A Suite of Tools for Making Effective Use of Automatically Generated Tests

Josie Holmes
Pennsylvania State University, USA

Alex Groce
School of Informatics, Computing, and Cyber Systems,
Northern Arizona University, USA
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 Making Effective Use of Automatically Generated Tests. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10-14, 2017 (ISSTA'17),* 4 pages.
DOI: 10.1145/3092703.3098220

1 INTRODUCTION

Automated test generation tools, in essence, exist in order to produce failing tests. However, once a tool has produced either one such test or a large set of such tests, the real work of a user of such a tool begins in earnest. First, users of course wish to be able to replay failed tests, and run batches of tests as regressions (possibly collecting code coverage information as well). If the test format of a tool is not executable, it is also useful to be able to produce executable, standalone tests. Second, with most test generation methods, it is important to reduce the size of the test to make it easier to understand (and quicker to run as a regression test) [7, 10, 20, 21, 29]. Most industrial-strength automated testing systems support test minimization, usually using a variation of delta-debugging [22, 27, 28]. Some such systems make it easy to customize the criteria by which a test is reduced. However, such systems usually do not provide algorithmic defenses against the problem of slippage (where, in the absence of a good failure labeling system, the reducer may change a failure due to one fault to a failure due to another, and often less interesting, fault [4, 16]). Additionally, a user may want to *semantically* simplify a test, to make it not only shorter but simpler and less complex in ways that go beyond mere test length [9]. Such functionality is less common, or only a byproduct of specialized minimization methods. Finally, users may want to be informed of the neighborhood of a failing test: which similar tests also fail (and, equally important, which similar tests do not fail) [9]. Again, this functionality is usually not present in current automated test generation tools.

The TSTL testing language and tool, currently implemented for testing Python programs, supports all of these features, both as command-line tools and API interfaces for more complex uses.

ISSTA'17, Santa Barbara, CA, USA

978-1-4503-5076-1/17/07...$15.00
DOI: 10.1145/3092703.3098220

2 A BRIEF TSTL PRIMER

TSTL [12, 13, 17] is a language, tool suite, and "library constructor" for testing Python programs. TSTL aims to offer both the immediate feedback of property-driven testing tools like QuickCheck and Hypothesis [5, 22] and longer-term automated test generation, as well as serve as a platform for experimenting with novel test generation and test manipulation methods. Unlike most QuickCheck-like tools, TSTL is focused on generating unit tests that consist of sequences of method/function calls [2], rather than generating input data for functions (though TSTL can generate arbitrary data, since data creation is usually easily expressed as a sequence of constructions and modifications of data). TSTL is available on github at https://github.com/agroce/tstl. Simply typing `pip install tstl` on a system with pip installed will also install TSTL.

The user of TSTL writes a test harness [8] that describes the actions that are possible during testing, and the pools of values that are generated during testing (these are both the inputs to methods tested and the objects to be tested, in most cases). Figure 1 shows a simple TSTL harness to test a Python stack implementation. The harness creates integer values (in the range 1-20) and stacks, up to 4 of each. It calls the various methods of the stack. For stack

```
@import stack

pool: <stack> 4        # These are the variables used in tests,
pool: <value> 4        # stack0...stack3 and value0...value3

<value> := <[1..20]>       # := is Python assignment, but
<stack> := stack.Stack()   # marks a value  as initialized

<stack>.push(<value>) # This is a test action
{IndexError} <value> := <stack>.pop() # ... as are all these
not <stack,1>.isEmpty() -> <value> := <stack>.pop()
{IndexError} <value> := <stack>.peek()
not <stack,1>.isEmpty() -> <value> := <stack>.peek()
```

Figure 1: A simple TSTL harness for a stack.

operations that can cause underflow, there is both a version of the action that guards the action with a check on stack emptiness and a version that throws away the return value and allows the call to throw an IndexError exception. TSTL also supports automatic differential [24] testing, where one implementation serves as a reference model for the Software Under Test, and other features. See the journal paper on TSTL [17] or the online documentation on github for more details.

