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, and usually also defines a set of 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. Such languages 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 a graph-based interface for testing, making generic test generation and manipulation tools that apply to any SUT possible. TSTL includes a compiler to Python as well as a suite of tools for generating, manipulating, and analyzing test cases. These include simple but easily extensible explicit-state model checkers. This paper showcases the features of TSTL by focusing on one large-scale testing effort, directed by an

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

end-user, to find faults in the most widely used Geographic Information Systems (GIS) tool, Esri's ArcGIS, using its ArcPy scripting interface. We also demonstrate TSTL's power for prototyping novel test generation algorithms.

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 [37,30]. 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 [24] and its imitators (e.g., PyUnit, HUnit, etc.). It is also how many industrial-strength random testing systems are written [35,33]. A KLEE "test harness" [14] 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 [2, 62] can easily be seen as offering a way to define a state space using Java itself as the modeling language, and CBMC [1,48] 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 [63].

Unfortunately, writing test harnesses this way is a highly repetitive and error-prone programming task, with many conceptual "code clones" (e.g. Figure 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 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 [33] 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 [32]. 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 [23, 55] 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 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

```
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.

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 Domain Specific Languages for Testing

The nature of test harness construction suggests the use of a *domain-specific language* (DSL) for testing [31]. DSLs [22] 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 [11] 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 [26,32,62,48,14]. 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 [47]. Version 4.0 of SPIN [45] 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 [33,46]. 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 [34]. 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).

```
@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.

1.2 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*.

The *template* aspect is based on 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 [41] 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

[1] We also have developed a beta version of TSTL for Java, 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.

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 [65] 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 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, in future implementations) 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.

1.3 Contributions

We showcase the TSTL approach and tools using an ongoing case study, applying 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 improved.

This paper presents the most complete presentation of the TSTL language and tools, and we hope that it satisfies two 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 [41]). This paper thus serves as a basic "TSTL cookbook."

– 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.

2 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 [21]. 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 [23, 55].

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 [12, 13, 57] is long-standing. Previous work in the field has often focused on non-traditional programming: e.g. spreadsheets [57], visual languages, or machine-learning systems [38]. 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 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 [32]. Even "push-button" tools for automated testing are sometimes difficult for expert users to install, apply, and configure [33,36,32]. 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.

3 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 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 [39] 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.

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

3.1 The Essentials of Pools and Actions

In TSTL, tests basically consist of assignments to value pools and function calls making use of those values. 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 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.

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)[3]. 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.

[3] 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.

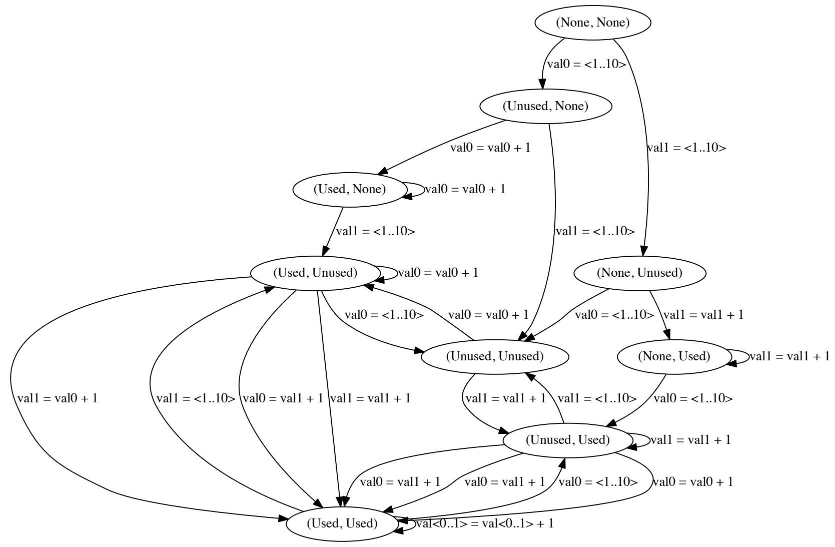

Fig. 5 Constraints on actions in a test, based on pool states

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).

3.2 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"]>`. 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 than it should copy the comma indexed appearance 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 [52] 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 format for feature classes.

3.3 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.

4 TSTL Tools

The following tools are provided in the TSTL distribution on github [41]. 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.

4.1 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.

4.2 Test Generators

TSTL comes with a complex, highly configurable, pure random tester (supporting numerous command-line options), and simple depth-first-search and breadth-first-search based backtracking model checkers. The included random tester provides a number of useful options, of which a subset are shown in Figure 6. One of these, the `stutter` option, is a (to our knowledge) novel idea in random testing, where any action enabled in a step has a fixed probability of being repeated. TSTL makes implementing novel test generation methods simple, as discussed below.

