International Journal on Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

TSTL: The Template Scripting Testing Language

Josie Holmes · Alex Groce · Jervis Pinto · Pranjal Mittal · Pooria Azimi · Kevin Kellar · James O'Brien

Received: date / Accepted: date

Abstract A test harness, in automated test generation, defines the set of valid tests for a system, as well as their correctness properties. The difficulty of writing test harnesses is a major obstacle to the adoption of automated test generation and model checking. Languages for writing test harnesses are usually tied to a particular tool and unfamiliar to programmers, and often limit expressiveness. Writing test harnesses directly in the language of the Software Under Test (SUT) is a tedious, repetitive, and error-prone task, offers little or no support for test case manipulation and debugging, and produces hard-to-read, hard-to-maintain code. Using existing harness languages or writing directly in the language of the SUT also tends to limit users to one algorithm for test generation, with little ability to explore alternative methods. In this paper, we present TSTL, the Template Scripting Testing Language, a domain-specific language (DSL) for writing test harnesses. TSTL compiles harness definitions into an interface for testing, making generic test generation and manipulation tools for all SUTs possible. TSTL includes tools for generating, manipulating, and analyzing test cases, including simple model checkers. This paper motivates TSTL via a large-scale testing effort, directed by an end-user, to find faults in the most widely used Geographic Information Systems tool.

Josie Holmes
Department of Geography
Pennsylvania State University
E-mail: jdh396@psu.edu

Alex Groce, Jervis Pinto, Pranjal Mittal, Pooria Azimi, Kevin Kellar
School of Electrical Engineering and Computer Science
Oregon State University
E-mail: agroce@gmail.com

James O'Brien
Risk Frontiers
Macquarie University
E-mail: James.OBrien@mq.edu.au

This paper emphasizes a new approach to automated testing, where, rather than focus on developing a monolithic tool to extend, the aim is to convert a test harness into a language extension. This approach makes testing not a separate activity to be performed using a tool, but as natural to users of the language of the system under test as is the use of domain-specific libraries such as ArcPy, NumPy, or QIIME, in their domains. TSTL is a language and tool infrastructure, but is also a way to bring testing activities under the control of an existing programming language in a simple, natural way.

Keywords Software testing · Domain-specific languages · Explicit-state model checking · End-user testing · Geographic Information Systems

1 Introduction

Software test automation encompasses two challenges: (1) automated execution and determination of results for human-created tests, and (2) truly automatic generation of tests. Both are critical for effective, efficient software testing, but only test generation offers the potential to discover faults without human determination that a particular execution scenario has the potential to behave incorrectly. Automated generation of tests relies on the construction of *test harnesses*. A *test harness* defines the set of valid tests (and, usually, a set of correctness properties for those tests) for the Software Under Test (SUT). This paper presents a language and tools applying insights from the world of explicit-state model checking to the problem of producing test harnesses for automated test generation, whether tests are produced by a exhaustive state-space exploration as in model checking, or via less systematic methods.

Building a test harness is a task that even experts in model checking and automated testing often find painful [52,45]. The difficulty of harness generation is one reason for the limited adoption of automated testing and model checking methods by the typical developer who writes unit tests. This is unfortunate, as even simple random testing can often uncover subtle faults.

The "natural" way to write a test harness is as code in the language of the SUT. This is obviously how most unit tests are written, as witnessed by the proliferation of tools like JUnit [36] and its imitators (e.g., PyUnit, HUnit, etc.). It is also how many industrial-strength random testing systems are written [50,48]. A KLEE "test harness" [18] for symbolic execution is written in C, with a few additional constructs to indicate which values are symbolic. This approach is common in model checking as well: e.g., Java Pathfinder [3, 93] can easily be seen as offering a way to define a state space using Java itself as the modeling language, and CBMC [68,69] performs a similar function in C, using SAT/SMT-based bounded model checking instead of explicit-state execution. JPF in particular has shown how writing a harness in the SUT's own language can make it easy to perform "apples to apples" comparisons of various testing/model checking strategies [94].

Unfortunately, writing test harnesses this way is a highly repetitive and error-prone programming task, with many conceptual "code clones" (e.g. Fig-

ure 1). A user faces difficult choices in constructing such a harness. For example, the example harness always assigns `val2` even though `call1` only uses `val1`, to avoid having to repeat the choice code for calls 2 and 3. The harness is almost certainly sub-optimal for random testing, where the lack of any memory for previously chosen values can make it hard to exercise code behaviors that rely on providing the same arguments to multiple method calls (e.g., `insert` and `delete` for container classes). The construction of a harness becomes even more complex in realistic cases, where the tested behaviors involve building up complex types as inputs to method calls, rather than simple integer choices. For example, consider the problem of testing a complex Python library. Figure 2 shows a portion of the Python documentation for one function in the ArcPy [31] site package for Geographic Information Systems (GIS) automation. Rather than taking a single integer, this function call requires complex inputs — a feature class or layer, an SQL expression, and other complex types that we can assume are also difficult to construct. A harness testing a typical real-world library must manage the creation of values of many such complex types. Moreover, because building up function inputs is itself complicated and requires complex method calls, these values cannot simply be produced on each iteration, but must be stored and selected for use in future calls. The code quickly becomes hard to read, hard to maintain, and hard to debug. In some cases [48] the code for a sophisticated test harness approaches the SUT in complexity and even size! The code's structure also tends to lock in many choices that would ideally be configurable.

One of the most important of these locked-in choices is the test generation method. Writing a harness by hand usually makes it hard to try out new strategies. Writing novel testing strategies in even such an extensible platform as Java Pathfinder is hardly a task for the non-expert. The harness in Figure 1 may support random testing and some form of model checking, if it is written in Java and can use JPF or a library for adaptation-based testing [47]. Such a harness will likely be completely inflexible as to generation method if written in Python, C, or another language without that level of tool support.

What the user really wants is to simply provide a concise version of the information in Figure 2, some configuration details (e.g., how many feature classes to keep track of at once), and then try different test generation methods. While some automated testing tools for Java [35, 80] can automatically extract method signatures from source code and produces tests, using such a tool locks a user into one test generation method. Completely automatic extraction

```
op = choice(operations);
val1 = choice(values);
val2 = choice(values);
if (op == op1 && guard1) {
    call1(val1);
} else if (op == op2 && guard2) {
    call2(val1,val2);
} else if (op == op3 && guard3) {
    call3(val1,val2);
...
```

Fig. 1 A test harness in the SUT's language.

```
MakeFeatureLayer_management(in_features, out_layer, where_clause, workspace, field_info)
   Creates a feature layer from an input feature class or layer file. The layer
   that is created by the tool is temporary and will not persist after the session
   ends unless the layer is saved to disk or the map document is saved.
INPUTS:
 in_features (Feature Layer):
   The input feature class or layer from which to make the new layer. Complex
   feature classes, such as annotation and dimensions, are not valid inputs to this tool.
 where_clause {SQL Expression}:
   An SQL expression used to select a subset of features. For more information on
   SQL syntax see the help topic SQL reference for query expressions used in ArcGIS.
...
```

Fig. 2 Documentation for a function in Esri's ArcPy site package.

also often fails to handle the subtle details of harness construction, such as defining guards for some operations, or temporal constraints between API calls that are not detectable by simple exception behavior. Understanding problems with automatic extraction can be hard with large libraries, since the extraction tends to either produce internal data structures only or produces a huge, impenetrable mass of code. The user *wants* a declarative harness, but often *needs* to program critical details of a harness, and build understanding of the system by performing harness development in small, incremental steps.

1.1 Contributions

In this paper we describe a complete, Domain Specific Language (DSL)-based approach that combines a simple means to produce a declarative harness with the full power of a complete programming language. TSTL (the Template Scripting Testing Language) compiles a declarative description of system state and actions into a library in the language of the System Under Test (SUT). This library allows the creation of objects providing an API for testing the SUT, including support for state comparison, abstraction, backtracking, automatic test case reduction, code coverage, and support for sophisticated regression testing.

Using an ongoing case study, we show how to apply TSTL and its tool suite to a large, real-world software library used in critical applications. The test effort has been driven and directed not by a software testing researcher (as is the usual case), but by a domain expert in the Geographic Information Systems (GIS) SUT. In the course of this effort, multiple faults and undocumented restrictions of the library under test have been discovered, and the TSTL language and tool suite have been transformed from a research prototype into a complete system for software testing.

This paper presents the most complete presentation of the TSTL language and tools, and we hope that it satisfies three critical goals:

– First, would-be users wanting to take advantage of automated test generation should be able to base their own testing efforts using TSTL on the example code in this paper (and that available in the TSTL github

repository [56]). This paper thus completely describes the concepts behind TSTL, the semantics of the language, and the tools available in TSTL. Previous papers on TSTL [54, 55] reported a much less full-featured version of the language using a difficult-to-read syntax.
- Second, researchers should be able to use the information in this paper to extend existing TSTL tools or build their own tools to explore novel test generation strategies, automated debugging methods, and other research prototypes. TSTL enables easy comparison of methods in a framework reducing the burden of implementation and avoiding irrelevant differences in performance due to underlying infrastructure. The growing set of SUTs included in the TSTL distribution, which includes large and widely used Python libraries, can provide benchmarks for experimental efforts.
- Finally, unlike previous publications on TSTL, this paper emphasizes the fact that TSTL, unlike other testing DSLs or tools, at heart transforms a definition of valid tests (and properties) for a System Under Test into a *programming language interface* for testing that system. Tests in TSTL are not inaccessible entities internal to a tool, or only represented as unit tests (i.e., programs) that cannot be easily manipulated and analyzed, but first-class objects in the language of the System Under Test. To our knowledge, this approach to testing has not been previously explored, and it was not emphasized (or even clearly presented) in earlier publications on TSTL.

The organization of this paper is as follows. In Section 2 we present the basic idea of a DSL for testing, and distinguish TSTL from other testing DSLs. Section 3 provides background on the ArcPy GIS case study used throughout the paper. Section 4 provides a full description, with examples, of the core TSTL language and semantics. Section 5 describes the tools included with TSTL, and Section 5.5 describes how researchers and developers can build their own TSTL-based testing tools to support additional testing, debugging, or regression strategies. Section 6 introduces the novel TSTL concept of making testing a first-class activity in a programming language, similar to how other libraries make GIS (ArcPy), scientific computing (NumPy [4], SciPy [6]) or bioinformatics (QIIME [19], Biopython [2], scikit-bio [5]) activities simple to use in either a scripted or interactive manner. Faults discovered in TSTL, in ArcPy and other systems, are described briefly in Section 7. We survey the most closely related work in Section 8, and summarize our conclusions in Section 9.