To make use of the stack harness, we save it into a file, such as stack.tstl. Then at a command prompt, we compile the file into a standalone interface for testing the stack, sut.py, and (usually) invoke the basic TSTL testing tool, tstl_rt to look for faults.

```
> tstl stack.tstl
> tstl_rt --timeout 30
```

If we forget how to use the tools, all TSTL command line tools produce a full list of options when called with the --help argument. For most tools this list is short and simple; the "random tester" tstl_rt, however, has a very large number of options, since it supports pure random testing, swarm testing [15], genetic-algorithm based testing, control over action probabilities, Markov-model driven testing, and a large array of other configuration settings.

TSTL generates 738 tests (of 100 operations each) and performs 73,749 actions. In this case, there is no fault to be found. This paper describes the tools TSTL provides for working with tests in the instance when a fault is detected.

3 THE BASIC TSTL TEST TOOLS

Instead of the fault-free stack, we can test a real-world program with real faults, such as the SymPy library for performing symbolic mathematics in Python [1]. The SymPy harness can be found in the TSTL github repository examples/sympy directory.

```
 > tstl sympy.tstl
 > tstl_rt --swarm --noCover --full
...
 UNCAUGHT EXCEPTION
 ERROR: (<type 'exceptions.RuntimeError'>,
 RuntimeError('maximum recursion depth exceeded',)
...
    return func(a, b)
...
 SAVING TEST AS failure.67076.test
...
 STOPPING TESTING DUE TO FAILED TEST
 36.4984600544 TOTAL RUNTIME
 > wc -l failure.67076.test
 72
 > head -n5 failure.67076.test
 self.p_v[3] = sympy.Symbol('j',positive=True)
 self.p_expr[0].evalf()
 self.p_expr[0] = self.p_expr[0] + self.p_expr[3]
 self.p_expr[3] = self.p_expr[3] + self.p_expr[1]
 self.p_expr[1] = self.p_expr[3] % self.p_expr[0]
```

We have instructed the random tester to use swarm testing [15] and not collect code coverage, in order to improve the chances of quickly finding a fault. By default tstl_rt uses delta-debugging to minimize tests before saving them, but we have also instructed tstl_rt to simply save the original test case --full. The unreduced test (which causes Python to enter an infinite recursion sequence) consists of 72 steps, saved in a non-executable, technically (but not very) human-readable, textual format (in an automatically generated file name, based on the process ID). This is not a very useful test, so we want to reduce it:

```
> tstl_reduce failure.67076.test reduced.test --noNormalize
STARTING WITH TEST OF LENGTH 72
REDUCING...
REDUCED IN 31.3780119419 SECONDS
NEW LENGTH 7
ALPHA CONVERTING...
c0 = sympy.Integer(4)                          # STEP 0
c1 = sympy.Integer(9)                          # STEP 1
v0 = sympy.Symbol('k',positive=True)           # STEP 2
expr0 = sympy.Rational(c1,c1)                  # STEP 3
expr1 = sympy.Product(expr0,(v0,c0,c0))        # STEP 4
expr2 = c1                                     # STEP 5
expr3 = expr2 % expr1                          # STEP 6
```

This test, reduced using standard delta-debugging [29], is short. Also, note that TSTL automatically alpha-converts the test so that it uses variables to store intermediate values in a reasonable way (starting with v0 rather than arbitrarily beginning with v3, for example). However, the test is neither as short as possible nor, more importantly, as simple as possible. For debugging we may well wonder: does it matter that c0 is 4 and c1 is 9? Is the use of the variable k relevant? If we want to know the answers, we can run the reducer to *normalize* [9] the test, in place of simply reducing it:

```
> tstl_reduce reduced.test normalized.test --noReduce
STARTING WITH TEST OF LENGTH 7
NORMALIZING...
NORMALIZED IN 383.565114975 SECONDS
NEW LENGTH 5
c0 = sympy.Integer(1)                          # STEP 0
v0 = sympy.Symbol('a')                         # STEP 1
expr0 = c0                                     # STEP 2
expr1 = sympy.Sum(expr0,(v0,c0,c0))            # STEP 3
expr0 = expr0 % expr1                          # STEP 4
```

Notice that normalizing a test is much more expensive than simply reducing it, but the payoff is an even shorter and simpler test[1]. By default, the TSTL random test generator reduces tests before saving them, and the standalone tstl_reduce tool is used to normalize interesting tests. Calling tstl_rt with the --normalize option avoids going through the standalone tool. Normalization here pays off by revealing that the use of a Rational and exact numeric/symbol values are not relevant.

