Finding Common Ground: Choose, Assert, and Assume

Alex Groce
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR, USA
alex@eecs.oregonstate.edu

Martin Erwig
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR, USA
erwig@eecs.oregonstate.edu

ABSTRACT

At present, the "testing community" is on good speaking terms, but typically lacks a common language for expressing some computational ideas, even in cases where such a language would be both useful and plausible. In particular, a large body of testing systems define a testing problem in the language of the system under test, extended with operations for *choosing inputs*, *asserting properties*, and *constraining the domain of executions considered*. While the underlying algorithms used for "testing" include symbolic execution, explicit-state model checking, machine learning, and "old fashioned" random testing, there seems to be a common core of expressive need. We propose that the dynamic analysis community could benefit from working with some common syntactic (and to some extent semantic) mechanisms for expressing a body of testing problems. Such a shared language would have immediate practical uses and make cross-tool comparisons and research into identifying appropriate tools for different testing activities easier. We also suspect that considering the more abstract testing problem arising from this minimalist common ground could serve as a basis for thinking about the design of usable embedded domain-specific languages for testing and might help identify computational patterns that have escaped the notice of the community.

Categories and Subject Descriptors

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

General Terms

Languages, Verification

Keywords

Random testing, model checking, symbolic execution, domain specific languages

WODA '12, July 15, 2012, Minneapolis, MN, USA

1. INTRODUCTION

Figure 1 shows 6 code fragments, covering four languages, six tools/libraries, and six different algorithms or programs. Figure 1a shows part of the Java code for a test harness that can perform either random testing [22] or reinforcement-learning based testing [15] using adaptation-based programming [5]. Figure 1b shows a portion of a PROMELA [23] model checking harness enabling the SPIN model checker to call C code for a file system used in a NASA mission [19], usable for either random testing or model checking. Figure 1c shows code for symbolic verification [32] of a binary tree class using NASA's open source Java Pathfinder model checker [2]. Figure 1d shows an example from the reference manual for the CBMC bounded model checker for C programs [25]. Figure 1e shows an example taken from the tutorial for the KLEE symbolic execution system [6], testing a regular expression library [3]. Finally, Figure 1f shows Haskell code using the QuickCheck testing library [7], taken from haskell.org's introduction to QuickCheck [1].

The differences between these examples are significant, but at a high level all of these fragments define (and the figure highlights):

- a *choice*
- a (possibly implicit) set of properties to be checked over the executions resulting from those inputs,
- and a set of (possibly empty) constraints over the inputs or executions to be generated.

In other words, these examples are all, essentially, test harnesses. Moreover, they are (with one partial exception) test harnesses that are largely written in the language of the software under test (SUT). The first and last examples (Figures 1a and 1f) are simply programs in the host language, using a library for testing; the other four examples all require special tools and/or compilation.

This paper proposes the following: the differences between these examples are more accidental than essential; in particular, the information contained in these code fragments is largely independent of the test generation (or verification) algorithm to be used. Therefore, a common method for expressing this kind of problem — testing or verification via nondeterministic choice — should be agreed upon by the testing community, and to the degree possible, used in all tools that accept the kind of "embedded" test/verification specification shown here, where the input is (almost) a program in the language of the System Under Test (SUT).

```
public static void
  testAvlTree (Object currentSUT,
               String S) {
  AvlTree SUT =
    (AvlTree)currentSUT;
  int v = chooseVal(S).ordinal();
  switch (chooseOp(S)) {
  case INSERT:
    SUT.insertElem(v);
    break;
  case DELETE:
    SUT.remove(v);
    break;
  case FIND:
    Object r = SUT.find(v);
    break;
  }
}
```

(a) ABP/Random

```
pick(pathindex, NUM_PATHS);
c_code {
  enter_nvfs();
  now.res =
    nvfs_unlink(path[now.pathindex]);
  now.nvfs_errno = errno;
  leave(); }
...
  c_code{
    enter_ramfs();
    now.ramfs_res =
      ramfs_unlink(path[now.pathindex]);
    now.ramfs_errno = errno;
    leave(); }
:: else -> skip
...
assert(res==ramfs_res);
assert(nvfs_errno==ramfs_errno);
```

(b) SPIN

```
static BinTree t = new BinTree()
public static void main(String[] args) {
  ...
  for (int i = 0; i < M; i++) {
    Verify.beginAtomic();
    String vn = "v" + i;
    SymbolicInt v =
      new SymbolicInt(vn);
    switch(Verify.random(1)) {
      case 0:  t.add(v); break;
      case 1:  t.remove(v); break;
    }
    Verify.endAtomic();
    Verify.ignoreIf
       (checkSubsumptionAndStore(t));
  }
}
```