2 Domain Specific Languages for Testing

The nature of test harness construction suggests the use of a *domain-specific language* (DSL) for testing [46]. DSLs [34] provide abstractions and notations to support a particular programming domain. The use of DSLs is a formalization of the long-standing approach of using "little languages," as advocated by Jon Bentley in a Programming Pearls column [15] and exemplified in such

system designs as UNIX. DSLs typically come in two forms: *external* and *internal*. An external DSL is a stand-alone language, with its own syntax. An internal DSL, also known as a domain-specific embedded language (DSEL), is hosted in a full-featured programming language, restricting it to the syntax (and semantics) of that language. Many attempts to define harnesses can be seen as internal DSLs [39,47,93,69,18]. Neither of these choices is quite right for test harnesses. Simply adding operations for nondeterministic choice still leaves most of the tedious work of harness definition to the user, and makes changing testing approaches difficult. With an external DSL, the user must learn a new language, and the easier it is to learn, the less likely it is to support the full range of features needed.

A novel approach is taken in recent versions of the SPIN model checker [63]. Version 4.0 of SPIN [61] exploited the fact that SPIN works by producing a C program from a PROMELA model to allow users to include calls to the C language in their PROMELA models. The ability to directly call C code makes it much easier to model check large, complex C programs [48,62]. C serves as a "DSEL" for SPIN, except that, rather than having a domain-specific language inside a general-purpose one, here the domain-specific language hosts a general-purpose language. A similar embedding is used in `where` clauses of the LogScope language for testing Mars Science Laboratory software [49]. We adopt this approach for our own language and embed the general-purpose language (for expressiveness) in a DSL (for concision and ease-of-use).

The most significant difference between TSTL and other DSLs for testing and verification, including SPIN, is that most such systems are primarily intended to be used as stand-alone tools. Whether model checkers [63,93], model-based testing tools [92], or random testing tools [80], these systems are primarily designed as "things to *run on* the system under test." TSTL can operate in this manner, but at heart it transforms a definition of valid tests into a library for creating, executing, manipulating, and analyzing test cases. An experienced TSTL user can interact with TSTL at an interactive command prompt in the language of the SUT, creating, saving, and modifying tests on-the-fly. TSTL tools are simply scripted formalizations of this mode of use, automating repetitive tasks. Such an approach is not possible with any other tool of which we are aware. Many tasks that are constrained to the functionality provided by tools included in other systems (e.g., replay of regression tests) in TSTL are simplified and made flexible by this approach.

2.1 TSTL: The Template Scripting Testing Language

TSTL is based on understanding a test harness as a declaration of the possible actions the SUT can take, where these actions are defined in the language of the SUT itself, with the full power of the programming language to define guards, perform pre-processing, and implement oracles. Our particular approach is based on what we call *template scripting*.

```
@from arcpy import *

pools:
  <fc> 3 CONST             # A feature class contains only lines, points, or polygons
  <newlayer> 3 CONST
  <op> 2 CONST
  <val> 2 CONST
  <whereclause> 2 CONST    # SQL clause to limit objects in new layers
  <fieldname> 2 CONST      # Extracted from the shape files
  <fieldlist> 2

actions:

<fc> := <["d1.shp", "d2.shp", "d3.shp"]>  # Just shapefiles for this example
<newlayer> := <["newl1", "newl2", "newl3"]>
{IOError} <fieldlist> := ListFields(<fc>) # Extract fields from a feature class
len(<fieldlist,1>) >= 1 -> <fieldname> := <fieldlist> [0].name
<fieldlist> = <fieldlist> [1:] # Skip to next field
<op> := <[">", "<", "<=", ">=", "=", "!="]>
<val> := <1..10>
<val> = <val> * 10
<val> = <val> + 1
<whereclause> := '"' + <fieldname> + '" ' + <op> + str(<val>)
<whereclause> = <whereclause> + ' AND ' + <whereclause>
<whereclause> = <whereclause> + ' OR ' +  <whereclause>
<whereclause> = 'NOT' + <whereclause>
{ExecuteError} MakeFeatureLayer_management(<fc>,<newlayer>)
{ExecuteError} MakeFeatureLayer_management(<fc>,<newlayer>,where_clause=<whereclause>)
```

Fig. 3 A small TSTL file to test one ArcPy function.

The *template* part of the name captures the fact that our method proceeds by processing a harness definition file to output code that enables testing, much as SPIN processes PROMELA/C. The harness description file consists of fragments of code in the SUT's language that are expanded, via the TSTL compiler, into a class that allows an independently written test generation or manipulation tool to generate, execute, or replay tests, without knowing any details of the SUT. A TSTL harness defines a *template* for action definition, and the compiler instantiates the template exhaustively. The *scripting* aspect indicates TSTL is designed to be very lightweight and as easy for users to pick up as a popular scripting language. TSTL works best when the SUT language is very concise, like most scripting languages, making "one-liners" of action definition possible; our initial implementation [56] is therefore for Python[1].

Figure 3 shows a simple TSTL harness for the function documented in Figure 2. Even this short harness supports constructing SQL where clauses of arbitrary length and selecting field names based on data files. Figure 4 shows a simple pure random test generator that can test any SUT (including this one) with a TSTL-defined harness. This harness, in 20 lines of code, not only provides automated test generation, but continuous reporting of incremental branch coverage, delta-debugging [96] for reduction of failing tests, and additional TSTL-specific post-processing that further reduces the size and complexity of test cases for debugging. The brevity of the test generator, no

[1] We also have released a beta version of TSTL for Java [65], to show that testing code in non-scripting languages is also possible.

```
import sut, random, time
rgen = random.Random()
sut = sut.sut()
NUM_TESTS = 1000
TEST_LENGTH = 200
for t in xrange(0,NUM_TESTS):
   sut.restart()
   for s in xrange(0,TEST_LENGTH):
       action = sut.randomEnabled(rgen)
       r = sut.safely(action)
       if len(sut.newBranches()) > 0:
          print time.time(),'NEW BRANCHES:', sut.newBranches()
       if (not r) or (not sut.check()):
          pred = sut.failsCheck if r else sut.fails
          print 'TEST FAILED:', sut.error()
          R = sut.reduce(sut.test(), pred)
          N = sut.normalize(R, pred)
          sut.generalize(N,pred)
```

Fig. 4 A simple random tester using the interface provided by TSTL.

matter how complex the SUT, is made possible by the common functionality of all TSTL-generated testing interfaces. The TSTL compiler produces a Python (or other target language) class that allows a test generation or manipulation tool to view a testing problem as exploration of a (possibly infinite) graph of states. Transitions in the graph are the available test actions, executed in the underlying language, and are guarded by both TSTL restrictions on the semantics of valid tests and user-defined guards on system behavior. States include both the (possibly unknown) state of the SUT and the TSTL state, including pools of values to be used in actions.

3 Motivating Case Study: Esri ArcPy

Esri is the single largest GIS software vendor, with about 40% of global market share. Esri's ArcGIS tools are extremely widely used for GIS analysis, in government, scientific research, commercial enterprises, and education. Automation of complex GIS analysis and data management is essential, and Esri has long provided tools for programming their GIS software tools. The newest such method, introduced in ArcGIS 10.0, is a Python site-package, ArcPy [31]. ArcPy is a complex library, with dozens of classes and hundreds of functions distributed over a variety of of toolboxes. Most of the code executed in carrying out ArcPy functions is the code for the ArcGIS engine itself. This source code, written in C++ (amounting to millions of lines), is not available. The source code for the latest version (10.3) of the Python site-package alone, however, which interfaces with the ArcGIS engine, is over 50,000 lines of code. This is a very large system (especially given the compactness of Python code), comparable in size to the largest software systems previously tested using automated test generation, such as core Java and Apache libraries [35, 80].

In order to improve the reliability of ArcPy, we are developing a framework for automated testing of ArcPy itself, as well as libraries based on ArcPy. The TSTL harness for ArcPy is already more than six times as large as the

next-largest such definition previously implemented in TSTL, even though the harness so far only includes a small portion of ArcPy API (Application Program Interface) calls. The first stage of testing has resulted in discovery of multiple faults in ArcPy/ArcGIS, and has required modifications to the TSTL language and, especially, to the tool chain supporting test replay, debugging, and test case understanding.

Previous work on automated test generation for APIs has been largely carried out by software testing researchers only, or (at most) by software testing researchers working with individuals who are primarily software developers. This paper describes TSTL in the context of a testing effort largely directed (and coded) by the first author, who is not a software developer by profession or education, but a GIS analyst. The problem of end-user testing [16, 17, 83] is long-standing. Previous work in the field has often focused on non-traditional programming: e.g. spreadsheets [83], visual languages, or machine-learning systems [53]. TSTL is partly designed to allow a user who is familiar with a software library but not expert in software testing techniques to test a traditional software API library. In one sense, this is a less difficult scenario than spreadsheets or visual forms, in that the testing is directed by an individual used to writing and thinking about code. The concepts in automated software testing are most easily understood by those who are also familiar with a conventional programming language. On the other hand, ArcPy is not a small user-developed program but a large, complex system. ArcPy was also not written by the end-user, or by any of the authors of this paper, nor have the authors received any assistance in the effort from Esri.

Automated testing systems more advanced than a simple hand-written loop generating a few random inputs to a handful of functions, or more complicated to use than a fully push-button system are often considered too difficult for practical use even by software developers or software QA staff [47]. Even "push-button" tools for automated testing are sometimes difficult for expert users to install, apply, and configure [48, 51, 47]. TSTL aims to be relatively easy to use for anyone familiar with basic Python development. By avoiding the use of a toy problem and presenting TSTL in the context of a more typical real-world system (vs. e.g., a simple container class), we hope to make it easier to apply to other real-world systems.

4 The TSTL Harness Language

The TSTL compiler takes as input a harness template file, and produces as output a Python class file that implements an interface other tools (or even users working interactively) can use to perform testing on the SUT via an SUT-independent interface.

The harness in Figure 2 shows many of the basic features of TSTL. The basic structure of a TSTL harness consists of three parts, usually written in order. First, harness code prefixed by an `@` or enclosed in `<@ @>` is treated as raw Python code, and essentially not interpreted by the TSTL compiler.

This code is reproduced almost literally in the output file[2]. Second, there is a preamble that almost always defines a set of *value pools* for use in testing, but also may include information on logging, correctness properties, source code locations for code coverage analysis, and other basic information that applies to the entire harness. Finally, the bulk of a TSTL harness (and the only non-optional element) is a set of *action definitions*. Actions are the possible steps to be taken in testing, and define the set of possible tests.

The original version of TSTL [54] required cumbersome use of Python functions to implement many simple operations, including guards. Current TSTL extends the language to make it possible to define very complex test spaces using only pools and actions, with helper functions only required for the usual reasons of abstraction and readability.

