A Little Language for Testing

Alex Groce and Jervis Pinto

School of Electrical Engineering and Computer Science
Oregon State University, Corvallis, OR

Abstract. The difficulty of writing test harnesses is a major obstacle to the adoption of automated testing and model checking. Languages designed for harness definition are usually tied to a particular tool and unfamiliar to programmers; moreover, such languages can limit expressiveness. Writing a harness directly in the language of the software under test (SUT) makes it hard to change testing algorithms, offers no support for the common testing idioms, and tends to produce repetitive, hard-to-read code. This makes harness generation a natural fit for the use of an unusual kind of domain-specific language (DSL). This paper defines a *template scripting* testing language, TSTL, and shows how it can be used to produce succinct, readable definitions of state spaces. The concepts underlying TSTL are demonstrated in Python but are not tied to it.

1 Introduction

Building a test harness is an often irksome task many users of formal methods or automated testing face from time to time [18,12]. The difficulty of harness generation is one reason for the limited adoption of sophisticated testing and model checking 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 Software Under Test (SUT). This is obviously how most unit tests are written, as witnessed by the proliferation of tools like JUnit [3] and its imitators (e.g., PyUnit, HUnit, etc.). It is also how many industrial-strength random testing systems are written [17,15]. A KLEE "test harness" [6] 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 [2,28] can easily be seen as offering a way to define a state space using Java itself as the modeling language, and CBMC [1,24] 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 [29].

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 way guards and choices are interleaved means that the state-space will be pointlessly

```
op = choice(operations);
val1 = choice(values);
val2 = choice(values);
switch (op) {
case op1:  if (guard1)
              call1(val1);
           break;
case op2:  if (guard2)
              call2(val1,val2);
           break;
case op3:  if (guard3)
              call3(val1,val3);
           break;
```

Fig. 1. A test harness in the SUT language

```
heap()                : returns a new heap
heap.insert(key,val)  : inserts a key with value, returning ref
heap.union(heap)      : merges two heaps
heap.extractMin()     : extracts the minimum
ref.delete()          : given a reference to a node, deletes it
ref.decreaseKey(key)  : decreases the key of ref's node
```

Fig. 2. A binomial heap to test

expanded to include many action and value choices that don't produce any useful behavior. This harness also always assigns `val2` even though `call1` only uses `val1`, to avoid having to repeat the choice code for calls 2 and 3. Moreover, this harness is possibly sub-optimal for a method such as 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 or model checking a binomial heap that supports several operations, defined in Figure 2. Such a harness must manage the creation and storage of values of multiple types, including heaps and references. Moreover, because building up heaps and references is complicated, they cannot simply be produced on each iteration, but must be remembered. As the interactions of multiple heaps (via `union`) and references into a heap are the source of all interesting behavior, the harness needs to decide how many heaps and references to store. The code quickly becomes hard to read, hard to maintain, and hard to debug. In some cases [15] 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 (such as how to handle storing heaps and references) that would ideally be configurable.

The definition of a harness also tends to be intimately tied to a single tool, with the only testing strategies available being those provided by that tool. 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 [14]. Such a harness

cannot support model checking or any sophisticated strategy without being rewritten if it is in a language like Python without verification tool support.

What the user really wants is to simply provide the information in Figure 2, some configuration details (e.g., how many `ref`s to keep around), and some information on which testing method to use (e.g., model checking, random testing, machine-learning based approaches). Some automated testing tools for Java [8,27] take a variation on this approach, automatically extracting the signatures of methods from source code and testing them. Unfortunately, completely automatic extraction 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. The user wants declarative harnesses, but often needs to program the details of a harness.

1.1 Domain Specific Languages for Testing

The properties of the problem at hand suggest the use of a *domain-specific language* (DSL) [13]. DSLs [7] 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" in computer science, as memorably advocated by Jon Bentley in one of his famous Programming Pearls columns [5] 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 [10,14,28,24,6]. Neither of these choices is quite right for harness definition. Simply adding operations for nondeterministic choice, as is done in most cases, still leaves most of the tedious work of harness definition to the user, and makes changing testing approaches difficult at best. 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 [23]. Version 4.0 of SPIN [21] made use of SPIN's nature as a tool that *outputs a C program* 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 [15,22]. 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 [16]. We adopt this approach: embed a general-purpose language (for expressiveness) in a DSL (for concision and ease-of-use).