(c) JPF

```
unsigned char nondet_uchar();
int main() {
  unsigned char a, b;
  unsigned int result=0, i;
  a=nondet_uchar();
  b=nondet_uchar();
  for(i=0; i<8; i++)
    if((b>>i)&1)
      result+=(a<<i);
  assert(result==a*b);
}
```

(d) CBMC

```
int main() {
  // The input regular expression.
  char re[SIZE];
  // Make the input symbolic.
  klee_make_symbolic(re, sizeof re,
                     "re");
  klee_assume(re[SIZE - 1]
              == '\0');
  // Try to match against a
  // constant string "hello".
  match(re, "hello");
  return 0;
}
```

(e) KLEE

```
import Data.Char
import Test.QuickCheck
instance Arbitrary Char where
    arbitrary =
      choose ('\32', '\128')
    coarbitrary c =
      variant (ord c `rem` 4)
deepCheck p = check (defaultCon-
fig  configMaxTest = 10000) p
deepCheck
  (\s -> length (take5 s) < 5)
```

(f) QuickCheck

Figure 1: **"Test code" for ABP/Random, SPIN, JPF, CBMC, KLEE, and QuickCheck. Shaded code is `choose/assert/assume` equivalent.**

1.1 Test Generation vs. Test Programming

The abstract commonality we believe to be present in the six examples in Figure 1 does not, of course, cover all interesting testing problems well. It is reasonable to distinguish between test generation in general, which considers test cases as "inputs to a program" and the style above, which we would like to call "test programming." In automated test generation algorithms, it is natural to think of two inputs to a testing tool: a program to run and a definition of possible inputs to send that program, from which the tool will select test cases. The input definition will often be "baked into" the generation system, for complex real-world testing systems. As an example, the Csmith compiler testing system [33] is a standalone tool that takes as input parameters a set of configuration options determining which C language features to use, and a random seed, and *outputs a C program*. This is test generation, but does not map well into the `choose/assert/assume` framework considered in this paper. Grammar-based testing in general has a different flavor than API-based testing. We therefore consider *test programming* as the subset of test generation problems that are based on a framework that resembles a kind of generalized unit test, or "test harness" as it is often referred to in the literature:

```
while (!done) {
  operation = choose();
  input = choose();
  assume(precondition);
  operation(input);
  assert(postcondition);
}
```

In fact, this generic harness is quite close to the actual structure of a large number of API-call based testing systems in explicit-state model checking and random testing, including the vast array of container-class testing frameworks [30]. Bounded exhaustive testing and other approaches also fit this framework. Symbolic execution based systems often feature shorter, fixed-length sequences of API calls (or fix the operation choice, but not the inputs [18]), due to the high cost of constraint solving for long sequences of calls. The term "test programming" arises from the fact that the test generation here is, at least in concept, a side effect of running the generalized unit test program. Moreover, many examples, particularly for random testing and model checking, add a large amount of hand-coded test manipulation, feedback [29, 17], etc. to the harness, making it much more of a "program" (albeit with `choose/assert/assume`) than simply a stylized way of writing a grammar for API calls mixed with a specification.

2. THE SIMPLE, PRACTICAL CASE FOR A COMMON LANGUAGE

In a sense, the most obvious reason to propose developing a common language for test programming problems is purely

negative: there does not seem to be a compelling reason to have a number of different languages that greatly exceeds the number of programming languages targeted by tools. In fact, C and Java share a sufficiently similar semantics and syntax at this level of abstraction that we suspect they could also benefit from a unified representation.

Figure 2 hypothesizes the re-creation of the C-language examples in Figure 1 in a common form, where the choices to be made, the restrictions on relevant executions, and the assertion of properties are all made in a common language. Ideally, any tool for C-language test programming could use this format as an input, perhaps (trivially, in most cases, we suspect) rewriting it in the form first shown. Furthermore, given such a language, it would be relatively easy for all the tools shown to produce outputs in a common format, such that compiling the programs with a single tool for *test execution* could run a test case defined by any of these systems, with the choice operator always reading in the next value stored in a file. At present, moving from a "test case" in some of these tools (e.g. CBMC) to something executable in a debugger is actually difficult, despite being conceptually trivial.