4.1 The Essentials of Pools and Actions

In TSTL, tests usually consist of assignments to value pools and function calls making use of those values. These are the most common forms of actions. Value pools are meta-variables in the target language, and support the complete set of types of the underlying language. A pool can contain simple types such as integers, or more complex types such as functions, container classes, file handlers, or even TSTL testing objects (to support testing TSTL itself). The notion of an action in TSTL is similiarly completely general: *any fragment of code in the host language can be an action.* Usually, it is most convenient to encapsulate complex actions by defining functions that perform the desired behavior, and making the action a call to such a function, but this is not required. As Andrews et al. have shown [9], this pool-and-action approach is sufficient to express the full generality of unit tests, in any language[3].

In order to make the core ideas clear, consider part of the harness shown in Figure 2, defining how to generate values used in SQL where clauses. The following, by itself, is a valid TSTL harness (albeit one that cannot discover any faults, since it performs no actions beyond simple integer addition):

```
pools:
  <val> 2 CONST

actions:

<val> := <1..10>
<val> = <val> + 1
```

There is only one pool, named `val` (optionally labeled as `CONST` to indicate that its value does not change unless it appears on the left hand side of an assignment). The pool has room to store two values. The state of the SUT is defined by the state of all pools. Initially, all pools are set to a special value (`None`) indicating the pool has not been initialized. For the most part, we can

[2] TSTL does have to scan `import`s to re-load modules, and also pre-processes function definitions to support pre- and post-conditions.

[3] In fact, TSTL tests are somewhat more general than this already very general and expressive form, in that we do not disallow loops and conditions in actions.

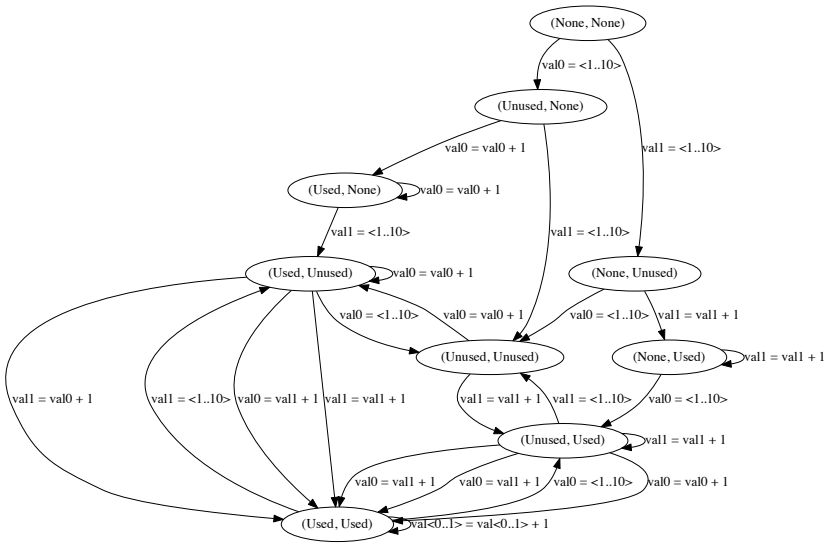

Fig. 5 Constraints on actions in a test, based on pool states

think of the `val` pool as two Python variables `val0` and `val1`. Another way to think about a pool is as a kind of informal "named type," with a limited set of variables that can contain the "type" and all possible action sequences that assign to its pool values serving as its specification. In Python it is not necessary to specify the actual type of a value pool, though an optional `: type` notation enables TSTL to perform runtime type-checking and ensure pools never contain incorrect types.

In this simple example, the only actions are initialization of a `val` and incrementing a `val`. Again, we emphasize that an action can in general be an arbitrary Python statement. Actions that include the `:=` form of assignment (a TSTL, not Python, operation) initialize pool values. When `<val>` appears in an action, that represents all possible pool values with that name: for our simple example, either `val0` or `val1`. An integer range is represented by `<i..j>`, and TSTL expands such ranges to produce an action with each possible choice. The first line in the actions section of this harness translates to 20 different possible actions:

```
val0 = 1
val0 = 2 ...
val0 = 10
val1 = 1 ...
val1 = 10
```

From the initial state of the system, only these 20 actions are *enabled*. *Enabled* actions are those that can be executed in the current state; the complete set of actions defined by a TSTL harness is always finite, and the enabled set

is always a subset of that finite set. The first concept that is essential to understanding TSTL semantics is that at any state of the system, the only actions that are enabled are those that do not *use* any non-initialized pool values. Any appearance of a pool value is considered a *use*, with the single exception of the left-hand-side of a `:=` initialization (not normal assignment)[4]. The second concept is that a value that has been initialized cannot be initialized (appear on the lhs of `:=`) until after at least one action that uses it has been executed. Figure 5 shows the consequences of these rules for the simple value assignment harness above. The nodes in the graph are labeled with (`state(val0), state(val1)`), where state is either `None` (uninitialized), `Unused` (initialized but never used) or `Used` (initialized and used at least once). Starting from the initial state `(None, None)`, a valid test is any path through the graph.

Tests that can be produced by this harness include, therefore, sequences like `val0 = 3; val0 = val0 + 1; val1 = 4; val1 = val0 + 2` and `val1 = 10; val0 = 6; val0 = val1 + 1; val0 = 2; val1 = 15`. However, `val0 = val0 + 1; val0 = 2` and `val0 = 1; val1 = 1; val1 = 4` are not valid tests, because they either use an uninitialized pool value, or re-initialize an unused pool (a clearly useless action sequence).

4.2 Other Core Language Features

The example TSTL harness in Figure 2 shows a few other important core elements of TSTL. First, choice templates are not limited to integer ranges, but can include arbitrary items in a list, e.g, `<fc> := <["d1.shp", "d2.shp", "d3.shp"]>`. Note that while in the example these items are (string) constants, they can be arbitrary expressions to be computed at runtime, or even incomplete code fragments that are only valid when combined with the rest of the action. Second, when an action raises an uncaught exception, this is normally considered a test failure. Prefixing an action with a set of exception names in curly braces (e.g., {`IOError`}) indicates that some exceptions are expected, and do not indicate a failure.

More critically, actions can also be prefixed by arbitrary guards, using the syntax `guard -> action`. The simple ArcPy harness chooses field names for SQL by first extracting a list of all fields in some feature class. It then allows a field name to be chosen by taking the name of the first field in the list. However, since the harness also allows the list of fields to be stepped through by discarding the initial element, the name extraction has to be guarded to ensure that tests won't try to extract names from an empty field list: `len(<fieldlist,1>) >= 1 -> <fieldname> := <fieldlist> [0].name`. The `<fieldlist,1>` construct, which can also be used outside of a guard, indicates that this pool value should not be produced using normal template expansion (instantiated as both `fieldname0` and `fieldname1`) but rather that it should copy (textually, not a copy of the object but the same variable use) the comma indexed appearance

[4] The definition of use is the only distinction between `:=` and normal Python assignment; `:=` is implemented as Python assignment, and appears as such when test cases are printed.

of that pool in each expansion (indexing starts from 1). This makes sure the guard is over the same pool value that is used in the action.

TSTL also supports post-conditions on actions, in the form `action => post-condition`, where the post-condition is checked after the action is performed. For example, because some known ArcPy bugs involve addition of incorrect characters to field names in a database, we could add code to check that field names in feature classes never change from their initial values. We can make sure that a library call to add a field to a feature class adds it to a database of all field names collected at the start of testing, and collect the set of fields in each feature class file at the beginning of each test, storing these in a dictionary. This example code shows two more features of TSTL: TSTL supports `init:` code in the preamble, which is called before each test starts. Ending a line in a backslash indicates the action continues on the next line of the file.

```
init: <fieldnames> = getAllFieldNames(getFeatureClasses())
{ExecuteError} not (<fc,1> in <hascursor>) -> \
   AddField_management(<fc>,<fieldname>,<fieldtype>); \
   <fieldnames> [<fc,1>].add([<fieldname,1>])
{IOError} <fieldlist> := ListFields(<fc>) \
  => sorted(<fieldlist,1>) == sorted(list(<fieldnames> [<fc,1>]))
```

Note the additional guard on adding fields — we have discovered that adding a field to a feature class that has any database cursors active tends to crash ArcPy. For more complicated post-conditions, the construct `pre<(expr)>` allows access to values of expressions from before the action was executed, as a further convenience for expressing properties.

When an assertion is an invariant on all post-action states, it can be included in the preamble. To check field names we would write `property: sorted(ListFields(<fc>))==sorted(list(<fieldnames> [<fc,1>]`.

This property checks all feature classes, not just those whose whose fields are extracted. The advantage is that the property will catch problems even if we never construct a `fieldlist`; the disadvantage is that testing slows to check all field names for all feature classes, after every action.

Another useful feature of TSTL is the ability to create *reference* pools, where every action on pool values is mirrored by an action on a reference version of that pool. This makes it possible to perform differential testing [76] on a per-pool basis, rather than at the whole-system level, allowing complex partial specifications. For example, in ArcPy we may want to ensure that operations are deterministic: no GIS operations produce different results, given the same underlying starting feature class data. Assuming in raw Python in the preamble we have defined `identityFunction` as an identity function and `copyFCName` as a function that takes a feature class name and transforms it into a generated name for a reference copy of the feature class, the following mirrors all actions on feature classes on a reference copy, and checks that the feature class and its reference always have the same fields.

```
pools:
  <basefc> 2 CONST
  <fc> 2 CONST REF
<basefc> := <[‘‘d1.shp’’, ‘‘d2.shp’’, ‘‘d3.shp’’]>; \
  CopyFeatures_management(<basefc,1>,copyFCName(<basefc,1>)
<fc> := identityFunction(<basefc>)
{IOError} <fieldlist> := ListFields(<fc>)
references:
  identityFunction ==> copyFCName
compares:
  ListFields
```

When instantiating the action templates, TSTL always produces a copy of every action containing any reference pool values. First, the pool values are replaced with their reference copies; second, all the syntactic transformations (which can include arbitrary Python regular expressions) in the `references` declaration are applied. Finally, if any string matches a regular expression in a `compares` declaration, the return values or assigned values in the action are compared with those for the reference version. In the ArcPy case, if our `copyFCName` is correctly defined, we can even check that behavior is equivalent for different underlying data file formats for feature classes.

4.3 How to Build a TSTL Harness

In the introduction, we noted that one problem with automatic extraction of harnesses by testing tools is that in order to effectively test complex systems, it is important to incrementally build testing capability. Often, as with software development, understanding the effort as it slowly increases in scope is essential. TSTL naturally supports this methodology. The ArcPy harness, though complex, was developed by starting with a small number of ArcPy functions, and determining their parameters. These functions were chosen because they were involved in unusual or problematic behavior experienced by the authors of the paper. Once functions have been chosen, and their parameters are known, developing a harness can often be a clean, iterative process:

1. Choose a new function (or set of related functions) to include in the harness.
2. Determine all parameter types for these functions.
3. If there is no pool that can produce these types, determine how to produce these types, and add pools and pool initialization actions for those pools. This may require adding some additional functions (in which case, go to step 1 and start with those functions, recursively).
4. Add an action to call the function(s) being added. If relevant, allow any expected exceptions, guards, and post-conditions to check.
5. Run testing, examine code coverage and failures to evaluate the added harness features, and repeat from step 1.