Now that we have an extremely simple test, we can replay it in a "verbose" mode to see more exactly what is happening during the test, as shown in Figure 2. This shows the values, types, and changes in values of every pool variable involved in each step of the test (and would show the state of a reference implementation, if we were performing automated differential testing).

In addition to replaying a single test, we can replay a number of saved tests using the tstl_regress command, which takes as input a list of all test files to run, and produces a coverage report in addition to the outcome of each test. By default it stops on the first

[1]For details on how much shorter and simpler, see the conference paper on test case normalization and generalization [9]. Note that normalization times are usually faster in the current release than reported in that paper, due to the implementation of a useful heuristic for reducing nearly 1-minimal tests suggested by David R. MacIver [23], the author of the Hypothesis tool.

```
> tstl_replay normalized.test --verbose
STEP #0: ACTION: c0 = sympy.Integer(1)
c0 = None : <type 'NoneType'>
=> c0 = 1 : <class 'sympy.core.numbers.One'>
==================================================
STEP #1: ACTION: v0 = sympy.Symbol('a')
v0 = None : <type 'NoneType'>
=> v0 = a : <class 'sympy.core.symbol.Symbol'>
==================================================
STEP #2: ACTION: expr0 = c0
c0 = 1 : <class 'sympy.core.numbers.One'>
expr0 = None : <type 'NoneType'>
=> expr0 = 1 : <class 'sympy.core.numbers.One'>
==================================================
STEP #3: ACTION: expr1 = sympy.Sum(expr0,(v0,c0,c0))
c0 = 1 : <class 'sympy.core.numbers.One'>
v0 = a : <class 'sympy.core.symbol.Symbol'>
expr0 = 1 : <class 'sympy.core.numbers.One'>
expr1 = None : <type 'NoneType'>
=> expr1 = Sum(1, (a, 1, 1)) : <class 'sympy.concrete.summations.Sum'>
==================================================
STEP #4: ACTION: expr0 = expr0 % expr1
expr0 = 1 : <class 'sympy.core.numbers.One'>
expr1 = Sum(1, (a, 1, 1)) : <class 'sympy.concrete.summations.Sum'>
RAISED EXCEPTION: <type 'exceptions.RuntimeError'>
  maximum recursion depth exceeded in cmp
FAILED STEP
(<type 'exceptions.RuntimeError'>,
RuntimeError('maximum recursion depth exceeded in cmp',)
...
```

Figure 2: Verbose replay of a TSTL test.

failing test, but can be directed to run all tests with `--keepGoing`. Regression runs can also generate an HTML coverage report using the facilities of the `coverage.py` library [3].

Finally, we can *generalize* the test, to see what alternative, similar tests also produce the same failure:

```
> tstl_generalize normalized.test
GENERALIZING...
#[
c0 = sympy.Integer(1)                                # STEP 0
#  or c0 = sympy.Integer(2)
#   - c0 = sympy.Integer(10)
v0 = sympy.Symbol('a')                               # STEP 1
#  or v0 = sympy.Symbol('b')
#   - v0 = sympy.Symbol('d')
#  or v0 = sympy.Symbol('x')
#   - v0 = sympy.Symbol('z')
#  or v0 = sympy.Symbol('e',positive=True)
#   - v0 = sympy.Symbol('l',positive=True)
#  swaps with step 2
#] (steps in [] can be in any order)
expr0 = c0                                           # STEP 2
#  or expr0 = sympy.Rational(c0,c0)
#  or expr0 = sympy.pi
#  or expr0 = sympy.E
#  or expr0 = sympy.I
#  swaps with step 1
expr1 = sympy.Sum(expr0,(v0,c0,c0))                  # STEP 3
#  or expr1 = sympy.Product(expr0,(v0,c0,c0))
expr0 = expr0 % expr1                                # STEP 4
#  or expr2 = expr0 % expr1
#  or expr3 = expr0 % expr1
GENERALIZED IN 239.682291985 SECONDS
```

With this information, the basic underlying structure of the fault is made clear: using the modulo operator on a `Sum` or `Product` over an empty range (whether that range is $2 \ldots 2$ or $\pi \ldots \pi$, with any variable name allowed by our SymPy harness, causes the failure. The ordering of operations, other than to the extent required for data flow, is not important.

