normalizing and generalizing test cases.

III. Normalization Algorithm

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

1) If t fails, $f(t)$ fails.
2) If t and t' fail due to the same fault, $f(t) = f(t')$.
3) If t and t' fail due to different faults, $f(t) \neq f(t')$.

Such a function would define a true *canonical form* for test cases, where each underlying fault is uniquely represented by a single test case. In general, it seems clear that defining such an f is (at least) as difficult as automatic fault localization and repair. Therefore, we aim at approximating the goal, by providing a set of simple transformations such that f reduces many tests to the same test, has low probability of reducing two

tests failing for different reasons into the same test, and f is not unreasonably expensive to compute. The implementation for f (in fact, for a family of f-approximating functions, with different tradeoffs in runtime and level of normalization) involves defining a set of rewrite rules such that for a test b (the base test), the rules define a finite set of candidate tests $c \in C(b)$, possible simplifications of b, where each c is the result of applying some rewrite, r_i to b. The notion of simplicity is defined by a restriction on the rewriting rules. For any test case t, let $R(t)$ be the length of the maximum number of rewrites that can be applied to t, e.g., the longest possible sequence such that $c_0 = r_{i1}(t), c_1 = r_{i2}(c_0), ...c_n = r_{in}(c_{n-1})$. We require that $\forall c \in C(b), R(c) < R(b)$. The number of possible rewrites for each candidate must be smaller than the number of possible rewrites for b. This implies further restrictons, e.g., no rewriting can ever reverse another rewrite's effects. Such a rewrite system is *strongly normalizing*: any sequence of rewrites chosen will eventually end in a term (test case) that cannot be further rewritten.

In the setting of TSTL, where test actions and have a defined total order, a simple principle can be applied to produce strongly normalizing rewrite rules: rewrites should reduce the sum of the indices of the actions in the test case. This approach provides effective normaliization at a significant, but not unreasonable, computational cost.

In order to formally define normalization, some additional notation is required. A *step* is an action paired with an index indicating its position in a text, where the first action is step 0, etc., e.g.: $(2 : a)$ indicates the third step of the test is action a (indexing is from 0). $\Delta(t, t')$ is the set of all steps in t such that $t(i) \neq t'(i)$.

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

The rewrite $t[x \Rightarrow y]$ denotes the test t with all instances of x replaced by y. Here, x and y can be actions, steps, or pools. $t[x \Leftrightarrow y]$ is similar, except that x and y are swapped. Rewrite $t(i, j)[x \Rightarrow y]$ is the same as $t[x \Rightarrow y]$, except that the replacement is only applied between steps i and j, inclusive. Finally, $t_i(x)$ denotes t with all steps containing x that are before step i moved to step i, preserving their previous order, and moving steps at i and after i to make room.

1) **SimplifyAll:** $c = b[a \Rightarrow a']$
   where $a' < a$
   Covers the case where all appearances of an action can be replaced with a simpler (lower-indexed) action.
2) **ReplacePool:** $c = b(i, j)[p \Rightarrow p']$
   where $p < p'$ and $0 \leq i < j < |b|$
   Covers the case when all appearances of an instance of a pool can be replaced with a lower-indexed instance of that pool (with possible restriction to a range of steps).
3) **ReplaceMovePool:** $c = b_{\rightarrow i}(p')[p \Rightarrow p']$
   where $p < p'$ and $0 \leq i < |b|$
   Covers the case when all appearances of an instance of a pool can be replaced with a lower-indexed instance of that pool, if all assignments to the new instance before a certain step are moved to that step.
4) **SimplifySingle:** $c = b[(i : a) \Rightarrow (i : a')]$
   where $a' < a$
   Covers the case where one action can be replaced with a simpler (lower-indexed) action.
5) **SwapPool:** $c = b(i, j)[p \Leftrightarrow p']$
   where $\Delta(c, b) < \Delta(b, c)$
   and $0 \leq i < j < |b|$
   and $p < p'$
   Covers the case where swapping two pool instances (within a range of steps) reduces the minimal action index of the modified steps.
6) **SwapAction:** $c = b[(i : a) \Rightarrow (i : b), (j : b) \Rightarrow (j : a)]$
   where $i < j$ and $b < a$
   Covers the case where two actions can be swapped in the test, with the lower-indexed action now appearing earlier.
7) **ReduceAction:** $c = b[(i : a) \Rightarrow (i : a')]$
   where $|ddmin(c)| < |ddmin(b)|$
   Covers the case where an action can be replaced by any action (not just lower-indexed actions) and this enables further delta-debugging-based reduction of the test case's length.

