TSTL: a Language and Tool for Testing (Demo)

Alex Groce, Jervis Pinto, Pooria Azimi, Pranjal Mittal
Oregon State University
agroce@gmail.com,pinto@eecs.oregonstate.edu,
azimip@onid.oregonstate.edu,mittal.pranjal@gmail.com

ABSTRACT

Writing a test harness is a difficult and repetitive programming task, and the lack of tool support for customized automated testing is an obstacle to the adoption of more sophisticated testing in industry. This paper presents TSTL, the Template Scripting Testing Language, which allows users to specify the general form of valid tests for a system in a simple but expressive language, and tools to support testing based on a TSTL definition. TSTL is a minimalist template-based domain-specific language, using the source language of the Software Under Test (SUT) to support most operations, but adding declarative idioms for testing. TSTL compiles to a common testing interface that hides the details of the SUT and provides support for logging, code coverage, delta debugging, and other core testing functionality, making it easy to write universal testing tools such as random testers or model checkers that apply to all TSTL-defined harnesses. TSTL is currently available for Python, but easily adapted to other languages as well.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability

Keywords

Domain-specific languages, testing tools, Python

1. INTRODUCTION

Automated testing often requires a user to write a *test harness* — essentially a program that defines and generates the set of valid tests for the Software Under Test (SUT). Such harnesses are common to random testing, many kinds of model checking, and various machine-learning influenced approaches [6]. Unfortunately, these harnesses themselves are complex software artifacts and it is all too easy to spend valuable testing time hunting down bugs in the test harness and not the SUT. Harness code is often highly repetitive (choosing between a set of available API calls to make, and assigning values to parameters in those calls, for example) and is almost always tightly coupled to one particular test generation method. TSTL [9] is a language and tool intended to make these difficulties less onerous.

First, TSTL allows a harness to be defined in a declarative style, and provides support for many common testing idioms. Second, adapting a more declarative approach allows TSTL to output a generalized interface for testing: a class that allows a testing tool to determine available test actions, check for success or failure of tests as they progress, and even automatically produce logs, collect and analyze code coverage, and delta-debug failed tests, without change for different SUTs, but without sacrificing the generality of a hand-built test generator: the definition of valid tests also encodes the idiosyncrasies of the SUT.

Figure 1 shows a simple TSTL file, defining valid tests of an AVL tree implementation. Most of the file is actually Python code. TSTL compiles such a file into Python code defining a class for testing tools to interface. Figure 2 shows an example of a simple random tester using that interface, concise but featuring delta debugging [17] and branch coverage reporting. The key idea of TSTL is to take a definition of valid actions (typically API calls) that can be performed by the SUT and transform this into a class representing a graph with transitions between the states produced by performing those actions. This allows us to separate the problem of describing the set of valid tests of a program from the problem of actually choosing and executing some of those tests. TSTL tracks code coverage, handles exceptions, and performs other book-keeping (including providing easy-to-use state storage and backtracking), resulting in a simple clean definition not only of test harnesses but of algorithms for searching their induced test graphs.

In this paper we demonstrate the current functionality of TSTL. A more detailed description of TSTL [9] is available elsewhere, providing further information on the algorithm used to compile (really, expand) TSTL harnesses, the core TSTL language in BNF, and the design choices behind TSTL. Since that paper was finalized, TSTL has been used by over 100 students, and refined based on their feedback and our own work in practical testing problems and research. New features include the ability to refer to the value of an expression at function entry with the construct `PRE%(expr)%`, a logging facility, support for serializing tests in order to