1.2 Template Scripting

In previous discussions, a harness has been thought of as imperative code that tests a system, even when the underlying use is more declarative, as in CBMC,

```
@import bh
pool: %INT% 4
pool: %HEAP% 3
pool: %REF% 4
%INT%:=%[1..20]%
%HEAP%:=bh.heap()
%REF%:=%HEAP%.insert(%INT%,%INT%)
%HEAP%.insert(%INT%,%INT%)
%HEAP%.union(%HEAP%)
%HEAP%.extractMin()
%REF%.delete()
%REF%.decreaseKey(%INT%)
```

Fig. 3. A simple harness definition for a binomial heap

or as a purely declarative model stating the available test operations, in which case the harness is often hidden from the user and generated by a tool. In this paper, we propose thinking of a harness as a *declaration of the possible actions the SUT can take*, but 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 in an imperative fashion*. 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 in the SUT's language for a test harness, much like SPIN. The harness description file consists of fragments of code in the SUT language that are expanded, via code-generation, into executable source code. The tool that outputs code basically defines a *template* for test harnesses in a programming language, and the harness definition tells the tool how to instantiate that template. Rather than generating a testing tool, our method outputs *a class defining a search space*. The *scripting* aspect simply means that our language is meant to be very lightweight, and assumes a host language without a rigorous type system (e.g. Python) or with effective type-inference (e.g. Scala), making minimal demands on the user. The design of the language also relies on the very-high-level nature of code in scripting languages, making the harness concise but expressive, and making "one-liners" of action definition possible.

Figure 3 shows a complete harness definition for the binomial heap class defined in Figure 2. The example is easily understood by splitting it into three sections. First, the single line proceeded by an "@" is raw Python, inserted into the output harness with no modification in most cases. This section can be used not only to import the SUT's code, but to define functions to be used in the body of the harness, as we will see below. Second, the lines beginning with `pool:` define the "pool" [27,10,4] of values that will be used during testing. In model checking terms, these store the state of the SUT. There is no type information here, because the template approach simply assumes the type system of the host language, but in an informal sense each pool value typically represents its own type in the template language, as shown by its usage below (a pool value will correspond to inputs of a particular type to method calls, in the most trivial instance, but can also be used to encode more fine-grained type distinctions not

```
import bh as b
class t(object):
   def act0(self):
      self.p_INT[0]=1
      self.p_INT_used[0]=False
   def guard0(self):
      return (self.p_INT_used[0])
...
   def act87(self):
      self.p_REF[0]=self.p_HEAP[0].insert(self.p_INT[1],self.p_INT[0])
      self.p_INT_used[1]=True
      self.p_INT_used[0]=True
      self.p_REF_used[0]=False
      self.p_HEAP_used[0]=True
   def guard87(self):
      return (self.p_INT[1] != None) and (self.p_INT[0] != None) and
        (self.p_REF_used[0]) and (self.p_HEAP[0] != None)
...
      self.actions.append((r"self.p_INT[0]=1",self.guard0,self.act0))
```

Fig. 4. Fragments of Python code for binomial heap harness

present in the host language). The numbers indicate how many values of a given pool "type" are needed. Here, at least two `INT`s are needed, unless both values provided to `insert` should always be the same. Similarly, there need to be at least two `HEAP`s if `union` is to be tested effectively. Because the performance of random testing and some learning algorithms depends heavily on pool sizes, we want to make it easy to experiment with them.

Finally, the remainder of the harness definition simply gives possible actions, one on each line. Each line is expanded into Python code for 1) the actual test action represented and 2) a guard that determines if that action is enabled, as shown in Figure 4. The functions for actions and guards are then added to a list that stores all possible SUT actions, with no remaining nondeterminism unless the SUT provides it. Nondeterminism is controlled by choosing which actions (whose guards are currently satisfied) to execute. Even in the absence of user-defined guards, some guards are automatically generated. First, no *uses* of a pool value are allowed until that value has been assigned (the generated harness initializes pool values as `None`, a special Python value). Second, no pool value can be assigned to unless it is either uninitialized or has been *used* at least once. This is critical to avoid the potential for some test strategies (such as random testing) to repeatedly perform useless assignments to values used in the actual testing (e.g., `INT[1] = 1` followed immediately by `INT[1] = 2`. Figure 5 shows an example of a test that can be generated by this harness. Note that assigning anything to `INT[3]`, `REF[0]` or `REF[1]` is not valid after the final action of the test, as these pool values have been assigned but not used.