Having to learn multiple embedded languages when one would suffice needlessly complicates the use of testing and verification tools and makes it more difficult to convince users outside the research community to use our tools. From the viewpoint of a non-expert, many of these tools provide the same functionality. Unfortunately, the effort required to learn one tool does not always pay off when using another, very similar tool. Maintaining an automated test suite becomes very difficult when multiple versions of "the same" test harness must exist to support a variety of tools, and there is no single-point-of-truth that can be modified when (for example) an assertion is modified due to a change in system requirements.

Moreover, the lack of a shared language makes cross-tool comparisons inside the research community more technically challenging, and more likely to be invalidated by subtle differences in supposedly "equivalent" harnesses. Such cross-tool comparisons are increasingly important, as a common question from practitioners to the dynamic analysis community is: "There is a multitude of tools, each requiring considerable effort to install, learn, and use. Which one will work best on my program?" Despite our best efforts, the answer we must give is often "We don't know; try several." Use of multiple languages not only makes it harder for practitioners to apply our advice, it makes it harder for the community to develop more sophisticated answers to the question.

Finally, lack of a common language forces every testing tool creator to invent and name these operations, which may result in somewhat ad hoc and confusing results. For example, students learning model checking often fail to understand that `Verify.random` in JPF (Figure 1c) tells the model checker to explore *all* possibilities, rather than working "just like" a random number generator. Abstracting to the notion of choice makes it somewhat clearer that both random generation and true nondeterminism with backtracking are "ways of choosing." Having different syntactic expressions for conceptually similar things in a single language makes appropriate generalization difficult: for different programming languages, it is reasonable for the expression of choice, assumption, and assertion to match the idiomatic forms of the "host language", but having different expressions for the same concept for a single host language (e.g., C in our example) needlessly multiplies entities.

3. OBSTACLES TO THE COMMON LANGUAGE

The chief obstacle to a common language lies in the definition of nondeterministic choice. Implementing `assert` and `assume` is usually non-problematic, though defining a general semantics for `assume` offers several possibilities, none completely satisfactory. For test execution, however, we can simply assume that tools produce inputs that guarantee that `assume` can be omitted without changing the program semantics, since otherwise the tool has violated the semantics of `assume`. During test generation (choice-making), a symbolic execution based method typically uses assumes as additional constraints, and explicit-state methods can provide similar functionality (at a higher price) by defining `assume` in the manner of Java Pathfinder's `ignoreIf` [21], which simply terminates the current trace if the assumption is violated, without reporting a failure[1].

In a language such as C, however, implementing `choose` such that an off-the-shelf C compiler can, using a proper library definition, compile examples such as these seems very difficult. One problem, relatively easy to handle, is that `choose` is essentially polymorphic. While defining different `choose` operations for primitive types is not overly onerous, the C (and Java) type systems are not ideal for composing such basic choice operations to handle complex types such as arrays, structures or objects.

Perhaps more importantly, it is with `choose` that the "identity of problem" between the test case generation problems begins to break down. In symbolic testing or model checking, the range of choice can often simply include all legal values for the type of the choice expression, with `assume` used to enforce preconditions on inputs. In random testing and explicit-state model checking, however, such "wide" nondeterministic choices do not work well. In random testing, the probability of matching two nondeterministic values becomes extremely low, often making tests ineffective (e.g., if such random items are added to containers, they will be almost impossible to select again for finding or removing). In explicit-state model checking, introducing a branching factor based on the number of legal values of even a "small" type such as a `short` makes breadth-first search impossible and turns depth-first search into a sampling algorithm rather than a complete search, even if most choices quickly terminate due to a violated assumption. In the example, we provide a range to `choose` in the SPIN example to show this mismatch. There is no simple, obvious solution to this problem — the non-symbolic methods need more information to restrict explicit search spaces, and the symbolic approaches take full advantage of the flexibility of minimal, assumption-based pruning. One possibility is to always use assumption-style constraints, but define the common language to allow "choice hints" that only apply when using explicit-state methods: `x = choose(1,20)` in C might indicate a choice where model checking and random testing can

[1]Producing accurate coverage results in random testing or model checking with `assume` does require some attention to rollback and commit that we suspect is lacking in most current tools.

```
pathindex = choose(0, NUM_PATHS);
enter_nvfs();
res = nvfs_unlink(path[pathindex]);
nvfs_errno = errno;
leave();
...
enter_ramfs();
now.ramfs_res = ramfs_unlink(path[pathindex]);
ramfs_errno = errno;
leave();
assert(res==ramfs_res);
assert(nvfs_errno==ramfs_errno);
```