ISSTA '15 Baltimore, MA USA

```
01 @import avl
02 @import math

03 <@
04 def heightOk(tree):
05     h = tree.tree_height()
06     l = len(tree.inorder())
07     if (l == 0):
08        return True
09     m = math.log(l,2)
10     # Demonstrate PRE
11     assert (PRE%(tree.inorder())%
12             == tree.inorder())
13     return h <= (m + 1)

14 def it(s):
15     l = []
16     for i in s:
17        l.append(i)
18     return sorted(l)
19 @>

20 source: avl.py

21 pool: %INT% 4
22 pool: %AVL% 2 REF
23 pool: %LIST% 2

24 log: 1 %AVL%.inorder()

25 property: heightOk(%AVL%)
26 property: %AVL%.check_balanced()

27 %LIST%:=[]
28 ~%LIST%.append(%INT%)
29 %INT%:=%[1..20]%

30 %AVL%:=avl.AVLTree()
31 %AVL%:=avl.AVLTree(%LIST%)

32 ~%AVL%.insert(%INT%) =>
   (len(%AVL,1%.inorder()) ==
      PRE%(len(%AVL,1%.inorder()))%+1)
   or PRE%(%AVL,1%.find(%INT,1%))%
33 ~%AVL%.delete(%INT%) =>
   (len(%AVL,1%.inorder()) ==
      PRE%(len(%AVL,1%.inorder()))%-1)
   or not PRE%((%AVL,1%.find(%INT,1%)))%
34 ~%AVL%.find(%INT%)
35 %AVL%.inorder()
36 len(%AVL,1%.inorder()) > 5 ->
   %AVL%.display()

37 reference: avl.AVLTree ==> set
38 reference: insert ==> add
39 reference: delete ==> discard
40 reference: find ==> __contains__
41 reference: (\S+)\.inorder\(\) ==> it(\1)
42 reference: METHOD(display) ==> CALL(print)

43 compare: find
44 compare: inorder
```

Figure 1: A simple TSTL file to test an AVL tree implementation, demonstrating various features of TSTL. The `AVLTree` class defines methods including `insert`, `delete`, `find`, etc.; it is not shown here due to lack of space.

```
import sut
import random
sut = sut.sut()
NUM_TESTS = 1000
TEST_LENGTH = 200
for t in xrange(0,NUM_TESTS):
   sut.restart()
   T = []
   for s in xrange(0,TEST_LENGTH):
       action = random.choice(sut.enabled())
       T.append(action)
       r = safely(action)
       if sut.newBranches() != []
          print 'NEW BRANCHES:'
          print sut.newBranches()
       if not r:
          print 'EXCEPTION:', sut.error()
          R = sut.reduce(T, sut.fails)
          print 'FAILING TEST:', R
       elif not sut.check():
          print 'PROPERTY FAILED:'
          print sut.error()
          R = sut.reduce(T, sut.failsCheck)
          print 'FAILING TEST:', R
```

Figure 2: A simple random tester using the interface provided by TSTL.

save them to a file, a method for expecting certain changes in values from a call, and a generalized delta debugging [17] facility supporting arbitrary properties (e.g., reducing a test case so it keeps the same code coverage [5]) in addition to bug reduction. While this tool paper focuses on TSTL's use in API-call based testing, TSTL can also support grammar-based testing, as shown in the NFM paper [9]. TSTL is available from github (`https://github.com/agroce/tstl`) or as a package in Pypi.

2. A BRIEF TSTL TUTORIAL

We demonstrate TSTL's use to test an AVL tree implementaiton (Figure 1) as an example of a typical SUT.

2.1 Section 1: Python Code

One of the challenges of defining a domain-specific language for testing is that test harnesses often need to perform arbitrary computation, and programmers do not want to learn a new language. TSTL is a template language that expands to source code in the SUT's language, and complex functionality is handled by using the underlying language, in this case Python. The harness begins (lines 1-2) with `import` statements to load any needed Python modules, including the actual SUT, the `avl` module. TSTL lines beginning with an "@" indicate raw Python code. TSTL processes such lines in some cases (for example, reimporting any modules when the SUT is re-initialized to its starting state), but in general simply reproduces them in compiled output.

Lines 3-19 define two utility functions for testing. The first of these checks that the AVL tree in question satisfies the AVL guarantee that a tree will be balanced. It also checks that after the height computation, the tree's inorder traversal is unchanged from its value at function entry. TSTL au-

tomatically expands this into the equivalent Python code, caching `PRE` expressions used multiple times in the body. The `it` function returns an "inorder traversal" of the Python set used as a reference implementation for the AVL tree, discussed below.

2.2 Section 2: Preamble

Lines 20-26 define some basic properties of the test harness. First, line 20 tells TSTL that the `avl.py` file contains all source code on which coverage should be collected.

Lines 21-23 define the *value pools* that are used in testing [1, 4]. A test in TSTL is a sequence of assignments and calls, and the value pool contains the values used. A pool value consists of a name (e.g. `%INT%`), an integer noting how many different values of that pool are available (here, the harness can remember up to 4 different `INT`s), and, optionally, the keyword `REF` indicating that two copies of all pool values of this type are to be maintained, one for the SUT itself and one as a reference implementation for differential testing [12, 8]. Pool values are always set off in "%" characters to distinguish them from actual Python variables, since pool values potentially represent several actual Python variables, and are used by TSTL to define the graph structure of a test, as explained in Section 2.3. Here the AVL trees are maintained as a reference pool, since we want to automatically test that the AVL tree implements a set.