2 The Template Scripting Testing Language (TSTL)

Figure 6 shows a BNF-style specification of the Template Scripting Testing Language (TSTL). Processing a harness definition involves iterating through the lines in the file and performing a set of transformations that result in an output

```
self.p_INT[1]=9
self.p_INT[2]=1
self.p_INT[0]=18
self.p_HEAP[0]=b.heap()
self.p_REF[2]=self.p_HEAP[0].insert(self.p_INT[0],self.p_INT[2])
self.p_INT[3]=17
self.p_INT[0]=18
self.p_REF[0]=self.p_HEAP[0].insert(self.p_INT[0],self.p_INT[1])
self.p_REF[0].decreaseKey(self.p_INT[1])
self.p_INT[1]=19
self.p_REF[1]=self.p_HEAP[0].insert(self.p_INT[0],self.p_INT[1])
self.p_REF[0]=self.p_HEAP[0].insert(self.p_INT[0],self.p_INT[1])
self.p_HEAP[1]=b.heap()
self.p_HEAP[1].union(self.p_HEAP[0])
```

Fig. 5. A valid action sequence (test) for the binomial heap harness

```
<template> ::= <template-line> EOL <template> | EOF
<template-line> ::= <raw> | <pool> | <property> | <init> |
                    <feature> | <reference> | <compare> | <action>
<raw> ::= @ <raw-code>
<pool> ::= pool: %<ID>% <INT> [REF]
<property> ::= property: <simple-code>
<init> ::= init: <simple-code>
<feature> ::= feature: <regexp>
<reference> ::= reference: <regexp> ==> <text>
<compare> ::= compare: <regexp>
<action> ::= <text> | <lhs> := <rhs> | guardedFN(<simple-code>)
<raw-code> ::= <text> | def guardedFN(<text>) | %COMMIT%
<lhs> ::= <simple-code>
<rhs> ::= <simple-code>
<simple-code> ::= <text> | <simple-code> <ID-use> <simple-code> |
                  <simple-code> <range> <simple-code>
<ID-use> ::= %<ID>% | ~%<ID>%
<range> ::= %[INT..INT]%
```

Fig. 6. The Template Scripting Testing Language in Pseudo-BNF

file that defines a class in the target language (Python in our current implementation). This class itself performs no testing; it instead defines an interface to a definition of the available actions of the SUT that any testing algorithm can use, shown in Table 1.[1] The methods in this interface are not defined by the user, but automatically generated by the TSTL "compilation." The basic transformation algorithm is relatively simple (our implementation for Python is less than 1,000 lines of code):

1. Output the `<raw>` Python code, transforming guarded functions into expanded Python code as described in Section 2.2.
2. Collect the set of pool values, properties, initialization code, features, references, and comparisons.
3. Replace each pool ID in actions and properties with the pool ID plus a `<range>` from 0 to the pool size - 1.

[1] This is not the entire set of methods TSTL compilation automatically generates: there are also methods for swarm testing [20], generalized delta-debugging [30,11], code coverage, and other common testing needs.

Method	Type	Purpose
`restart`	`()→()`	resets pools, executes `<init>` code
`actions`	`()→[(str,()→bool,()→())]`	returns a list of all possible actions
`enabled`	`()→[(str,()→bool,()→())]`	returns actions with True guard
`check`	`()→bool`	executes `<property>` assertions
`state`	`()→STATE`	returns deep copy of pool values
`replay`	`[(str,()→bool,()→())] → bool`	replays a test, returns whether it failed
`backtrack`	`STATE→()`	sets pools to STATE

Table 1. SUT Class Methods

4. Recursively expand each action and property range, creating copies with each value in the range instantiated. At this point all actions should be deterministic[2]
5. Collect assignments and uses from actions; assignments are IDs on the `lhs` of a `:=`; uses are IDs appearing in an action, such that ID is not an assignment or marked with a ˜.
6. Generate guards for each action: first, ensure no values are used that have no value; second, ensure no assignments to values that have a value that has not been used are made; third, add any guarded function calls as extra guards (see Section 2.2).
7. For any actions involving pools marked as `ref`, copy with reference.
8. Apply all transformations indicated by `<reference>` (text matching `regexp` is replaced by the given `text`), then add an assertion of equal return values for any transformed code that matches a `<compare>` regexp.
9. Perform any language-specific transformations.