(a) SPIN

```
int main() {
  unsigned char a, b;
  unsigned int result=0, i;
  a=choose();
  b=choose();
  for(i=0; i<8; i++)
    if((b>>i)&1)
      result+=(a<<i);
  assert(result==a*b);
}
```

(b) CBMC

```
int main() {
  // The input regular expression.
  char re[SIZE];
  re = choose();
  assume(re[SIZE-1]) == '\0');
  // Try to match against a
  // constant string "hello".
  match(re, "hello");
  return 0;
}
```

(c) KLEE

Figure 2: **Testing using a common extension of C.**

restrict x to between 1 and 20[2]. Since ranges are sometimes rooted in actual program specification (where these are the only valid values) or useful for focusing on simple counterexamples [20], it might be useful to have the option to enable such ranges in symbolic tools as well.

4. RELATED WORK

The problem of a generalized language for test specification is hardly novel. Indeed, the core concepts here date to the earliest work on program correctness, including that of Dijkstra [10] and its extensions [28]. The notion of a common, embedded notation suitable for verification/semantics definition but primarily aimed at *testing* is somewhat obvious, but to our knowledge has not previously been emphasized. In 2008, Groce and Joshi [19] pointed out that random testing [22] and many uses of explict-state model checking [8] could, at the highest level, for sufficiently large state spaces, be seen simply as different exploration strategies. BoogiePL [9] and other languages have been proposed as front ends to theorem provers, essentially supporting a `choose/assert/assume` semantics.

The specific problem of nondeterministic choice is also quite longstanding. From McCarthy's `amb` [26] and Floyd's `choice` [13], to Dijkstra's guarded command language (again) [10] and Milner's work [27], this wheel has been repeatedly invented [31] with slightly different purposes and semantics, often including a notion of program or execution correctness. The gap between these efforts and modern testing and verification tools is the essential point noted by this paper. It is likely that a testing-based approach to this problem will end up sharing results and needs with the continuing body of work on nondeterminism in general, but with a focus on test generation rather than (e.g.) concurrency and true (exhaustive) nondeterminism.

It has been observed that Haskell's QuickCheck [7] is a *domain-specific language* (DSL) for testing. A DSL offers notations and abstractions that are designed to work in a specific application domain [14], here testing. The notion of internal domain specific languages (DSLs) or, as they are often called, domain-specific *embedded* languages (DSELs) [24] is long standing, but to our knowledge has not seriously been investigated in the context of testing and verification, despite the fact that the community has been using what amount to ad-hoc DSELs for quite some time now.

[2]Ideally, the range might be an annotation in a comment, but in C this is impractical since many tools only see a C program after pre-processing has removed comments.

While this paper focuses on the problem of "test programming" and does not consider a language or notation for generalized test input definition, the work of Andrews et al. arguably provides a way to see "test programming" as a specialized case of test generation, by introducing a canonical form for such API tests [4]. As discussed above, an alternative to the approach suggested here ("test programming"), the approach taken by most current tools, is to define the general form of test cases to include API calls and methods. This is essentially what Randoop does [29].

Our thoughts on testing DSELs were inspired by observing the nature of our own use of Adaptation Based Programming (ABP) as a DSEL for testing [15, 16], and seeing the similarities in structure of random and learning-guided test harnesses to those used in explicit-state model checking.

5. CONCLUSION: TOWARDS A COMMON LANGUAGE

The key practical reasons for having a common language to express test programming in a uniform way across different programming languages were already discussed above. In our opinion the two overarching benefits are the following, both of which may increase adoption of our tools by practitioners as a secondary effect:

1. A common testing language facilitates the re-use of testing harnesses.

2. A common testing language provides a platform for research on testing that avoids an unnecessary balkanization of the research community.

What research is required to realize the vision of a common testing language?

We believe that a principled first step is the design of a small *core calculus*, which we call the *Testing Calculus* (*TC*), that encapsulates the generic semantics of the `choose`, `assert`, and `assume` operations. The formalization of this calculus should be, if possible, generic in the semantics of the object language (C, Java, Haskell, ...), which is not quite the case with the classical efforts in nondeterminism. If that turns out to be impossible to achieve, we need to identify a clear "semantic interface" that allows the parameterization of the calculus by (semantic) idiosyncrasies of particular object languages. Ideally, the *TC* would also enable us to define many test generation algorithms cleanly in a language-agnostic form.

An important aspect of the design of the *TC* is the properties of its operations, specifically the interaction with the

semantics of the object language. To this end, required and desired properties of `choose`, `assert`, and `assume` should be expressed in axioms, and these axioms used to guide the definition of the *TC* semantics. This semantics-directed approach to language design [11] has a number of advantages [12]. Specifically, it leads to more general language designs, which is of particular importance in the context of designing a common testing language for different object languages.