These steps, combined with occasional refactoring or generalization of parts of the harness, can effectively test even a large library, while maintaining tester understanding and control. In the ArcPy test harness development, most of the effort was spent in this cycle, with major exceptions being the

implementation of a method allowing users to provide their own GIS data as a basis for testing, and efforts to improve the TSTL tool infrastructure to support testing a complex application in a Windows environment.

Note that because TSTL defines the structure of a potentially infinite number of tests, users are expected to define correctness of a system via *property-based testing* [23]. As in model checking, the correctness of the system is not specified via users determining the specific output for each test sequence, but by defining general properties over all executions. TSTL includes idiomatic support for common forms of property-based specification, including invariants, post-conditions on actions, and comparison with the outputs of a reference implementation [76].

4.4 TSTL and Other Languages

At heart, the TSTL "language" consists of the syntax and semantics for pools, nondeterministic choice[5], guards, pre-conditions, pre- values, and a few other elements on top of an existing language: the abstract basis of TSTL has a conceptually small footprint.

While the implementation effort to compile to an SUT interface in a different language is considerable, there is no difficulty (beyond engineering effort) in mapping TSTL to languages other than Python. In one summer, an advanced high school student was able to produce a working version for Java [65]. Implementing TSTL for Scala, Ruby, or even C/C++ would require effort but no research breakthroughs. As with the Python and Java TSTL implementations, there is no need (due to the intentional template structure) to even parse the underlying language. The primary development effort is translating the TSTL "runtime" of utility functions to a new language. For Scala, Ruby, or Swift we believe this would be quite trivial. For C/C++ the relative lack of functional language features would be frustrating, but certainly not a blocking difficult. The ideas behind TSTL are abstract and generally applicable, even if the current implementation is built for Python.

5 TSTL Tools

The following tools are provided in the TSTL distribution on github [56]. Installing the TSTL module allows the compiler, called `tstl`, to be used at the command line. Other tools are included in the `generators` and `utilities` directories of the distribution as Python scripts.

[5] Nondeterministic choice [28,75,33,45] is both inherent in the notion of a TSTL action, and represented more concretely by the syntactic sugar of the `<[...]>` notations. Arguably, building the language and semantics around nondeterministic choice to represent a transition system/state space is the core idea of TSTL.

- `depth <int>`: Determines the length of generated tests.
- `timeout <int>`: Determines the maximum time spent generating tests, in seconds.
- `seed <int>`: Determines the random seed for testing.
- `maxTests <int>`: Determines the maximum number of tests to be generated.
- `running`: Produce on-the-fly, time-stamped code coverage information, for analyzing performance of testing algorithms.
- `replayable`: Produce a log of the current test, so even it crashes Python the test can be reproduced, delta-debugged, made stand-alone, or otherwise analyzed.
- `total`: Produce a total log of all test activity, including across resets, for systems where reset is not complete (so tests across resets can be delta-debugged).
- `quickTests`: Produce "quick test" files [43], each containing a minimal test to cover a set of branches of the SUT.
- `normalize`: Apply additional, custom term-rewriting based simplifications of test cases that often further minimize delta-debugged test cases.
- `generalize`: Apply generalization that elaborates each failing test with annotations showing similar tests that also fail.
- `swarm`: Apply swarm testing [58] to test generation.

Fig. 6 Some options for the TSTL random test generator.

5.1 The TSTL Compiler

Given a harness file defined in the language discussed above, the TSTL compiler generates a stand-alone Python class that allows testing of the SUT. This generated code does not depend on the TSTL system being installed, only on any modules the testing itself uses, and on whether code coverage is requested. By default, the compiler produces a class supporting code coverage using the `coverage.py` module, and assumes this is installed. The TSTL compiler also allows a user to control some fine-grained coverage measures (e.g, is coverage measured during initialization and module reloads?), and force a system to use replay-based backtracking by default.

5.2 Test Generators

TSTL comes with a complex, highly configurable, pure random tester (supporting numerous command-line options). The included random tester provides a number of useful options, of which a subset are shown in Figure 6.

To our knowledge, TSTL's random tester is the first general-purpose random testing tool to incorporate the powerful swarm testing [58] algorithm, which has previously been used to test compilers [38,71] and file systems. TSTL's version of swarm testing is more sophisticated than previously published versions, in that it analyzes the dependency graph of TSTL actions to avoid producing degenerate test configurations, improving performance over naive swarm testing. In addition to these stable, commonly used options, the random tester includes novel experimental options, such as the ability to guide random testing by a user supplied Markov model or operational profile [59]. TSTL makes implementing novel test generation methods simple, as discussed below.

The base TSTL tools also use the TSTL interface to support explicit-state model-checking [24,63], using either Depth-First-Search (DFS) or Breadth-

First-Search (BFS) strategies. TSTL uses Python's `deepcopy` tools to provide a simple, easy to use interface for automatically storing and restoring pool states, making these algorithms trivial to implement. There is no fundamental technical difference between performing (theoretically exhaustive) systematic searches of a well-defined transition system or random exploration. Using the same transition system definition for both purposes has considerable advantages, as pointed out in previous work [52], especially for effectively infinite-state systems where random walks will never saturate and exhaustive searches will never complete. TSTL can (unlike SPIN or most explicit-state model checkers) apply BFS or DFS search even to systems without support for backtracking. TSTL's abstract interface to an SUT (transition system) can be configured to provide replay-based simulation of state storage and retrieval. This is required when, for example, a library uses a C extension and so Python's `deepcopy` does not allow full copying of the state of pool objects, or when the system has hidden global state that cannot be captured in a pool value. While state storage and backtracking is usually faster than replay, we find that for some systems the opposite is true, particularly for shallow search depths (which are typical for BFS of a complex system).

TSTL also supports custom abstraction of pools. If a pool is declared with an `ABSTRACT` annotation, the function after the `ABSTRACT` keyword is used to abstract all values for state-matching purposes during exhaustive exploration methods, via the `abstract` function. The `state` method returns the concrete state of the system (since these are required for backtracking), but applying the `abstract` function to this state returns an abstract version of the state to be used in state matching. The core of a BFS, for example, can be expressed quite simply, irrespective of whether the system is using state storage and backtracking or replay (or has an abstraction or not) as:

```
old = sut.state()
for act in sut.enabled():
   sut.safely(act)
   new = sut.state()
   # repr produces a hashable string representation
   absNew = repr(sut.abstract(new))
   if absNew not in visited:
      queue.append(new)
      visited[absNew] = True
      ...
   sut.backtrack(old)
```

This loop iterates through all enabled actions from the current state, and adds any not previously visited to the search queue. Similar code works as the core of a DFS. Implementing heuristic model checking searches is also trivial, whether those searches are SUT/error specific [30] or structural [57].

While not required for any of our testing efforts thus far, encoding temporal logic checking is also simple. A Büchi automata can be encoded in Python, querying the SUT state and action choices to determine transitions. Composing this with the SUT state is trivial. We managed to implement the well-known nested DFS algorithm [25] in less than 40 lines of code, taking the property automata as a tuple input `(initial, trans, accept)`, where `trans: (state, action, sut state) -> state`.

```
import sut
import glob
sut = sut.sut()
failed = 0
total = 0
for testFile in glob('regressions\*.test'):
   t = sut.loadTest(testFile)
   total += 1
   if sut.failsAny(t):
       failed += 1
       print testFile,'FAILED:',sut.failure()
print total,'TESTS EXECUTED'
print failed,'TESTS FAILED'
print len(sut.allBranches()),'BRANCHES COVERED'
print len(sut.allStatements()),'STATEMENTS COVERED'
print sut.report('coverage.txt'),'% STATEMENTS COVERED'
```

Fig. 7 A small script to run stored regression tests

5.3 Utilities for Test Case Manipulation

TSTL provides the tools `sandboxreducer` and `standalone` for manipulating saved test cases produced by the random tester or the simple model checkers. These were developed as part of the ArcPy testing process. ArcPy faults tend to crash the system (this is also the reason the `total` option was introduced), and thus cannot be simplified for debugging inside the test generator. The sandbox reducer takes a testing log and, using subprocesses to handle crashes, produces delta-debugged and normalized test cases. It is also useful to report failing tests not as TSTL's internal test file format, but as standalone Python programs that cause a failure. The `standalone` utility takes a test log or a saved test case and produces a complete, stand-alone Python program that requires neither the generated TSTL interface nor any other TSTL tools.

However, use of these tools is often not required. Figure 7 shows a simple, but complete, Python script for running regression tests generated using TSTL for a system. This script examines the `regressions` directory, replays each test in the directory, and reports on failing tests and code coverage. Such a script can easily be customized to provide different tests for different budgets (e.g., prioritized by time, coverage, or to execute coverage-based 'quick tests').

5.3.1 ArcPy Regression Generation

One difficulty for ArcPy users is ensuring that their existing scripts and tools work on new versions of ArcGIS. Each recent major release (10.2 and 10.3) after ArcPy's introduction has potentially included some changes in the behavior of API calls. Detecting when such changes cause a script to break is difficult. A first step would be an automatic way to find when the return values for calls differ between ArcPy versions. Because installing multiple versions of ArcGIS on the same system is difficult or impossible, our method for finding differences relies on choosing a reference version (10.3 in our current efforts), and generating a set of standalone tests that 1) cover a large amount of ArcPy functionality, including invalid inputs to functions and 2) record the return

values and exceptions raised by calls. These tests can be run on any ArcPy version, and will report differences between the tests and version 10.3. Performing this kind of differential testing [76] on old or new major releases, or across 64 bit and 32 bit versions, is easy. In the long run, we also want to enable TSTL to produce Python 3.0+ code, for use with ArcGIS Pro, which uses Python 3.4 instead of 2.7. This has motivated a branch to TSTL to support Python 3.0 (unfortunately, Python 3.0 is not fully backwards compatible with earlier versions, and Python 2.7 is still the most widely used Python, and non-pro versions of ArcPy only work with 2.7).