Due to lack of space here, we cannot elaborate on every aspect of TSTL. Instead, we present some example uses to highlight salient features.

```
@def guardedAppend(l,item,limit):
@  if len(l) >= limit:
@    return False
@  %COMMIT%
@  l.append(item)
property: (len(%LIST%) < 10) or (6 not in %LIST%)
property: %VAL% != -1
pool: %LIST% 1
pool: %VAL% 1
%LIST% := []
guardedAppend(~%LIST%,%VAL%,10)
%VAL% := %[1..10]%
print %LIST%
```

Fig. 7. A toy bounded list generation example (`%COMMIT%` is expanded into a check for when the function is used as a guard)

[2] assuming the SUT itself is determinstic

```
@import avl
@import bintree
pool: %INT% 4
pool: %AVL% 1 REF
%INT%:=%[1..20]%
%AVL%:=AvlTree()
%AVL%.insert(%INT%)
%AVL%.delete(%INT%)
%AVL%.find(%INT%)
reference: AvlTree() ==> BinTree()
compare: find
```

Fig. 8. Using a reference as oracle

2.1 Oracles

TSTL handles test oracles in two ways. First, users can specify properties that the `check` method will automatically verify using assertion statements, expanded for each pool item involved. Figure 7 shows how properties are defined, in this case with quite trivial properties. Note that because raw Python can be used, and properties can call arbitrary Python code, it is easy to encode even complex specifications by defining a Python function that takes the pool values of interest as input and returns a Boolean, then adding it as a property. A second popular approach to the oracle problem is differential testing [25], also known as testing with a reference. TSTL supports this by making it easy to define how to transform actions on the SUT into actions on the reference, and when to compare values from calls to the SUT and reference. Figure 8 shows a simple example, where an AVL tree in Python is tested by comparing its behavior to a simple (unbalanced) binary tree implementation. All that is required to do this is 1) to mark the AVL pool as a `ref` pool, meaning it will have a copy that contains a reference implementation 2) to explain how to transform the call that initialized the AVL tree to initialize the binary tree and 3) to indicate that results from calling `find` on the AVL and reference should be compared. TSTL automatically generates the required code based on this information.

2.2 Guards and Function Calls

As Figure 7 shows, TSTL makes it simple to define functions and call them in actions. Obviously, some actions cannot be expressed as one line of code. In these cases, we expect that the user will define a function whose inputs can be any pool values, constants, etc. and perform more complex tasks. We exploit this feature to implement user-defined guards for actions easily. If a function is named `guardedFN`, where `FN` is a function name, TSTL will automatically add an additional parameter to the function definition when it generates a harness. This parameter indicates whether the call is to actually perform the action, or simply check if the action is enabled. The function definition should check the guard and return `False` if it is not satisfied. At the point where the user indicates that a "real" action is to follow (which typically modifies SUT state) the function definition should include a `%COMMIT%`, which will be replaced with code that checks for a speculative call and simply returns `True` without proceeding if the

call is in a guard context. The translation of the relevant code from Figure 7 is shown in Figure 9, with a comment to indicate where the %COMMIT% was.

```
def guardedAppend(l,item,limit, SPECULATIVE_CALL = False):
  if len(l) >= limit:
    return False
  if SPECULATIVE_CALL: return True                                    # /%COMMIT/%
  l.append(item)
...
   def act1(self):
      guardedAppend(self.p_LIST[0],self.p_VAL[0],10)
      self.p_VAL_used[0]=True
   def guard1(self):
      return (self.p_LIST[0] != None) and (self.p_VAL[0] != None)
        and (guardedAppend(self.p_LIST[0],self.p_VAL[0],10,True))
```

Fig. 9. User-defined guard example, Python code generated

2.3 Miscellaneous Notes on TSTL

In order to effectively test the SUT, it is often important to build up complex values before calling the code under test. Making every appearance of a pool element in an action a use, therefore allowing the value to be reset to its initial state can "suppress" [19] behaviors, even if it does not strictly prevent them. TSTL therefore allows the use of a ~ before a use of a pool ID, as shown in Figure 7 to indicate that a reference to a pool ID should not count as a use, it is simply building up a complex input. Another mitigation for suppression effects is provided by the <feature> definitions, which allow TSTL to support swarm testing [20]. Swarm testing is a random testing approach in which each test disables some randomly chosen API calls or grammar features, in order to better explore the state space of the system. A feature definition indicates that any action matching the regexp is considered an instance of a certain feature, and is disabled if that feature is disabled. TSTL has strong out-of-the-box support for a variety of testing algorithms, some state-of-the-art like swarm testing.