Now that we understand the fault, we may want a non-TSTL test to run in a debugger to try out possible solutions. Generating a standalone Python executable test is easy:

```
> tstl_standalone normalized.test normalized.py
```

In this example, reduction or normalization has always been with respect to a failure. However, simply by providing the `--coverage` option to `tstl_reduce` or `tstl_generalize` the same approaches can be applied to reduce tests by their code coverage, a useful method for producing very efficient regression tests [6, 7]. Running `tstl_rt` with the `--quickTests` option will also produce a suite of such coverage-based reduced regression tests.

4 AVOIDING SLIPPAGE

Test slippage [4, 16] is when a weak labeling of failed tests (e.g., simply checking that a failing test still causes some kind of uncaught exception) results in a test that originally failed due to one fault being reduced to a test that fails due to a different fault.

There is a need for flexibility in handling slippage and fault signatures in general; with some programs, many exceptions may reveal the same fault, with other programs even the same assertion on the same line of code can be violated due to different underlying faults. TSTL therefore provides a few ways to avoid slippage, and also some ways to intentionally induce "good" slippage where a failing test is reduced to produce multiple tests that fail due to different faults [16]. First, the random test generator and the reduction, and generalization tools all take the `--keepLast` option, which forces reduced tests to have the same final action as the original test. This is a heuristic for avoiding slippage discovered during file system testing at NASA [10]. Second, the reducer and generalizer take a `--matchException` argument that forces reductions to fail due to the same type of exception (but not exact message); this is the default behavior for the random tester, where the user has more reason to be concerned about losing the original fault since it is not stored in a file.

While these methods are useful for producing more precise labels for failure, they are not helpful in instances where precise labeling is impossible, such as many differential testing settings [4]. For these cases, and for using reduction as a mutation-based fuzzing tool to look for new faults, TSTL provides two more modes. First, using the `--multiple` option configures `tstl_reduce` to use the `comb-block` algorithm [16] to attempt to produce as many reduced tests as possible, that are all as different as possible from each other. The effort extended to consider combinations of test components can be configured with the `--recursive` and `--limit` options. Second, the `--random` flag to the reducer causes the order of possible reductions to be randomized, so that different runs of the reducer will produce different reduced tests.

5 API ACCESS TO TOOL FUNCTIONALITY

In addition to the command-line tools described here, TSTL also makes it easy to perform sophisticated test manipulations in code. When a TSTL harness is compiled it produces an `sut` module providing an abstract interface for testing the SUT. It is this interface that `tstl_rt`, `tstl_reduce`, and the other tools interact with, making test generation and manipulation independent of the SUT.

The interface includes `reduce`, `normalize` and `generalize` methods for reduction and normalization that provide many more parameters for fine-tuned control of the algorithms than are provided by the command line tools. These methods are all higher-order functions, so the predicate for the algorithm to maintain as true can be an arbitrary function of a test. The interface to the SUT also provides methods to return commonly used predicates, such as matching the coverage of a test, or failing a property check. Because TSTL's reduction implementations do not require their initial input to satsify the predicate, this can be used for unusual applications. For example, if are testing an XML parser, have a long, high-coverage test, and wish to modify it to produce an input that takes as long as possible to parse, we can define a function:

```
def takesLonger(t):
      global WCET, SUT
      start = time.time()
      SUT.replay(t)
      elapsed = time.time() - start
      if elapsed > WCET:
         WCET = elapsed
         return True
      return False
```

and call `SUT.reduce(longTest,takesLonger)` after setting WCET to the runtime for the initial test.