We generate coverage-based regression tests using an approach called *quick testing* [43,44], which takes a set of tests produced by random testing or model checking, and applies a test case reduction algorithm [96] to produce smaller tests that have the same code coverage as the very large, highly redundant, original set of test cases. Automatic quick-testing was added to TSTL's random test generator to support ArcPy testing. Combined with standalone test generation, this allows us to produce test cases that can be run on any version of ArcPy, and explore a large variety of behavior of the code. With ArcPy, coverage alone, unlike previous quick testing efforts, is insufficient to ensure a useful regression test. Because coverage only considers the Python behavior of ArcPy (since we do not have access to the source for the ArcGIS engine), it may group behaviors that are not similar together. We added the ability to combine coverage preservation with preservation of all ArcPy messages indicating a successful GIS engine operation, after abstracting away such details as the runtime of the operation, and so forth.

However, just producing these coverage-and-engine-behavior preserving standalone tests is not sufficient for good version comparison, since standalone test cases as produced only check for properties defined in TSTL. An additional option was added to the standalone test generator, allowing it to record the actual return values of all calls, the set of exceptions thrown, the success/failure messages from the ArcPy engine, and so forth to more precisely record a test's behavior on an ArcPy version.

5.4 Visualization of Action Spaces

Understanding the structure of the action graph produced by even a relatively simple TSTL harness can be difficult. The structure is often infinite, and even in cases where there is a finite state space (perhaps introduced by abstraction) the graph is usually far too large for a convenient display. However, we have found that a visual representation of typical trajectories through the system can be very helpful for understanding a complex test system. The `makegraph` utility takes as input a number of traces to produce, a starting depth, additional depth, and a test width. It then produces in pdf form a number of graphs for traces like the one shown in Figure 8. These trace graphs show, in bold, the actual action sequence chosen by a pure random tester, starting after a number of actions not shown (represented by the "..." node) and continuing

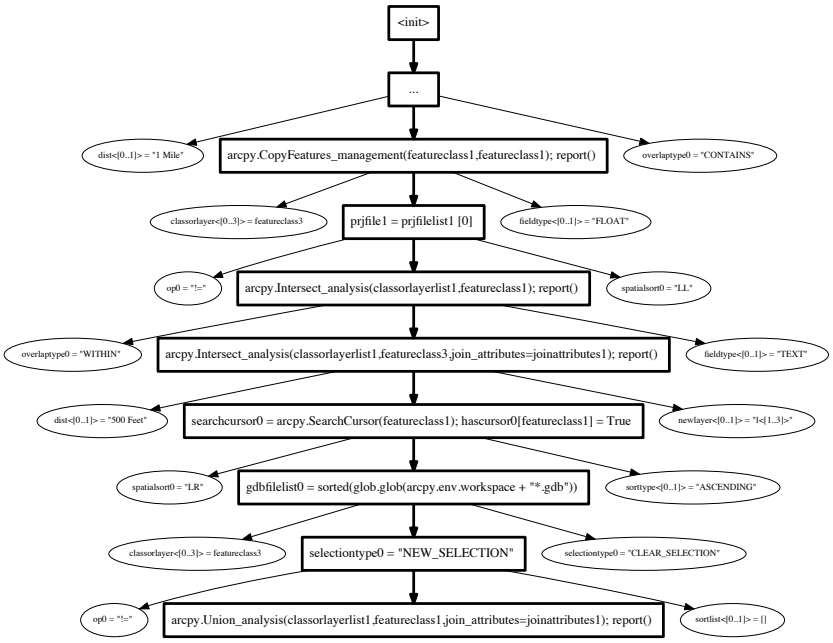

Fig. 8 Start depth 20, depth 8, width 3 trace visualization for ArcPy testing.

up to the depth limit. In addition to the actions taken, the graph also shows a random subset of additional enabled actions, with each step showing a number of actions equal to the width. Because many actions are extremely similar, the graphing utility also summarizes actions that are the same, except for pool choice or integer constant, using the `<[i..j]>` notation of TSTL.

5.5 Building Your Own Testing Tools in TSTL

Describing the full interface provided by TSTL for use in testing tools is beyond the scope of this paper. However, examining the source code of the included testers can provide a good starting point. Implementing new test case manipulations usually involves understanding TSTL internal structures and how tests are stored. Implementing novel test generation algorithms can often rely on just a handful of methods, shown in Figure 9 (TSTL provides nearly 100 methods for generating and manipulating tests, but this minimal set can implement many test generation algorithms).

For example, a researcher aware of the literature showing that for many systems it is difficult to outperform random testing, due to its very low overhead [47,58], may consider simple modifications of random testing that do not greatly increase overhead. One such example, with implementation, is shown

- **restart():** resets the system state and aborts the current test.
- **test():** returns the current test.
- **replay(test):** replays a test, and returns a Boolean indicating success or failure of the test.
- **enabled():** returns a list of all currently enabled actions.
- **randomEnabled(random):** given a Python random number generator object, returns a random enabled action, efficiently (avoiding unnecessary guard evaluations).
- **safely(action):** performs action (usually changing SUT state) and returns a Boolean indicating whether the action performed raised any uncaught exceptions.
- **check():** returns a Boolean indicating whether any properties fail for the current state.
- **error():** returns either `None` (no error for the last action or `check`), or a Python object representing an uncaught exception or failed property's backtrace.
- **state():** returns the current SUT state, as a set of values for all pools; for systems where state cannot be restored by pool values, or `deepcopy` does not work, returns the current test.
- **backtrack(state):** takes a state or test produced by `state` and restores the system to it.
- **reduce(test,predicate):** takes a test and a predicate (function from test -¿ boolean), and returns a (possibly smaller) test also satisfying the predicate).
- **allBranches():** returns the set of branches covered during all testing.
- **newBranches():** returns the set of branches covered during the last action executed that had not previously been covered.
- **currBranches():** returns the set of branches covered during the current test.
- **saveTest(test, filename):** saves a test in a file.
- **loadTest(filename):** loads a test from a file (and returns that test as the function's return value).

Fig. 9 Some core methods for testing an SUT.

```
goodTests = []
startTime = time.time()
while (time.time() - startTime <= TIMEOUT):
   if (len(goodTests) > 0) and (rgen.random() < PEXTEND):
     sut.backtrack(rgen.choice(goodTests)[1])
   else:
     sut.restart()
   for s in xrange(0,TEST_LENGTH):
      action = sut.randomEnabled(rgen)
      r = sut.safely(action)
      if len(sut.newBranches()) > 0:
         print time.time(),len(sut.allBranches()),'NEW BRANCHES:', sut.newBranches()
   if MEMORY > 0:
      goodTests.append((sut.currBranches(), sut.state()))
      goodTests = sorted(goodTests, reverse=True)[:MEMORY]
```

Fig. 10 Implementing a very simple novel testing algorithm.

in the original TSTL paper [54]. We present another here. Since the focus of this paper is on showing how to use TSTL, not novel test generation methods, we leave a complete development and statistically valid evaluation of our proposed approach to future work, but discuss briefly how to go about prototyping and evaluation using TSTL.

The idea is to perform random testing, but keep the final state of tests with unusually high coverage as potential starting points for future tests, potentially extending them far beyond the normal test length limit. The approach is parameterized by `MEMORY`, the number of "good" tests to store, by `PEXTEND`, the probability of choosing to extend a "good" test rather than start a new test, by the `TEST_LENGTH` and by a `TIMEOUT` parameter. Leaving out imports and other boilerplate, the entire implementation is shown in Figure 10.

The implementation is trivial, relying only on the TSTL API and some very simple Python tools (sorting with automatic lexical ordering, time li-

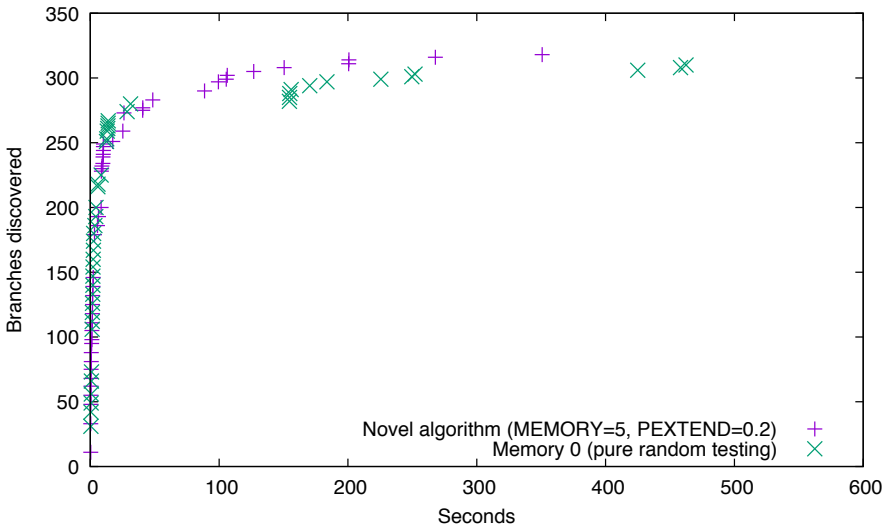

Fig. 11 Comparing branch coverage for 10 minute runs of two test generation methods.

brary, etc.). We omit handling of failed tests, assuming the goal of this algorithm is simply to improve code coverage in fault-free systems for experimental evaluation. This simple tool can be applied to any TSTL harness and will produce output showing when, in time, new branches were covered by the system. This data can be used to produce Average-Percent-Branches-Detected (APBD) values and discovery curves [97,85,84]. Comparison with simple random testing is easy, since setting `MEMORY` to 0 gives pure memoryless random testing (alternatively, to avoid the overhead of the comparisons with 0, a dedicated version for random testing can be written). A major threat to validity in many comparisons of testing or explicit-state model checking algorithms is that different underlying infrastructure for different algorithms may end up outweighing even moderately sized effects due to the underlying algorithms. With TSTL, fair comparisons are much easier, since the TSTL interface does most of the computational work that is common to multiple algorithms, with the same overhead. Evaluating an algorithm can be as simple as finding a large number of suitable programs without failures (or where failures don't make coverage values invalid) and performing enough trials to establish statistical validity for comparisons with APBD values for known algorithms. Evaluation in terms of discovered faults or time-until-discovery of a fault is nearly as simple. This algorithm is of some interest, in that while it requires backtracking, the frequency of backtracking is low enough to be potentially applicable even to systems like ArcPy where backtracking is only possible via expensive test replay. While a mature version of this method would require many SUTs and experiments, as well as investigation of suitable values for `MEMORY` and `PEXTEND`, Figure 11 shows that average branch discovery curves for ArcPy can sometimes be improved, even using the arbitrarily chosen parameters of a size

5 memory and a 20% probability of using a "good" test as a starting point. The simplicity of the Python implementation makes performing automatic experiments with different parameters and test lengths trivial. Experiments can also take advantage of Python libraries for automatic statistical analysis and plotting of results.