TSTL can (unlike SPIN or most explicit-state model checkers) apply BFS or DFS search even to systems without support for backtracking (e.g., where Python's `deepcopy` fails because some object state is handled by a Python C

- `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.
- `stutter <float P>`: If this option is provided, the random tester will repeat any still-enabled actions with probability P.
- `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 [29], 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.

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

extension), using replay-based state storage and retrieval. The model checkers take some of the same command-line options as the random tester, but are considerably less full-featured, since exhaustive generation is applicable to fewer Python libraries (for instance, it is possible but impractical for the ArcPy implementation, due to the overhead of backtracking via replay). 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. without support for backtracking (e.g., where Python's `deepcopy` fails because some object state is handled by a Python C extension), using replay-based state storage and retrieval.

4.3 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.

4.4 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

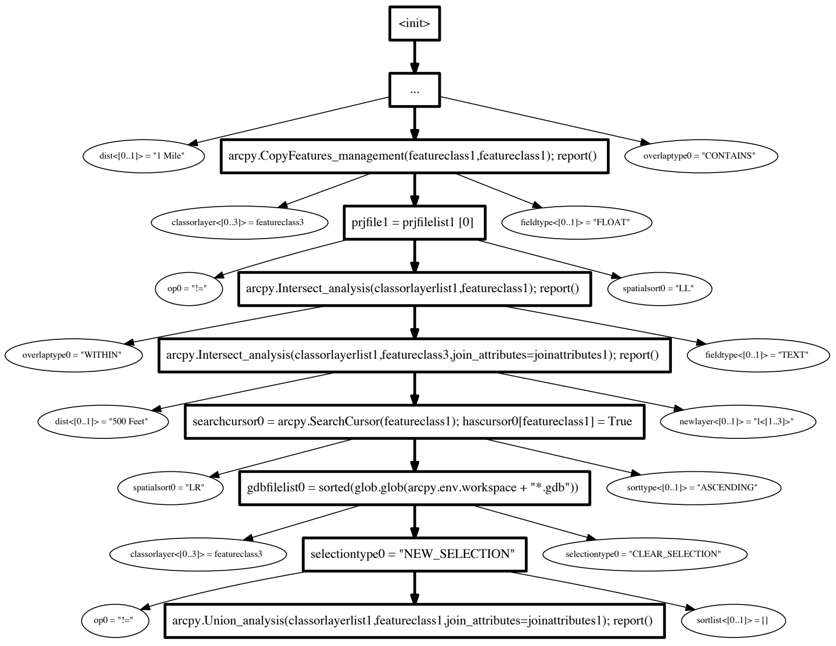
<init>
...
dist<[0..1]> = "1 Mile"
arcpy.CopyFeatures_management(featureclass1,featureclass1); report()
overlaptype0 = "CONTAINS"
classorlayer<[0..3]> = featureclass3
prjfile1 = prjfilelist1 [0]
fieldtype<[0..1]> = "FLOAT"
op0 = "!="
arcpy.Intersect_analysis(classorlayerlist1,featureclass1); report()
spatialsort0 = "LL"
overlaptype0 = "WITHIN"
arcpy.Intersect_analysis(classorlayerlist1,featureclass3,join_attributes=joinattributes1); report()
fieldtype<[0..1]> = "TEXT"
dist<[0..1]> = "500 Feet"
searchcursor0 = arcpy.SearchCursor(featureclass1); hascursor0[featureclass1] = True
newlayer<[0..1]> = "l<[1..3]>"
spatialsort0 = "LR"
gdbfilelist0 = sorted(glob.glob(arcpy.env.workspace + "*.gdb"))
sorttype<[0..1]> = "ASCENDING"
classorlayer<[0..3]> = featureclass3
selectiontype0 = "NEW_SELECTION"
selectiontype0 = "CLEAR_SELECTION"
op0 = "!="
arcpy.Union_analysis(classorlayerlist1,featureclass1,join_attributes=joinattributes1); report()
sortlist<[0..1]> = []

Fig. 7 Start depth 20, depth 8, width 3 trace visualization for ArcPy testing.

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 7. 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 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 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

- **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.
- **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.

Fig. 8 Some core methods for testing an SUT.

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 8 (there are many more methods for e..g, code coverage, 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 [32,42], may consider simple modifications of random testing that do not greatly increase overhead. One such example, with implementation, is shown in the original TSTL paper [39]. 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 9.