6 RELATED WORK

The tools described here are obviously inspired by delta-debugging [29] and the idea that tests should not contain extraneous parts not needed to cause test failure (or other behavior of interest [6, 7]). Delta-debugging and slicing [21] produce subsets of the original test, but do not modify parts of the test to obtain further simplicity. Our work on normalization [9] extends this idea to rewrite tests into a more canonical' form.

Zhang [30] proposed an approach to semantic test simplification that is also able to modify, rather than simply remove, portions of a test. However, Zhang's simplification operates directly over a fragment of Java, rather than using an abstraction of test actions, with limited power: no new methods can be invoked, statements cannot be re-ordered, and no new values are used. It also does not even force a test to use fixed variable names when variable name is irrelevant. CReduce [27] performs some simple normalization as part of its test reduction for C code. By writing a TSTL harness that is in the form of constructor calls to create an AST, TSTL can reduce and normalize hierarchically structured input data in ways similar to CReduce and Hierarchical Delta Debugging [25]. The methods for avoiding slippage are based on both our recent work [16] and older heuristics for avoiding test slippage [10].

The most closely related work to our test generalization [9] is Pike's SmartCheck [26]. 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 [11, 14, 18].

7 CONCLUSIONS AND FUTURE WORK

This paper presents a set of tools, part of the TSTL [17] testing language and tool suite, for letting users make the most of the tests the tool generates. In addition to standard replay, regression, and minimization, TSTL implements some powerful new techniques from the recent literature for manipulating tests [9, 16].

As future work, we plan to continue to develop TSTL's tools for working with tests. Some improvments are simple: for instance, the TSTL random tester currently provides simple fault localization over the tests generated during a run (if there are any failures) [19], but not for regression tests. More importantly, we plan to continue to use TSTL as a platform for experimenting with and making available novel methods for making use of automatically generated tests, including methods for composing and de-composing tests and generating information from tests that can be used to guide future testing.

REFERENCES

[1] SymPy. http://www.sympy.org/en/index.html.
[2] J. Andrews, Y. R. Zhang, and A. Groce. Comparing automated unit testing strategies. Technical Report 736, Department of Computer Science, University of Western Ontario, December 2010.
[3] N. Batchelder. Coverage.py. https://coverage.readthedocs.org/en/coverage-4.0.1/.
[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*, pages 197–208, 2013.
[5] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.
[6] 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.
[7] 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.
[8] A. Groce and M. Erwig. Finding common ground: Choose, assert, and assume. In *International Workshop on Dynamic Analysis*, pages 12–17, 2012.
[9] 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.
[10] 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.
[11] A. Groce and D. Kroening. Making the most of BMC counterexamples. *Electron. Notes Theor. Comput. Sci.*, 119(2):67–81, Mar. 2005.
[12] A. Groce and J. Pinto. A little language for testing. In *NASA Formal Methods Symposium*, pages 204–218, 2015.
[13] 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.
[14] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.
[15] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *International Symposium on Software Testing and Analysis*, pages 78–88, 2012.
[16] J. Holmes, A. Groce, and A. Alipour. Mitigating (and exploiting) test reduction slippage. In *Workshop on Automated Software Testing*, 2016.
[17] 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.
[18] 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.
[19] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Automated Software Engineering*, pages 273–282, 2005.
[20] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *International Symposium on Software Reliability Engineering*, pages 267–276, 2005.
[21] 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.
[22] D. R. MacIver. Hypothesis: Test faster, fix more. http://hypothesis.works/.
[23] D. R. MacIver. Personal communication.
[24] W. McKeeman. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.
[25] G. Misherghi and Z. Su. Hdd: hierarchical delta debugging. In *International Conference on Software engineering*, pages 142–151, 2006.
[26] L. Pike. SmartCheck: automatic and efficient counterexample reduction and generalization. In *ACM SIGPLAN Symposium on Haskell*, pages 53–64, 2014.
[27] 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.
[28] J. Ruderman. Bug 329066 - Lithium, a testcase reduction tool (delta debugger). https://bugzilla.mozilla.org/show_bug.cgi?id=329066, 2006.
[29] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.
[30] S. Zhang. Practical semantic test simplification. In *International Conference on Software Engineering*, pages 1173–1176, 2013.