6 TSTL as a Testing Library Generator

While TSTL is easily used as "just another tool" that allows testing of an SUT, plus a "construction kit" to build your own testing tools, TSTL can also be understood as a generator of libraries. ArcPy is a site package that makes it easy to perform GIS tasks using Python. NumPy [4] and SciPy [6] are libraries that make performing scientific computing tasks easy with Python. QIIME [19], Biopython [2], and scikit-bio [5] are libraries that make performing bioinformatics tasks easy using Python. Such libraries can be of great importance in their subject domains: ArcPy is extremely widely used, and the subject of university courses in many GIS programs, and the Nature Methods paper introducing QIIME has been cited more than 5,000 times to date, according to Google Scholar. TSTL, also, can be seen as a tool that generates a library making it easy to perform testing tasks *for a specific SUT*, using Python.

ArcPy, NumPy, QIIME and the other libraries can be seen as introducing the entities essential to their respective tasks as first-class, easily manipulable objects in the Python language. TSTL makes tests (for a specific SUT, but using a common interface across all SUTs) first-class, easily manipulable objects. TSTL does for tests, SUT state, and code coverage what ArcPy does for feature classes, spatial references, and other GIS concepts, what NumPy does for efficiently represented arrays and matrices, and what QIIME does for protein sequences.

The key to understanding TSTL as a library generator is the idea of a test. A TSTL test is a list of actions[6]. The individuals of the action type are dependent on the SUT. TSTL allows the user to generate, manipulate, and inspect tests in the same way NumPy lets a user explore the behavior of arrays. This means that TSTL can also be used interactively, as the following example (using some TSTL-supplied methods not shown in Table 9) shows:

```
>>> import sut, random
>>> sut = sut.sut()
>>> r = random.Random()
>>> (t1, ok) = sut.makeTest(100,r)
>>> print ok
True
>>> sut.prettyPrintTest(t1)
fieldname1 = "newf1"                                                    # STEP 0
polytable0 = arcpy.env.workspace + "\polyneig.dbf"                      # STEP 1
...
arcpy.Erase_analysis(classorlayer0,classorlayer1,featureclass0); report() # STEP 98
arcpy.Buffer_analysis(classorlayer3,featureclass0,dist1); report()       # STEP 99
```

[6] We expect this type to be the same for any TSTL version for any language: a list is the simplest way to express pure sequence, which is the essence of a test.

Here, a user generates a length 100 test for ArcPy, and then prints it. The user can also modify the test slightly, using standard Python list modification, generate another test, produce a third test that is the *composition* of the first two tests, and reduce that third test to remove any redundant (with respect to code coverage) steps in it.

```
>>> t1[1] = sut.playable(t1[1][0].replace("newf1","newf2"))
>>> (t2,ok) = sut.makeTest(100,r)
>>> print ok
True
>>> t3 = t1 + t2
>>> sut.replay(t3)
>>> bc = sut.currBranches()
>>> t4 = sut.reduce(t3, sut.coversBranches(bc))
>>> len(t4)
78
...
```

The user is causing ArcGIS to perform GIS operations, but without specifying those operations; the GIS tasks are conceptually reduced to the actions of an arbitrary SUT, but the printed test makes it clear what is going on at the SUT level. While this example shows the `makeTest` interface being used with the default generator (pure random testing), a user can supply a more complex generator as a function. For example, to implement testing such that if the last action resulted in any new coverage, it is repeated, a user could first define a generator function (assume that `r` is a random number generator defined globally, as above):

```
def repeatGen(lastAction,sut):
   if (lastAction != None) and (len(sut.newBranches()) > 0) and (lastAction[1]()):
      return (lastAction, lastAction)
   newAction = sut.randomEnabled(r)
   return (newAction, newAction)
```

and then generate a length 100 test using this strategy easily:

```
>>> (t5, ok) = sut.makeTest(100, sgenerator=repeatGen)
```

The function `repeatGen` takes a state (initially `None` to indicate a new test) consisting of a single action — the last action executed. It then, if there is a`lastAction`, and the last step of testing increased branch coverage, and the action is still enabled, returns that action as both the next action to perform and the new state of the generator. Otherwise, it generates a random action, and returns that as the state and the action.