These rules alone do not determine a complete normalization method; it is also required to determine an 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 both performed at once, interleaved (e.g., for every possible replacement of a pool, both rules are checked, in the order given above). The core algorithm, given a set of ordered rewrite rules defining $C(b)$ 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 b. In most cases, this predicate will be "the test fails" but we also have preserve code coverage in regression suites [8].

Algorithm 1 Basic algorithm for simplification

```
1:  while modified do
2:      modified = False
3:      for c ∈ C(b) do
4:          if pred(c) then
5:              modified = True
6:              b = c
7:              break (exit For loop)
8:          end if
9:      end for
10: end while
11: return b
```

The cost of normalizing a test case is, in the worst case, extremely large. Consider a testing scenario where there are n actions, each of which is always enabled. For a test case of k steps, each of which is the highest indexed action, checking the candidates produced by the **SimplifySingle** rule alone requires

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

Fig. 5. Original failing test case for numpy (42 steps)

performing n^k test executions. If we further assume that the sequence of successful rewrites first rewrites the final step by reducing its index by one, which makes it possible to reduce the index of the previous step by one, and so forth, it is easy to see that nearly kn^k test case runs are required. In practice, the number of required test case runs is proportional to the length of the test k, but most actions are not enabled at most steps, and the rules are applied in an order that quickly converges on a normal form.

For test cases with a very short runtime, this algorithm is usually practical, more expensive than delta-debugging but requiring less than a minute to run. However, when test case execution is expensive, the set of candidate test cases must be further restricted. In our experiments, we found that restricting action replacements to cases where the Levenshtein [9] distance (text edit distance) between the code for the actions was in some range was effective in reducing runtime, while almost always having no impact on the final result. In practice, most test actions can only be replaced by syntactically similar actions, without completely changing test semantics.

```
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
#  or array1 = array0 - array0
#  or array2 = array0 - array0
#  or array3 = array0 - array0
#  or array4 = array0 - array0
#  or array5 = array0 - array0
assert (np.array_equal(array0,array0))
```

Fig. 6. Normalized and generalized test case (15 steps)

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

Fig. 7. Original test case for ESRI arcpy library (19 steps)

IV. Generalization Algorithm

V. Case Studies

VI. Related Work

This work builds on the ideas behind delta-debugging [1], that test cases should not contain extraneous information that is not needed to reproduce failure. It extends this beyond length to include a notion where some actions in a test are considered simpler than others, which in some cases also allows further reduction of the length of the test case as well. Delta-debugging and slicing [10] for test case reduction are limited, generally, to producing subsets of the original test case, not modifying parts of the test to obtain further simplicity.

Normalization is in part motivated by the fuzzer taming [4] problem: determining how many distinct faults are present in a large set of failing test cases. This is a key problem in practical application of automated testing. Our previous work on fuzzer

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

Fig. 8. Normalized and generalized test case for ESRI arcpy library (12 steps)

taming in fact uses delta-debugging as a simple normalization method that reduces some test cases to syntactical duplicates; the methods are orthogonal.

Zhang [11] proposed an alternative approach to semantic test simplification. Like our approach, Zhang's is semantic, 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. This limits its ability to simplify tests and makes it a fairly weak normalizer (in fact it makes no attempt to perform syntactic normalization, e.g., not making sure to use the same variables when variable name is irrelevant to failure).

CReduce [12] performs some simple normalization as part of a complex test-case reduction scheme for C code, and the peephole-rewrite scheme used in CReduce is also an inspiration for the approach taken by our normalizer.

Work on producing readable tests [13], [14] is also related, in that it aims to "simplify" tests in a way that goes beyond measuring test length. Work on readable tests is primarily intended to assist debugging in a human-factors sense, while our normalization and generalization aims to increase the information density of a test (hopefully increasing readability as well, of course), and to address the fuzzer taming problem. The approaches are essentially orthogonal: our normalized tests could be altered to improve readability by these methods.

Test case generalization is also related to dynamic invariant generation, in that it informs the user of invariants over a series of test executions satisfying some property [15].

VII. Conclusions and Future Work

References

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

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

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

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

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

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

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

[8] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction for quick testing," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 243–252.

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

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

[11] S. Zhang, "Practical semantic test simplification," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1173–1176. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486953

[12] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Conference on Programming Language Design and Implementation*, 2012, pp. 335–346.

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

[14] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 107–118.

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