Finally, we note that TSTL is not restricted to API testing. Figure 10 demonstrates TSTL's support for encoding grammars for generating strings. It also provides an example of mixing range values and explicit values in assignment.

2.4 Output Language

The language and tool presented here are not inherently tied to any language. With trivial modifications, the harness maker could output Scala code instead of Python. In principle, C or Java could also serve as the base for the DSEL. In fact, it should be simple to output PROMELA models with embedded C, given a harness with C as the base language, though maintaining the "declarative" approach would make the PROMELA somewhat difficult to read (each SPIN nondeterministic choice would need to pick the nth action, with the guards

```
@import sys
@import calculator as c
pool: %EXPR% 7
pool: %NUM% 5
%NUM% := '%[-100..100]%'
%NUM% := str(sys.maxint)
%NUM% := str(-sys.maxint - 1)
%EXPR% := %NUM%
~%EXPR% = '(' + ~%EXPR% + ')'
~%EXPR% = ~%EXPR% + '+' + ~%EXPR%
~%EXPR% = ~%EXPR% + '*' + ~%EXPR%
~%EXPR% = ~%EXPR% + '-' + ~%EXPR%
~%EXPR% = ~%EXPR% + '/' + ~%EXPR%
c.calculate(%EXPR%)
reference: c.calculate ==> eval
compare: calculate
```

Fig. 10. Harness for a simple calculator class

```
t = SUT.t()
for ntests in xrange(1,config.maxtests+1):
    t.restart()
    test = []
    for s in xrange(0,config.depth):
        (name,guard,act) = random.choice(t.enabled())
        test.append(name)
        act()
        if not t.check():
            print "FAILED TEST:", test
            sys.exit(1)
print ntests, ``SUCCESSFUL''
```

Fig. 11. A simple random tester

being the enabled check). Python was chosen for several reasons: first, it is a widely adopted language in the real world, particularly in the testing community. Second, Python programs can particularly benefit from more effective automated testing because the lack of a good type system means Python code may fail in surprising and frustrating ways.

3 Using the Harness to Test and Experiment

It is simpler to show how the interface described in Table 1 is used than to explain each method. Figure 11 shows the core of the implementation of a pure random tester for an arbitrary SUT, omitting boilerplate such as import statements, command-line option parsing, and checking for timeout. A few points are important: first, the test algorithm is entirely SUT-agnostic. All interaction with the SUT is performed through the API in Table 1. The use of pools and the `(name, guard, action)` tuple list reduces the complex problem of choosing values and operations as shown in Figure 1 to the uniform simplicity of picking one enabled action and calling it as a function, storing the `name` as a human-readable identifier for the test behavior. Note that when reference oracles are

used, the call to `act` is also typically enclosed in a `try` block to record the failing test, as is done with `check`.

```
t = SUT.t()
t.restart()
visited = []
S = []
S.append(t.state(), [])
test = []
while S != []:
    (v, test) = S.pop()
    t.backtrack(v)
    if (v not in visited) and (len(test) < config.maxdepth):
        visited.append(v)
        trans = t.enabled()
        for (name, guard, act) in trans:
            test.append(name)
            act()
            if not t.check():
                print "FAILED TEST:", test
                sys.exit(1)
            S.append((t.state(), test))
```

Fig. 12. A really simple DFS-only model checker for safety properties

Perhaps more impressively, a natural consequence of encoding a state space is that we can easily implement a (very simple) model checker, as shown in Figure 12. Of course, as a model checker it is highly inefficient, since the visited check is implemented as a linear search through a list of visited states. The inefficient linear search can be easily improved through the use of a hash table for pool states. TSTL makes use of Python's `deepcopy` functionality to automatically provide backtracking for many SUTs. To our knowledge, no other frameworks makes it as easy to use either backtracking or replay for state restoration as TSTL. State copies are often more efficient than replay. However, for simple SUTs and shallow depths replay may be better, and it works for some hard-to-copy SUTs.