The implementation is trivial, relying only on the TSTL API and some very simple Python tools (sorting with automatic lexical ordering, time library, 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 [66]. Comparison with simple random testing is easy, since setting `MEMORY` to 0 gives pure memoryless random testing (alternatively,

```
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. 9 Implementing a very simple novel testing algorithm.

to avoid the overhead of the comparisons with 0, a dedicated version for random testing can be written). A major threat in most comparisons of testing algorithms is that different underlying infrastructure for different methods 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 10 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 Faults Discovered Using TSTL

6.1 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 3) some properties that check for data corruption and determinism of GIS analysis, we are not focusing on these until we have a

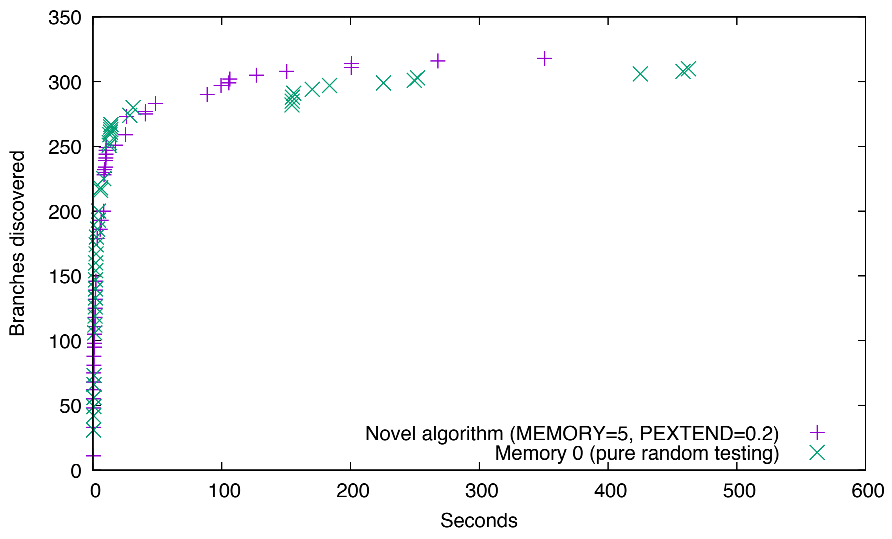

Fig. 10 Comparing branch coverage for 10 minute runs of two test generation methods.

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

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 11). 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 11 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.

6.2 Faults in Other Systems

TSTL has 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, the even more widely-used `OpenCL` infrastructure, 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 involved. Second, TSTL testing found cases where distance metrics that were supposed to be symmetric in the popular fuzzy-string-matching library FuzzyWuzzy 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, used by many professional astronomers and astrophysicists. Finally, TSTL has been used to discover faults in the TSTL API itself.

7 Related Work

There is a vast amount of previous work on automated generation of tests for (API-based) software systems [55,23,27] and random testing in particular [35, 55,8,15,9,60,44,43,17,16,10,37,7,6,20], 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 [4] or recent software testing research in general [54].

To our knowledge, there has been no previous proposal of a concise domain-specific-language [22] like TSTL, to assist users in building test harnesses. There is limited previous work on building common frameworks for random testing and model checking [37], or proposing common terminology for imperative harnesses [30]. Earlier publications on TSTL itself [39,40] 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 [41], and provided little practical guidance to potential users of TSTL. However, some interesting details of the current TSTL implementation were presented in the NASA Formal Methods paper [39] 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: e.g. Korat [53], which has a much more fixed input domain specification, or the tools built to support the Next Generation Air Transportation System (NextGen) software [25]. The closest of these is the UDITA language [26], an extension of Java with non-deterministic choice operators and `assume`, which yields a very different language that shares our goal. TSTL aims more at the *generation* of tests than the *filtering* of tests (as defined in the UDITA paper), while UDITA supports both approaches. This goal of UDITA (and resulting need for first-class `assume` statements) means that it must be hosted inside a complex (and sometimes non-trivial to install/use) tool, JPF [62], 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.

The design of the SPIN model checker [47] and its model-driven extension to include native C code [45] 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 [34] 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 [5], and common to some other test generators [55].

Some recent work on automated test generation has given more attention to practical, rather than primarily algorithmic, 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 [35,36,33] have a largely practical focus, and the work of Lei and Andrews [50] emphasizes the need for delta-debugging in realistic random testing. We found that test case readability was important in our efforts, and recent academic work [19,18] 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 [51]. 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., [64,3]), 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 [28], 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 [12,13]. 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 [59]: 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 [57,58]. Other work has focused on errors end-users make in specifying systems [56], 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 [38,49,61]. 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.

8 Conclusions

This paper presents the latest version of the TSTL [39–41] 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.

References

1. http://www.cs.cmu.edu/ modelcheck/cbmc/
2. JPF: the swiss army knife of Java(TM) verification. http://babelfish.arc.nasa.gov/trac/jpf
3. AbSharma: Functional testing of GIS applications (automated testing). http://osgeo-org.1560.x6.nabble.com/Functional-Testing-of-GIS-applications-Automated-Testing-td4493673.html
4. 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)