Line 24 tells TSTL that if the logging level is at least 1, the inorder traversal of every AVL pool should be printed after each test action.

Lines 25 and 26 declare the properties that must be invariantly true after each test action: first, the height of every AVL tree must be ok, as defined by our Python function, and second the AVL implementation's own function for ensuring it is balanced must return `True`. Recall that our harness maintains two AVL pool values, so TSTL produces these for each AVL tree in the pool.

2.3 Section 3: Actions

Lines 27-34 are the heart of the harness definition, the actual steps that can be taken in a valid test. Lines 27-28 allow the construction of arbitrary length lists of integers. Line 29 defines the range of integers allowed in the `INT` pool, using the `%[..]%` construct.. Lines 30 and 31 provide for initialization of AVL trees, either with the constructor for an empty tree or the constructor that takes a list and produces an AVL tree containing all elements in the list. Lines 32-36 allow the basic AVL tree API calls — insertion, deletion, searching, inorder traversal, and printing. Note that code can be proceeded by a list (in curly braces) of exceptions to be expected as possible valid behavior. The two most interesting features shown here are 1) that guards can be defined by a user, limiting the conditions in which an action is enabled, by the syntax `guard -> action` and 2) a check to perform after an action has been performed (which can include use of `PRE` can be provided, using the syntax `action => check`.

In large part, this section of the harness is just Python code, with one line for each action. The TSTL compiler is responsible for producing a valid graph definition based on the pool values used from this code. Actions using a pool value are not enabled until the pool is initialized — lines with an assignment using `:=` define initializations. Furthermore, a pool value cannot be re-initialized until it has been used in some other action, to avoid useless test sequences such as assigning first 1 and then 5 to the same `INT` pool value. The use of a "~" before a pool value use indicates that this usage should not enable re-initialization. Appending an item to a LIST, for example, does not "use it up" but constructing an AVL tree from a list allows that pool value to be re-initialized. AVL trees cannot be over-written until they have been traversed at least once.

2.4 Section 4: Reference Definition

One of the most convenient features of TSTL is its integrated support for differential testing, where the SUT's behavior is compared to that of a reference implementation [12, 8], and differences are reported as bugs. Differential testing is a powerful method for finding bugs in software ranging from simple container classes to optimizing compilers [16], but its implementation in a testing system often requires writing a large amount of repetitive, and sometimes subtle, code. TSTL allows a harness to indicate, using textual patterns, how calls to the SUT are to be transformed into calls to a reference, and which return values from the SUT and reference should be compared for equivalence.

Lines 37-42 show that to create a reference for an AVLTree, the matching constructor (either taking no argument or a list) should be called in Python's built-in `set` class. AVL insertion is handled by the `add` method for sets, and deletion with `discard`. AVLTree `find` is equivalent to the `__contains__` method on Python sets, and an inorder traversal (sorted list of elements) is provided by our `it` function. Note that the last reference definition, on line 41, has to use regular expressions to transform an AVLTree method call into a function call on the reference set. There is also, as shown in line 42, syntactic sugar for this transformation (or the opposite). Finally, lines 43 and 44 tell TSTL that whenever an action containing a `find` call or an `inorder` call is performed, the return values for the operation on the SUT should be compared to the operation on the reference. If there is no `compare:` indicator, return values of calls are not compared, the more typical case for many reference operations (e.g. our AVLTree's `add` does not return the same type as set's `insert`, so comparison is not possible).

2.5 Using the Harness

In order to test `avl.py`, a TSTL user would first compile the harness by typing: `tstl avl.tstl`. The `tstl` tool takes several command line arguments, such as the class and Python file to output, and whether to include code coverage instrumentation, but by default compiles the given file into `sut.py` and makes a class called `sut`. The self-contained file `sut.py` produced from our example harness is over 7,000 lines of Python code (this includes actual testing code plus support code common to all TSTL-generated harnesses). If the random tester in Figure 2 is in a file called `rt.py` the user can then test the AVL implementation (or any other SUT compiled into `sut.py`) by typing: `python rt.py`. If the AVL tree code has a bug, it is likely the tool will quickly detect it and print out a short test case that exposes the problem. The TSTL release includes some simple testers, including a much more configurable random tester than is shown here, a rudimentary model checker, and some artificial-intelligence based algorithms for testing (a variant of beam search [9], for example).