Exploring Novel Testing Algorithms: In order to demonstrate how TSTL facilitates the design and evaluation of testing approaches, we provide the following simple algorithm motivated by classical beam search. Note that we do not claim this algorithm is highly effective in general, the point is to show that it is extremely easy to implement and compare new algorithms using TSTL.

Figure 13 shows a modified random testing algorithm. It performs almost like traditional random testing (as shown in Figure 11) except for the following change: at each step of the test, the state is saved. Instead of randomly selecting a single enabled action at random, this strategy picks k actions at random, and tries each one (backtracking to the old state after each action but the final one). However, if any action covers a never-before-explored branch of the SUT, it is chosen and testing proceeds to the next step immediately.[3] In less than thirty

[3] The coverage analysis is provided in this example by a simple Python coverage instrumentation tool, but TSTL offers integration with the very popular `coverage.py` as well.

```
t = SUT.t()
coverTool.clearCoverage()
for ntests in xrange(1, config.maxtests+1):
    t.restart()
    test = []
    print ntests+1, len(coverTool.getCoverage())
    for s in xrange(0,config.depth):
        possible = t.enabled()
        random.shuffle(possible)
        old = t.state()
        cov = coverTool.getCoverage()
        last = min(config.k, len(possible))
        pos = 0
        for (name, guard, act) in possible[:config.k]:
            pos += 1
            test.append(name)
            act()
            if not t.check():
                print "FAILING TEST:", test
                sys.exit(1)
            covNew = coverTool.getCoverage()
            if (pos == last) or (len(covNew) > len(cov)):
                break
            coverTool.setCoverage(cov)
            t.backtrack(old)
```

Fig. 13. A novel random testing algorithm, which is essentially random beam search, demonstrating the use of backtracking for state restoration

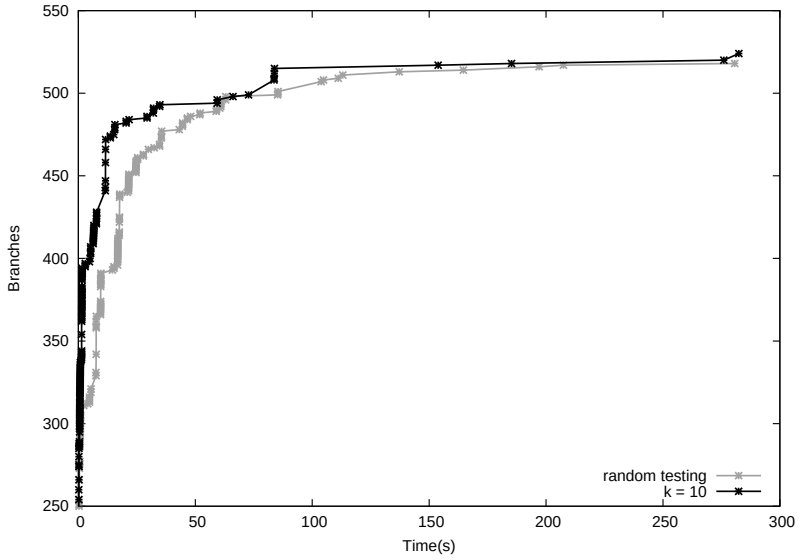

Fig. 14. Testing time vs. branches seen, traditional random testing vs. beam-search like method with $k = 10$, for Dominion simulator

minutes, we modified the random tester to perform this algorithm and both it and the default random tester to output the time at which each new branch is first covered during testing. Figure 14 shows the branch discovery rate of random testing compared to the novel test harness based on beam-search. The SUT (an implementation of strategy simulation for the card game Dominion) was taken from our work on applying machine learning to test generation, where it had proven difficult to improve on random testing. The experiment shows that, for this subject, the curve of covered code increases much more rapidly using the modified beam search than with traditional random testing. This simple experiment shows the ease with which researchers can explore novel testing strategies in TSTL. The benefits of providing backtracking are also evident here — other experiments show the same algorithm using replay performs considerably worse on average, at the test lengths required for good code coverage.

4 Related Work

To our knowledge, there has been no previous proposal of a concise *language* like TSTL to assist users in building test harnesses. One line of related work is our own previous work on building common frameworks for random testing and model checking [18] and proposing common terminology for imperative harnesses [12]. Work on domain-specific languages also informed our approach [7].