We do not believe such an interactive approach to testing is natural with most other tools that generate either model checking traces or method-call sequence tests. In fact, the very idea of test case composition as shown in the first interactive code fragment may seem quite strange to users of either model checking or conventional unit tests. for generating API-call sequen. Although TSTL tests are sequences of actions — essentially small, deterministic programs with no inputs — they can be interacted with and generated like tests in QuickCheck [23] or Hypothesis [73], where tests are usually just function inputs (lists, data structures, etc.). This ease-of-use for exploratory testing is a primary reason TSTL was first implemented for Python, rather than a

```
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. 12 Deleting a feature class does not invalidate or delete layers that depend on it.

compiled language such as Java or C (hence the "Scripting" part of "Template Scripting Testing Language"). A Swift, Ruby, or Scala version would also provide a way to interact with tests in a simple, immediate, and basically "functional" way.

7 Faults Discovered Using TSTL

7.1 ArcPy Faults

In the process of testing ArcPy with TSTL, we discovered at least five distinct faults (thus far) that can cause an ArcPy script to crash. While we have (as discussed in Section 4) some properties that check for data corruption and determinism of GIS analysis, we are not focusing on these until we have a reliable way to avoid system crashes. In order to give an idea of what TSTL test cases look like, we discuss briefly one of these ArcPy crashes.

ArcPy crashes when the feature class from which a layer is produced is deleted, and the layer is used in a `SelectLayer` call (this version shows an attribute-based selection, but location selection will cause the same problem):

(Figure 12). The underlying issue seems to be that while operations on a deleted feature class properly notify a user the feature class does not exist, ArcPy or ArcGIS does not track that layers produced from a feature class should also be deleted/invalidated when the feature class is deleted. Layers are not copies of a feature class, but essentially new *views* of a feature class. This means that when the underlying feature class is modified or deleted, the view needs to be updated to reflect that change, and this is not correctly implemented. Figure 12 shows part of an annotated, reduced, normalized, and generalized test stand-alone test case (with the boilerplate, function definitions, and imports removed) for this fault. The final line of code crashes ArcPy and the Python interpreter. Comments indicate alternative similar tests that also fail. In this case, TSTL's additional reduction steps (based on term rewriting in the action language) remove almost half the steps in the original, delta-debugged test case.

Other faults (or documentation lapses) in ArcPy we have discovered include crashes when computing statisics over database fields of a layer using a deleted field and crashes due to seemingly reasonable modifications of feature classes while a database cursor is active. In order to deal with the latter, which seems more in the line of an undocumented behavioral restriction than a "bug," we now drastically limit database modification when a cursor is active. We have reported these problems to Esri, but have not received a response. The problems discovered may be previously known to Esri, but are not generally known to the ArcPy user community, and those that could be considered API limitations (that cause unexplained crashes when violated) are not documented.

7.2 Faults in Other Systems

TSTL is only slightly over a year old. However, students using TSTL in graduate classes on software testing have already, with minimal assistance, discovered faults in some real-world systems. Not all of these are confirmed and reported yet.

First, TSTL testing revealed a fault in either the widely-used `PyOpenCL` library [67], the even more widely-used `OpenCL` infrastructure [66], or (possibly) the NVIDIA hardware being used. We are still investigating this problem, but it appears to be a genuine fault, though debugging and assigning blame is complex due to the layers of software and hardware involved. Second, TSTL testing found cases where distance metrics that were supposed to be symmetric in the popular fuzzy-string-matching library FuzzyWuzzy [41] were asymmetric, if the default Python string match library was used instead of a Levenshtein-distance library. Third, TSTL testing revealed numerous problems with the `astropy.table` module of the AstroPy library [1], used by many professional astronomers and astrophysicists. TSTL has also been used to discover faults in the TSTL API itself.

The github repository for the graduate class in question (`https://github.com/agroce/cs562w16`) contains the TSTL code for testing these systems (and many other student projects).

Most significantly, TSTL was able to discover at least 15 previously undiscovered faults in the widely used SymPy library for symbolic mathematics in Python [91]. We have reported these faults, and hope to collaborate with the SymPy team to provide assistance in localizing and fixing them using TSTL. The SymPy effort was able to move from decision-to-test to first discovered fault in the course of a single day, due to the much higher ease-of-use for the post-ArcPy version of TSTL used. For a straightforward testing task like SymPy, building a TSTL harness is now quite simple, largely a matter of thinking about what the user wishes to test.

8 Related Work

There is a vast amount of previous work on automated generation of tests for (API-based) software systems [80,35,40] and random testing in particular [50, 80,12,21,13,89,60,59,23,22,14,52,11,10,29], some dating back to the early 1980s. It is far beyond the scope of this paper to explore that literature in detail. The interested reader is directed to the cited papers, as well as general surveys of automated test generation in particular [8] or recent software testing research in general [79].

TSTL is a domain-specific-language (DSL) [34] for testing, in the spirit of the famous QuickCheck tool for Haskell [23] that inspired much current interest in property-based testing. To our knowledge, there has been no previous proposal of a concise DSL like TSTL, to assist users in building test harnesses for API-call sequences, and to make test generation and manipulation first-class activities in a language. QuickCheck itself produces tests that are simply values input to functions, not sequences of actions. Some QuickCheck-like tools, such as ScalaCheck [78] or the excellent Python tool Hypothesis [73] include limited, and rather cumbersome to use, state-machine-based tools for generating call-sequence tests. These tools are also generally limited to the included methods for generating such tests, and make building novel sequence generation algorithms based on state-exploration and backtracking (model checking methods) or feedback from coverage difficult at best.

There is limited previous work on building common frameworks for random testing and model checking [52], or proposing common terminology for imperative harnesses [45]. Earlier publications on TSTL itself [54,55] presented a language considerably more limited in functionality and with a more difficult-to-read syntax. These publications omitted details of the tools provided in the TSTL distribution for off-the-shelf testing [56], and provided little practical guidance to potential users of TSTL. More critically, these papers did not descirbe the long-term core concept of TSTL as primarily making testing and explicit-state verification a first-class, library-supported activity for any programming language. Some technical details of the current TSTL imple-

mentation were presented in the NASA Formal Methods paper [54] that we omit in the interest of space (and because implementation details are subject to change).

There exist various testing tools and languages of a somewhat different flavor than TSTL that address the problem of helping users generate tests. Korat [77], for example, has a much more fixed input domain specification, as do the tools built to support the Next Generation Air Transportation System (NextGen) software [37]. The model-based TestStories approach of Felderer et al. [32], the software-as-a-service oriented approach of Santiago et al. [87], and the use-case-based DSL for extracting tests proposed by Im et al. [64] similarly aim at a fairly restricted (and simply different) goal of turning certain inputs (in a DSL tied to a model or documentation) into tests or test skeletons. Utting et al. survey the wide variety of specifically model-based testing approaches, including many tools [92]; while TSTL is not specifically model-based, it can be used to facilitate model-based testing in theory. Behavior-driven-development (BDD) [20] is often supported by extracting formally meaningful parts of a specification document to (partially) instantiate a test.

Perhaps the most similar system to TSTL is the UDITA language [39], an extension of Java with non-deterministic choice operators and `assume`, which yields a very different language but shares our goal of making it easy to define the set of valid tests for a system. 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` statements) means that it must be hosted inside a complex (and sometimes non-trivial to install/use) tool, JPF [93], rather than generating a stand-alone simple interface to a test space, as with TSTL. Building "UDITA" for a new language is far more challenging than porting TSTL. UDITA supports many fewer constructs to assist harness development. It is impossible to "interact" with a UDITA test harness in a simple way, as with TSTL; UDITA *is* a tool, while TSTL has tools that are built on top of a common library interface.

The design of the SPIN model checker [63] and its model-driven extension to include native C code [61] inspired the flavor of TSTL's domain-specific language, though our approach is more declarative than the "imperative" model checker produced by SPIN, and our system less tied to a particular method of exploration. Work at JPL on languages for analyzing spacecraft telemetry logs in testing [49] provided a working example of a Python-based declarative language useful in testing. The pool approach to test case construction is derived from work on canonical forms and enumeration of unit tests [9], and common to some other test generators [80].

A primary difference between all such systems and TSTL is that TSTL is meant to be much more general, without focusing on a particular system model or specification method. TSTL, unlike any other system of which we are aware, aims to make test cases first-class objects, and provides tools for development of code that manipulates, executes, and analyzes test cases. In a sense, TSTL is less a tool for creating tests than a method for generating a language extension

(custom to each SUT, but with a common interface) that makes test activities as well supported as, e.g., ArcPy makes GIS operations, or NumPy makes numeric operations. Because testing is intimately tied to a particular SUT, this process is more involved than in typical libraries, requiring the step of compiling a TSTL model, but the end-goal is similar. No such language-based interface is possible in more traditional testing or model checking tools, to our knowledge. Extending even a highly extensible system such as Java PathFinder [93] is much more like extending an existing complex software system than simply using a library API.

A second aspect of TSTL is that while it is intended to be useful to researchers in software testing and model checking, and help prototype new algorithms, it is primarily designed to be a practical tool and language extension for users. Some other recent work on automated test generation has given more attention to practical, rather than primarily algorithmic or "pure" research, issues than in the past. We suspect this shows a growing technological maturity for automated test case generation. E.g., the papers by the NASA/JPL group on testing the Curiosity rover's file system [50, 51, 48] have a largely practical focus, and the work of Lei and Andrews [72] emphasizes the need for delta-debugging in realistic random testing. Pike's SmartCheck [82] is not intended to increase fault detection so much as to improve the usability of test cases produced by QuickCheck. We found that test case readability was important in our efforts, and recent academic work [27, 26] has introduced principled methods for improving the readability of test cases for humans, even though this does not improve fault detection, coverage, or other traditional measures of test effectiveness.

The literature on testing GIS (Geographic Information Systems) software in particular seems to consist of one paper proposing a very limited application of automated testing to assist GIS users, primarily in model development [74]. That work does not target the reliability or correctness of the underlying GIS engine, or GIS libraries. There is also some discussion of automated testing for GIS in various blog posts and discussion groups (e.g., [95, 7]), but no formal academic case studies. These discussions also tend to focus on application testing or GUI testing, rather than testing of library code used across GIS applications. There is a simple extension to Python unit testing modules for the GRASS open source GIS system [42], but this does not provide any automated test generation.

There is a significant body of work on end-user testing of software, part of the larger field of end-user software engineering [16, 17]. End-user software engineering examines how software can best be produced by developers who do not have a traditional computer science background, and are often primarily interested in an application of programming, rather than software development as a profession. GIS developers are (we believe) a typical example [88]: they are technically skilled individuals whose primary expertise is not in software development, but who, in order to pursue their goals, must develop, maintain, and test significant software systems.

The earliest work focusing on software testing for end-user software engineers explored how to test spreadsheets [83,86]. Other work has focused on errors end-users make in specifying systems [81], and how end-users of machine learning systems (who may be machine learning experts, or individuals with no programming knowledge at all) can test such systems [53,70,90]. To our knowledge, no previous work considers function call sequence testing for end-users. Previous work on such testing has often been performed only by software testing researchers, not even including traditional developers.

9 Conclusions

This paper presents the latest version of the TSTL [54–56] domain-specific language for testing, which enables a declarative style of test harness development, where the focus is on defining the actions in valid tests, not determining exactly how tests are generated. Because TSTL, inspired by the SPIN model checker, produces a software-under-test-independent interface for testing, TSTL makes it possible for users to easily apply different test generation methods to the same system without undue effort. The same approach makes it possible for researchers to rapidly prototype novel test generation methods, and evaluate them in a context where differences in test infrastructure not relevant to the algorithms at hand can be minimized.

TSTL has, in the year since its initial introduction, already been used to discover previously unknown (to our knowledge) faults in multiple Python libraries, including the very widely-used ArcPy site package for GIS scripting. As future work, we plan to continue to use TSTL to explore novel testing algorithms, investigate the relative strengths of systematic, stochastic, and directed test generation methods, and apply TSTL to look for faults in widely used libraries. Finally, we plan to port TSTL to additional programming languages beyond Python and Java, and add automatic support for new properties, including information-flow based security checks.