3. RELATED WORK

To our knowledge, there has been no previous tool providing similar functionality. There exist various testing tools and languages of a somewhat different flavor: e.g. Korat [13], which has a much more fixed input domain specification, or the tools built to support the Next Generation Air Transportation System (NextGen) software [3]. The closest of these is the UDITA language [4], a Java extension with non-deterministic choice operators and `assume`, which yields a different language but that shares some of TSTL's goal. TSTL aims more at the *generation* of tests than the *filtering* of tests (as defined in the UDITA paper), while UDITA supports both approaches. This goal of UDITA (and resulting need for first-class `assume`) means that it is hosted inside a complex tool, JPF [15], rather than generating a stand-alone simple interface to an SUT. Building a version of UDITA for a new language is far more challenging than porting TSTL. UDITA also supports many fewer constructs to assist test harness development; e.g., users must implement their own logging, delta debugging, etc.. Work at JPL on languages for analyzing spacecraft telemetry logs [7] is another example of a working example of a Python-based declarative language for testing purposes. The pool approach to test case construction is derived from work on canonical forms and enumeration of unit tests [1].

4. CONCLUSIONS AND FUTURE WORK

TSTL is currently a useful working prototype, capable of production-quality testing. We do expect that the experience of more users will introduce some changes in the language and interface. Our hopes for TSTL are threefold. First, we hope that TSTL can provide an easy way for non-academic developers to use automated testing in their own work, without much effort, and to take advantage of the work of testing researchers, who will provide TSTL tools for their testing algorithms. Second, we hope that TSTL can serve as a way for students to learn about automated testing, since Python is now one of the most popular introductory languages in computer science [14]. Finally, TSTL can provide a way for researchers to rapidly prototype new testing algorithms, and apply them with ease to any SUTs for which a TSTL harness has been defined. Writing meta-analysis tools to automatically apply multiple algorithms (or the same algorithm with different parameters) to TSTL SUTs and statistically analyze the results is fairly easy, and we have some rudimentary scripts for this purpose already in place in our own environment.

The next step for TSTL is to move beyond Python to provide testing support for other languages. Scala, Ruby, and other scripting languages are obvious targets (we believe a Scala implementation would be a relatively easy effort, and perhaps allow integration with sophisticated log analysis tools [2]). Java, C, and C++ may require somewhat more complex implementations, but should also be possible. Finally, we believe TSTL can also target harnesses for model checkers such as CBMC [11] and SPIN [10].

Acknowledgments: The authors would like to thank the students (especially Francis Vo) in winter term 2015 sections of CS 362 and 562 at Oregon State University for their comments, contributions, and complaints about TSTL. A portion of this research was funded by NSF CCF-1217824 and NSF CCF-1054876.

5. REFERENCES

[1] 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.

[2] H. Barringer and K. Havelund. TRACECONTRACT: a Scala DSL for trace analysis. In *Formal Methods*, pages 57–72, 2011.

[3] D. Giannakopoulou, F. Howar, M. Isberner, T. Lauderdale, Z. Rakamarić, and V. Raman. Taming test inputs for separation assurance. In *International Conference on Automated Software Engineering*, pages 373–384, 2014.

[4] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *International Conference on Software Engineering*, pages 225–234, 2010.

[5] 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*, pages 243–252. IEEE, 2014.

[6] A. Groce and M. Erwig. Finding common ground: choose, assert, and assume. In *Workshop on Dynamic Analysis*, pages 12–17, 2012.

[7] A. Groce, K. Havelund, and M. Smith. From scripts to specifications: The evolution of a flight software testing effort. In *International Conference on Software Engineering*, pages 129–138, 2010.

[8] 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.

[9] A. Groce and J. Pinto. In *NASA Formal Methods Symposium*, 2015. To appear.

[10] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[11] D. Kroening, E. M. Clarke, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[12] W. McKeeman. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.

[13] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A tool for generating structurally complex test inputs. In *International Conference on Software Engineering*, pages 771–774, 2007.

[14] E. Shein. Python for beginners. *Communications of the ACM*, 58(3):19–21, 2015.

[15] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, Apr. 2003.

[16] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294, 2011.

[17] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.