There exist various testing tools and languages of a somewhat different flavor: e.g. Korat [26], which has a much more fixed input domain specification, or the tools built to support the Next Generation Air Transportation System (NextGen) software [9]. The closest of these is the UDITA language [10], 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`) means that it is hosted inside a complex (and sometimes non-trivial to install/use) tool, JPF [28], 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 also supports far fewer constructs to assist test harness development.

The design of the SPIN model checker [23] and its model-driven extension to include native C code [21] inspired our flavor of domain-specific language, though our approach is more declarative than the "imperative" model checker produced by SPIN. Similarly, work at JPL on languages for analyzing spacecraft telemetry logs in testing [16] provided a working example of a Python-based declarative language for testing purposes. The pool approach to test case construction is derived from work on canonical forms and enumeration of unit tests [4].

5 Conclusions and Future Work

We believe that the little language defined in this paper could be of considerable use to software developers who would like to use more automated testing, but do

not want to learn complex new languages and tools. We expect that it also will prove useful to researchers who would like to rapidly prototype new testing and model checking methods and easily try their ideas out on new SUTs. The use of a template language makes it easy to exploit the usability of a scripting language, and the declarative approach makes implementing new testing algorithms easy.

Our future work is to further develop the TSTL language and tool, based on other users' experiences. One goal is to make use of TSTL easy out-of-the-box, which means including many example harnesses, SUTs, and testing algorithms. A second task is to improve the core language to include more functionality. For example, one obvious language omission is the inability to express desired probabilities for random testing. More automatic ranges, or a shorthand for including multiple concrete values as choices on one line for grammar encoding would also be useful. We also plan to extend TSTL to handle more host languages, including Scala, Java, C (possibly including use of KLEE [6]), and PROMELA. Additionally, we plan to use TSTL as a basis for further research in using machine learning techniques to improve software testing [13]. A development version of TSTL is available at `https://code.google.com/p/harness-maker`.

Acknowledgments: The authors would like to thank Klaus Havelund, Gerard Holzmann, Rajeev Joshi, John Regehr, Alan Fern, Martin Erwig, and the anonymous NFM'15 reviewers for their comments and ideas. A portion of this research was funded by NSF CCF-1217824 and NSF CCF-1054876.

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. JUnit. `http://junit.sourceforce.net`
4. Andrews, J., Zhang, Y.R., Groce, A.: Comparing automated unit testing strategies. Tech. Rep. 736, Department of Computer Science, University of Western Ontario (December 2010)
5. Bentley, J.: Programming pearls: little languages. Communications of the ACM 29(8), 711–721 (1986)
6. 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)
7. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010)
8. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. pp. 416–419. ESEC/FSE '11, ACM (2011)
9. 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)
10. 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)

11. 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)
12. Groce, A., Erwig, M.: Finding common ground: choose, assert, and assume. In: Workshop on Dynamic Analysis. pp. 12–17 (2012)
13. Groce, A., Fern, A., Erwig, M., Pinto, J., Bauer, T., Alipour, A.: Learning-based test programming for programmers pp. 752–786 (2012)
14. 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)
15. 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)
16. 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)
17. 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)
18. 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)
19. Groce, A., Zhang, C., Alipour, M.A., Eide, E., Chen, Y., Regeher, J.: Help, help, I'm being suppressed! the significance of suppressors in software testing. In: IEEE International Symposium on Software Reliability Engineering. pp. 390–399 (2013)
20. Groce, A., Zhang, C., Eide, E., Chen, Y., Regehr, J.: Swarm testing. In: International Symposium on Software Testing and Analysis. pp. 78–88 (2012)
21. Holzmann, G., Joshi, R.: Model-driven software verification. In: SPIN Workshop on Model Checking of Software. pp. 76–91 (2004)
22. Holzmann, G., Joshi, R., Groce, A.: Model driven code checking. Automated Software Engineering 15(3–4), 283–297 (2008)
23. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
24. 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)
25. McKeeman, W.: Differential testing for software. Digital Technical Journal of Digital Equipment Corporation 10(1), 100–107 (1998)
26. 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)
27. 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)
28. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering 10(2), 203–232 (Apr 2003)
29. 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)
30. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. Software Engineering, IEEE Transactions on 28(2), 183–200 (2002)