As a second step, after the *TC* semantics have been established, a concrete syntax has to be developed that can be customized to accommodate syntactic requirements of the different object languages as well as the semantic adjustments required by the potentially needed semantic interface.

The formalization of a common testing language through a testing calculus will help the research community to identify, and focus on, important questions. It will help to share and reuse results, and potentially enable cross-tool comparison or cooperation in verification efforts when actualized in real-world languages. The generic testing calculus will not only be a DSL for expressing test harnesses, it will also be DSL for the research community to exchange ideas and advance the field.

6. ACKNOWLEDGEMENTS

The authors would like to thank Jamie Andrews, John Regehr, Eric Eide, Chaoqiang Zhang, Amin Alipour, and Rajeev Joshi for helpful discussions related to the contents of this paper. A portion of this research was funded by NSF grant CCF-1054786.

7. REFERENCES

[1] Introduction to QuickCheck. http://www.haskell.org/haskellwiki/Introduction_to_QuickCheck.

[2] JPF: the swiss army knife of Java(TM) verification. http://babelfish.arc.nasa.gov/trac/jpf.

[3] KLEE - tutorial two. http://klee.llvm.org/Tutorial-2.html.

[4] Jamie Andrews, Yihao Ross Zhang, and Alex Groce. Comparing automated unit testing strategies. Technical Report 736, Department of Computer Science, University of Western Ontario, 2010.

[5] Tim Bauer, Martin Erwig, Alan Fern, and Jervis Pinto. Adaptation-based programming in Java. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 81–90, 2011.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating System Design and Implementation*, pages 209–224, 2008.

[7] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

[8] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[9] Robert DeLine and K. Rustan M. Leino. Boogiepl: a typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[10] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[11] M. Erwig and E. Walkingshaw. Semantics First! Rethinking the Language Design Process. In *Int. Conf. on Software Language Engineering*, LNCS 6940, pages 243–262, 2011.

[12] M. Erwig and E. Walkinhgshaw. Semantics-Driven DSL Design. In M. Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. IGI Global, 2012. To appear.

[13] Robert W. Floyd. Nondeterministic algorithms. *J. ACM*, 14(4):636–644, 1967.

[14] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[15] Alex Groce. Coverage rewarded: Test input generation via adaptation-based programming. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 380–383, 2011.

[16] Alex Groce, Alan Fern, Martin Erwig, Jervis Pinto, Tim Bauer, and Amin Alipour. Learning-based test programming for programmers. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, 2012. To appear.

[17] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.

[18] Alex Groce, Gerard Holzmann, Rajeev Joshi, and Ru-Gang Xu. Putting flight software through the paces with testing, model checking, and constraint-solving. In *International Workshop on Constraints in Formal Verification*, pages 1–15, 2008.

[19] Alex Groce and Rajeev Joshi. Random testing and model checking: Building a common framework for nondeterministic exploration. In *Workshop on Dynamic Analysis*, pages 22–28, 2008.

[20] Alex Groce and Daniel Kroening. Making the most of BMC counterexamples. In *Workshop on Bounded Model Checking*, pages 67–81, 2004.

[21] Alex Groce and Willem Visser. Heuristics for model checking Java programs. *Software Tools for Technology Transfer*, 6(4):260–276, 2004.

[22] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

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

[24] P. Hudak. Building Domain-Specific Embedded Languages. *ACM Computing Surveys*, 28(4es):196–196, 1996.

[25] Daniel Kroening, Edmund M. Clarke, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[26] John McCarthy. A basis for a mathematical theory of computation, preliminary report. In *Papers presetnted at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, pages 225–238, 1961.

[27] Robin Milner. *A Calculus of Communicating Systems*. 1980.

[28] Greg Nelson. A generalization of dijkstra's calculus. *TOPLAS*, 11(4):517–561, Oct. 1989.

[29] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.

[30] Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, and Darko Marinov. Testing container classes: Random or systematic? In *FASE*, pages 262–277, 2011.

[31] Harald Søndergaard and Peter Sestoft. Non-determinism in functional languages. *Comput. J.*, 35(5):514–523, October 1992.

[32] Willem Visser, Corina Păsăreanu, and Radek Pelanek. Test input generation for Java containers using state matching. In *ISSTA*, pages 37–48, 2006.

[33] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation*, pages 283–294, 2011.