5. Andrews, J., Zhang, Y.R., Groce, A.: Comparing automated unit testing strategies. Tech. Rep. 736, Department of Computer Science, University of Western Ontario (2010)
6. Andrews, J.H., Groce, A., Weston, M., Xu, R.G.: Random test run length and effectiveness. In: Automated Software Engineering, pp. 19–28 (2008)
7. 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)
8. 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)
9. Arcuri, A., Briand, L.: Adaptive random testing: An illusion of effectiveness. In: International Symposium on Software Testing and Analysis, pp. 265–275 (2011)
10. 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)
11. Bentley, J.: Programming pearls: little languages. Communications of the ACM **29**(8), 711–721 (1986)
12. Burnett, M., Cook, C., Rothermel, G.: End-user software engineering. Comm. ACM **47**(9), 53–58 (2004)
13. Burnett, M.M., Myers, B.A.: Future of end-user software engineering: beyond the silos. In: Future of Software Engineering, pp. 201–211 (2014)
14. 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)
15. Chen, T.Y., Leung, H., Mak, I.K.: Adaptive random testing. In: Advances in Computer Science, pp. 320–329 (2004)
16. 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)
17. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of haskell programs. In: ICFP, pp. 268–279 (2000)

18. 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)
19. 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)
20. Duran, J.W., Ntafos, S.C.: Evaluation of random testing. IEEE Transactions on Software Engineering **10**(4), 438–444 (1984)
21. Esri: What is ArcPy? http://resources.arcgis.com/EN/HELP/MAIN/10.1/ index.html000v000000v7000000
22. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010)
23. 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)
24. Gamma, E., Beck, K.: JUnit. http://junit.sourceforce.net
25. 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)
26. 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)
27. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Programming Language Design and Implementation, pp. 213–223 (2005)
28. GRASS Development Team: Testing GRASS GIS source code and modules. https://grass.osgeo.org/grass71/manuals/libpython/gunittest_testing.html
29. 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)
30. Groce, A., Erwig, M.: Finding common ground: choose, assert, and assume. In: Workshop on Dynamic Analysis, pp. 12–17 (2012)
31. 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)
32. 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)
33. 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)
34. 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)
35. 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)
36. 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)
37. 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)
38. 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)
39. Groce, A., Pinto, J.: A little language for testing. In: NASA Formal Methods Symposium, pp. 204–218 (2015)
40. 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)

41. Groce, A., Pinto, J., Azimi, P., Mittal, P., Holmes, J., Kellar, K.: TSTL: the template scripting testing language. https://github.com/agroce/tstl
42. Groce, A., Zhang, C., Eide, E., Chen, Y., Regehr, J.: Swarm testing. In: International Symposium on Software Testing and Analysis, pp. 78–88 (2012)
43. Hamlet, R.: Random testing. In: Encyclopedia of Software Engineering, pp. 970–978. Wiley (1994)
44. Hamlet, R.: When only random testing will do. In: International Workshop on Random Testing, pp. 1–9 (2006)
45. Holzmann, G., Joshi, R.: Model-driven software verification. In: SPIN Workshop on Model Checking of Software, pp. 76–91 (2004)
46. Holzmann, G., Joshi, R., Groce, A.: Model driven code checking. Automated Software Engineering **15**(3–4), 283–297 (2008)
47. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
48. 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)
49. 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)
50. Lei, Y., Andrews, J.H.: Minimization of randomized unit test cases. In: International Symposium on Software Reliability Engineering, pp. 267–276 (2005)
51. Maogui, H., Jinfeng, W.: Application of automated testing tool in GIS modeling. In: World Congress on Software Engineering, pp. 184–188 (2009)
52. McKeeman, W.: Differential testing for software. Digital Technical Journal of Digital Equipment Corporation **10(1)**, 100–107 (1998)
53. 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)
54. 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)
55. 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)
56. 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)
57. 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)
58. 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)
59. Segal, J.: Some problems of professional end user developers. In: IEEE Symp. Visual Languages and Human-Centric Computing (2007)
60. 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)
61. 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)
62. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10**(2), 203–232 (2003)
63. 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)
64. XBOSOFT: GIS software testing - lessons learned. http://xbosoft.com/gis-software-testing-lessons-learned/
65. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. Software Engineering, IEEE Transactions on **28**(2), 183–200 (2002)
66. 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)