More generally, TSTL takes the approach to GIS embodied in ArcPy, or to biology embodied in QIIME [19], and applies it to testing: TSTL makes it possible for users to create, manipulate, and execute test cases in the context of an easily learned programming language. Just as ArcPy makes the automation of GIS tasks easier, TSTL aims to make the automation of all testing tasks easier, by supporting the functionality common to most testing tasks. The choice of System Under Test in TSTL is analogous to the choice of using GIS to analyze epidemiological data or urban traffic routes using ArcPy; ArcPy provides features that are common across all such applications. TSTL provides the same functionality with respect to software testing.

References

1. AstroPy: a community Python library for astronomy. `http://www.astropy.org/`
2. Biopython. `http://biopython.org/wiki/Biopython`

3. JPF: the Swiss army knife of Java(TM) verification. http://babelfish.arc.nasa.gov/trac/jpf
4. NumPy. https://www.numpy.org
5. scikit-bio. http://scikit-bio.org/
6. SciPy. https://www.scipy.org
7. AbSharma: Functional testing of GIS applications (automated testing). http://osgeo-org.1560.x6.nabble.com/Functional-Testing-of-GIS-applications-Automated-Testing-td4493673.html
8. Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P.: An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software **86**(8), 1978–2001 (2013)
9. Andrews, J., Zhang, Y.R., Groce, A.: Comparing automated unit testing strategies. Tech. Rep. 736, Department of Computer Science, University of Western Ontario (2010)
10. Andrews, J.H., Groce, A., Weston, M., Xu, R.G.: Random test run length and effectiveness. In: Automated Software Engineering, pp. 19–28 (2008)
11. Andrews, J.H., Haldar, S., Lei, Y., Li, C.H.F.: Tool support for randomized unit testing. In: Proceedings of the First International Workshop on Randomized Testing, pp. 36–45. Portland, Maine (2006)
12. Andrews, J.H., Menzies, T., Li, F.C.: Genetic algorithms for randomized unit testing. IEEE Transactions on Software Engineering (TSE) **37**(1), 80–94 (2011)
13. Arcuri, A., Briand, L.: Adaptive random testing: An illusion of effectiveness. In: International Symposium on Software Testing and Analysis, pp. 265–275 (2011)
14. Arcuri, A., Iqbal, M.Z.Z., Briand, L.C.: Formal analysis of the effectiveness and predictability of random testing. In: International Symposium on Software Testing and Analysis, pp. 219–230 (2010)
15. Bentley, J.: Programming pearls: little languages. Communications of the ACM **29**(8), 711–721 (1986)
16. Burnett, M., Cook, C., Rothermel, G.: End-user software engineering. Comm. ACM **47**(9), 53–58 (2004)
17. Burnett, M.M., Myers, B.A.: Future of end-user software engineering: beyond the silos. In: Future of Software Engineering, pp. 201–211 (2014)
18. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Operating System Design and Implementation, pp. 209–224 (2008)
19. Caporaso, J.G., Kuczynski, J., Stombaugh, J., Bittinger, K., Bushman, F.D., Costello, E.K., Fierer, N., Pena, A.G., Goodrich, J.K., Gordon, J.I., et al.: QIIME allows analysis of high-throughput community sequencing data. Nature methods **7**(5), 335–336 (2010)
20. Chelimsky, D., Astels, D., Helmkamp, B., North, D., Dennis, Z., Hellesoy, A.: The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends, 1st edn. Pragmatic Bookshelf (2010)
21. Chen, T.Y., Leung, H., Mak, I.K.: Adaptive random testing. In: Advances in Computer Science, pp. 320–329 (2004)
22. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Experimental assessment of random testing for object-oriented software. In: D.S. Rosenblum, S.G. Elbaum (eds.) International Symposium on Software Testing and Analysis, pp. 84–94. ACM (2007)
23. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of haskell programs. In: ICFP, pp. 268–279 (2000)
24. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (2000)
25. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory efficient algorithms for the verification of temporal properties. In: Proceedings of the 2nd International Workshop on Computer Aided Verification, CAV '90, pp. 233–242. Springer-Verlag, London, UK, UK (1991). URL http://dl.acm.org/citation.cfm?id=647759.735018
26. Daka, E., Campos, J., Dorn, J., Fraser, G., Weimer, W.: Generating readable unit tests for Guava. In: Search-Based Software Engineering - 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings, pp. 235–241 (2015)
27. Daka, E., Campos, J., Fraser, G., Dorn, J., Weimer, W.: Modeling readability to improve unit tests. In: Foundations of Software Engineering, ESEC/FSE, pp. 107–118 (2015)
28. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs, New Jersey (1976)

29. Duran, J.W., Ntafos, S.C.: Evaluation of random testing. IEEE Transactions on Software Engineering **10**(4), 438–444 (1984)
30. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. Int. J. Softw. Tools Technol. Transf. **5**(2), 247–267 (2004). DOI 10.1007/s10009-002-0104-3. URL http://dx.doi.org/10.1007/s10009-002-0104-3
31. Esri: What is ArcPy? http://resources.arcgis.com/EN/HELP/MAIN/10.1/index.html000v000000v7000000
32. Felderer, M., Zech, P., Fiedler, F., Breu, R.: A tool-based methodology for system testing of service-oriented systems. In: Advances in System Testing and Validation Lifecycle (VALID), 2010 Second International Conference on, pp. 108–113 (2010). DOI 10.1109/VALID.2010.12
33. Floyd, R.W.: Nondeterministic algorithms. J. ACM **14**(4), 636–644 (1967). DOI 10.1145/321420.321422. URL http://doi.acm.org/10.1145/321420.321422
34. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010)
35. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: ACM SIGSOFT Symposium/European Conference on Foundations of Software Engineering, pp. 416–419 (2011)
36. Gamma, E., Beck, K.: JUnit. http://junit.sourceforce.net
37. Giannakopoulou, D., Howar, F., Isberner, M., Lauderdale, T., Rakamarić, Z., Raman, V.: Taming test inputs for separation assurance. In: International Conference on Automated Software Engineering, pp. 373–384 (2014)
38. Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, A., Marinov, D.: Comparing non-adequate test suites using coverage criteria. In: International Symposium on Software Testing and Analysis, pp. 302–313 (2013)
39. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in UDITA. In: International Conference on Software Engineering, pp. 225–234 (2010)
40. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Programming Language Design and Implementation, pp. 213–223 (2005)
41. Gonzalez, J.: FuzzyWuzzy. https://pypi.python.org/pypi/fuzzywuzzy
42. GRASS Development Team: Testing GRASS GIS source code and modules. https://grass.osgeo.org/grass71/manuals/libpython/gunittest_testing.html
43. Groce, A., Alipour, M.A., Zhang, C., Chen, Y., Regehr, J.: Cause reduction for quick testing. In: Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on, pp. 243–252. IEEE (2014)
44. Groce, A., Alipour, M.A., Zhang, C., Chen, Y., Regehr, J.: Cause reduction: Delta-debugging, even without bugs. Journal of Software Testing, Verification, and Reliability **26**(1), 40–68 (2016)
45. Groce, A., Erwig, M.: Finding common ground: choose, assert, and assume. In: Workshop on Dynamic Analysis, pp. 12–17 (2012)
46. Groce, A., Fern, A., Erwig, M., Pinto, J., Bauer, T., Alipour, A.: Learning-based test programming for programmers. In: International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, pp. 752–786 (2012)
47. Groce, A., Fern, A., Pinto, J., Bauer, T., Alipour, A., Erwig, M., Lopez, C.: Lightweight automated testing with adaptation-based programming. In: IEEE International Symposium on Software Reliability Engineering, pp. 161–170 (2012)
48. Groce, A., Havelund, K., Holzmann, G., Joshi, R., Xu, R.G.: Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning. Annals of Mathematics and Artificial Intelligence **70**(4), 315–349 (2014)
49. Groce, A., Havelund, K., Smith, M.: From scripts to specifications: The evolution of a flight software testing effort. In: International Conference on Software Engineering, pp. 129–138 (2010)
50. Groce, A., Holzmann, G., Joshi, R.: Randomized differential testing as a prelude to formal verification. In: International Conference on Software Engineering, pp. 621–631 (2007)
51. Groce, A., Holzmann, G., Joshi, R., Xu, R.G.: Putting flight software through the paces with testing, model checking, and constraint-solving. In: Workshop on Constraints in Formal Verification, pp. 1–15 (2008)

52. Groce, A., Joshi, R.: Random testing and model checking: Building a common framework for nondeterministic exploration. In: Workshop on Dynamic Analysis, pp. 22–28 (2008)
53. Groce, A., Kulesza, T., Zhang, C., Shamasunder, S., Burnett, M.M., Wong, W., Stumpf, S., Das, S., Shinsel, A., Bice, F., McIntosh, K.: You are the only possible oracle: Effective test selection for end users of interactive machine learning systems. IEEE Trans. Software Eng. **40**(3), 307–323 (2014)
54. Groce, A., Pinto, J.: A little language for testing. In: NASA Formal Methods Symposium, pp. 204–218 (2015)
55. Groce, A., Pinto, J., Azimi, P., Mittal, P.: TSTL: a language and tool for testing (demo). In: ACM International Symposium on Software Testing and Analysis, pp. 414–417 (2015)
56. Groce, A., Pinto, J., Azimi, P., Mittal, P., Holmes, J., Kellar, K.: TSTL: the template scripting testing language. `https://github.com/agroce/tstl`
57. Groce, A., Visser, W.: Model checking Java programs using structural heuristics. In: International Symposium on Software Testing and Analysis, pp. 12–21 (2002)
58. Groce, A., Zhang, C., Eide, E., Chen, Y., Regehr, J.: Swarm testing. In: International Symposium on Software Testing and Analysis, pp. 78–88 (2012)
59. Hamlet, R.: Random testing. In: Encyclopedia of Software Engineering, pp. 970–978. Wiley (1994)
60. Hamlet, R.: When only random testing will do. In: International Workshop on Random Testing, pp. 1–9 (2006)
61. Holzmann, G., Joshi, R.: Model-driven software verification. In: SPIN Workshop on Model Checking of Software, pp. 76–91 (2004)
62. Holzmann, G., Joshi, R., Groce, A.: Model driven code checking. Automated Software Engineering **15**(3–4), 283–297 (2008)
63. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
64. Im, K., Im, T., McGregor, J.D.: Automating test case definition using a domain specific language. In: Proceedings of the 46th Annual Southeast Regional Conference on XX, ACM-SE 46, pp. 180–185. ACM, New York, NY, USA (2008). DOI 10.1145/1593105.1593152. URL `http://doi.acm.org/10.1145/1593105.1593152`
65. Kellar, K.: Tstl-java. `https://github.com/flipturnapps/TSTL-Java`
66. Khronos Group: The open standard for parallel programming of heterogenous systems. `https://www.khronos.org/opencl/`
67. Klockner, A.: PyOpenCL. `https://mathema.tician.de/software/pyopencl/`
68. Kroening, D.: The CBMC homepage. `http://www.cs.cmu.edu/~modelcheck/cbmc/`
69. Kroening, D., Clarke, E.M., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 168–176 (2004)
70. Kulesza, T., Burnett, M., Stumpf, S., Wong, W.K., Das, S., Groce, A., Shinsel, A., Bice, F., McIntosh, K.: Where are my intelligent assistant's mistakes? a systematic testing approach. In: Intl. Symp. End-User Development, pp. 171–186 (2011)
71. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 216–226 (2014)
72. Lei, Y., Andrews, J.H.: Minimization of randomized unit test cases. In: International Symposium on Software Reliability Engineering, pp. 267–276 (2005)
73. MacIver, D.R.: Hypothesis: Test faster, fix more. `http://hypothesis.works/`
74. Maogui, H., Jinfeng, W.: Application of automated testing tool in GIS modeling. In: World Congress on Software Engineering, pp. 184–188 (2009)
75. McCarthy, J.: A basis for a mathematical theory of computation, preliminary report. In: Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference, IRE-AIEE-ACM '61 (Western), pp. 225–238. ACM, New York, NY, USA (1961). DOI 10.1145/1460690.1460715. URL `http://doi.acm.org/10.1145/1460690.1460715`
76. McKeeman, W.: Differential testing for software. Digital Technical Journal of Digital Equipment Corporation **10(1)**, 100–107 (1998)

77. Milicevic, A., Misailovic, S., Marinov, D., Khurshid, S.: Korat: A tool for generating structurally complex test inputs. In: International Conference on Software Engineering, pp. 771–774 (2007)
78. Nilsson, R.: ScalaCheck: property-based testing for Scala. https://www.scalacheck.org
79. Orso, A., Rothermel, G.: Software testing: A research travelogue (2000–2014). In: Proceedings of the on Future of Software Engineering, FOSE 2014, pp. 117–132 (2014)
80. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: International Conference on Software Engineering, pp. 75–84 (2007)
81. Phalgune, A., Kissinger, C., Burnett, M., Cook, C., Beckwith, L., Ruthruff, J.: Garbage in, garbage out? an empirical look at oracle mistakes by end-user programmers. In: IEEE Symp. Visual Languages and Human-Centric Computing, pp. 45–52 (2005)
82. Pike, L.: SmartCheck: automatic and efficient counterexample reduction and generalization. In: ACM SIGPLAN Symposium on Haskell, pp. 53–64 (2014)
83. Rothermel, G., Burnett, M., Li, L., DuPois, C., Sheretov, A.: A methodology for testing spreadsheets. ACM Trans. Software Eng. and Methodology **10**(1), 110–147 (2001)
84. Rothermel, G., Untch, R., Chu, C., Harrold, M.J.: Test case prioritization. Trans. Softw. Eng. **27**, 929–948 (2001)
85. Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Test case prioritization: An empirical study. In: Proceedings of the IEEE International Conference on Software Maintenance, ICSM '99, pp. 179–188. IEEE Computer Society, Washington, DC, USA (1999). URL http://dl.acm.org/citation.cfm?id=519621.853398
86. Rothermel, K., Cook, C., Burnett, M., Schonfeld, J., Green, T., Rothermel, G.: WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation. In: Intl. Conf. Software Eng., vol. 22, pp. 230–240 (2000)
87. Santiago, D., Cando, A., Mack, C., Nunez, G., Thomas, T., King, T.M.: Towards domain-specific testing languages for software-as-a-service. In: Proceedings of the 2nd International Workshop on Model-Driven Engineering for High Performance and Cloud computing co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MODELS, pp. 43–52 (2013)
88. Segal, J.: Some problems of professional end user developers. In: IEEE Symp. Visual Languages and Human-Centric Computing (2007)
89. Sharma, R., Gligoric, M., Arcuri, A., Fraser, G., Marinov, D.: Testing container classes: Random or systematic? In: Fundamental Approaches to Software Engineering, pp. 262–277 (2011)
90. Shinsel, A., Kulesza, T., Burnett, M.M., Curan, W., Groce, A., Stumpf, S., Wong, W.K.: Mini-crowdsourcing end-user assessment of intelligent assistants: A cost-benefit study. In: Visual Languages and Human-Centric Computing, pp. 47–54 (2011)
91. SymPy Development Team: SymPy. http://www.sympy.org/en/index.html
92. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verif. Reliab. **22**(5), 297–312 (2012). DOI 10.1002/stvr.456. URL http://dx.doi.org/10.1002/stvr.456
93. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10**(2), 203–232 (2003)
94. Visser, W., Păsăreanu, C., Pelanek, R.: Test input generation for Java containers using state matching. In: International Symposium on Software Testing and Analysis, pp. 37–48 (2006)
95. XBOSOFT: GIS software testing - lessons learned. http://xbosoft.com/gis-software-testing-lessons-learned/
96. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. Software Engineering, IEEE Transactions on **28**(2), 183–200 (2002)
97. Zhang, C., Groce, A., Alipour, M.A.: Using test case reduction and prioritization to improve symbolic execution. In: International Symposium on Software Testing and Analysis, pp. 160–170 (2014)