Coverage Rewarded: Test Input Generation via Adaptation-Based Programming

Alex Groce
School of Electrical Engineering and Computer Science
Oregon State University, Corvallis, OR
agroce@gmail.com

***Abstract*—This paper introduces a new approach to test input generation, based on reinforcement learning via easy to use adaptation-based programming. In this approach, a test harness can be written with little more effort than is involved in naïve random testing. The harness will simply map choices made by the adaptation-based programming (ABP) library, rather than pseudo-random numbers, into operations and parameters. Realistic experimental evaluation over three important fine-grained coverage measures (path, shape, and predicate coverage) shows that ABP-based testing is typically competitive with, and sometimes superior to, other effective methods for testing container classes, including random testing and shape-based abstraction.**

***Keywords*-software testing; reinforcement learning**

I. Introduction

The majority of modern software systems rely on at least one module that (1) presents a well-defined set of API calls or events that (2) modify a (potentially complex) state. Container and string libraries, database packages, GUI back-ends, and other examples appear in a wide range of programs. Such components are often both critical and more reused than other software artifacts, increasing the payoff for effective testing, which is currently the most effective method for verifying rich properties, such as functional correctness, of these components. Generation of quality tests for an API automatically has been widely studied [1], [2], [3]. Tools for generating test input sequences are now widely available, e.g. the SPIN model checker, which supports model checking of C code [4], the Java model checker JPF [5], and the concolic testing tool CREST [6]. Unfortunately, use of these tools requires programmers to learn a new approach to testing, sometimes including a new language for describing tests, and often requires the installation of numerous untested software packages, including theorem provers, unfamiliar programming languages, etc.; model checking and concolic testing also sometimes fail to scale to complex code. These tools are therefore not yet widely adopted, even when they are highly appropriate to a testing task. At present, programmers seem to prefer to code their own tests in the same language as the program tested rather than use tools that take a program and produce tests, or that require special compilation or execution environments.

Random testing, on the other hand, can require no investment on the part of a programmer beyond the generalization of unit tests to sequences of random method calls with random parameters. Moreover, recent evaluations of methods for testing container classes have shown that random testing can often be as effective as approaches based on model checking [1], [2], [7]. Unfortunately, random testing does not work well even for all container classes [2]. It would be of benefit to programmers to have access to other approaches that strike a similar *trade-off between effectiveness and ease-of-use* as random testing but work well in cases where random testing does not do well.

Adaptation-based programming (ABP) [8] allows a programmer to exploit reinforcement learning [9] to "implement" difficult algorithms. Rather than writing a function to compute a value, the programmer simply asks the ABP-library to "suggest" a value, given a context (the context is information on the current state of the system). The programmer then signals a reward to the ABP library based on the results of taking this "advice." The ABP-library uses a reinforcement learning (RL) algorithm to optimize expected reward. RL is an approach to the problem of learning controllers that maximize expected reward in controllable stochastic transition systems. E.g., to program tic-tac-toe in ABP, a programmer would allow the library to suggest a move based on the board state, and provide a reward if the moves resulted in a win. Each game would constitute one "episode" of learning, since moves in previous games have no influence on the reward for future games. Initially, behavior of the ABP-based player would be essentially random. Over time, however, the adaptive process (using the RL algorithm) should improve its behavior.

The key insight of this paper is that a programmer can take an ABP approach to generating tests for a program with a clear API. She lets the ABP library select methods to call and parameters for the selected method calls for the program being tested (called the Software Under Test, or SUT). In practice, the programmer essentially writes a random testing harness, replacing calls to a pseudo-random number generator with calls to the `suggest` method, using, e.g., a string representation of the SUT's current state as a context. Each test sequence (from container initialization until we begin a new test on a new container) constitutes an episode. Figure 1 shows an example ABP test harness. Notice that the harness is just a standard Java program, making calls to an ABP library.

What can our programmer reasonably use as a reward, in order to "encourage" an adaptive process to thoroughly test

```
AdaptiveProcess test=AdaptiveProcess.init();
Adaptive<String,TestOp>opChoice=test.initAdaptive(String.class,TestOp.class);
Adaptive<String,TestVal>valChoice=test.initAdaptive(String.class,TestVal.class);
for (int i = 0; i < NUM_ITERATIONS; i++) {
  // Empty test and reg objects
  SUT = new SplayTree(); Oracle = new BinarySearchTree();
  // The state is simply a linearization of the SplayTree
  String context = SUT.toString();
  for (int j = 0; j < M; j++) {
    // Used just like pseudo-random number generator
    // AllVals fields contain a set of all values of the type
    TestOp o = opChoice.suggest(context, TestOp.AllVals);
    TestVal v = valChoice.suggest(context, TestVal.AllVals).ordinal();
    switch (o) {
      case INSERT: r1 = SUT.insert(v);  r2 = Oracle.insert(v); break;
      case REMOVE: r1 = SUT.remove(v);  r2 = Oracle.remove(v); break;
      case FIND:   r1 = SUT.find(v);    r2 = Oracle.find(v); break;
    }
    assert ((r1 == null && r2 == null) || r1.equals(r2));
    context = SUT.toString(); // Update the context
    if (!states.contains(context)) { // Is this a new state?
      states.add(context); test.reward(1000);
    }
  }
  test.endEpisode();
```

Fig. 1. ABP Test Harness for SplayTree Using BinarySearchTree as Oracle

code? The SplayTree example provides an instance of the general answer. After each test step, the harness checks to see if the current SUT state has been observed before. If not, it adds it to the set of visited states and *rewards the ABP library for exposing new behavior of the SUT*. In other words, the programmer can provide *rewards based on increases in test coverage*. It is easy to augment instrumentation to not only record coverage, but to signal an appropriate reward for new coverage. This gives the ABP's adaptive process a goal that the programmer can hope will correlate with effective testing, with no overhead beyond that required in computing coverage. Initially, choices will be random, and ABP-based testing will be random testing. However, after the adaptive process has learned a policy, the choices will usually be chosen to optimize expected reward; the remainder of the time the process will provide a random value. This alternation of "optimal" choices and random choices ensures that testing can *improve* over time but that *exploration* is never completely abandoned. Note that the adaptive process will *only* receive a reward for its first exploration of a new coverage element, whether that element is a statement, a branch, a shape, a path, or a predicate valuation. Experimental results indicate that this unusual reward structure does not prevent RL from learning a policy that, over time, improves test suite coverage. Informally, we can think of this as playing a game against an opponent who never "falls for" the same trick twice.

II. Related Work

The problem of generating test input sequences has recently considered generation of tests for container classes, with random testing, shape-abstraction based "model checking" and symbolic execution emerging as the most promising methods [1], [2]. The only previous work on using reinforcement learning in software testing, to our knowledge, is that of Veanes et al. [10], which considered only model-based online testing of reactive systems and a reward based on an *ad hoc* planning-type problem. The general idea of "learning" either tests or specifications is also relatively widely studied. E.g., Andrews et al. [3] used genetic algorithms in the Nighthawk tool. ABP-based testing is similar in that both approaches learn *how to construct test cases* rather than learning an ideal set of test cases; however, ABP-based programming learns what method to call and what input to provide based on a context, while Nighthawk aims at tuning probabilities for better random test generation.

III. Experimental Results

Most of the SUTs included in the experimental results are taken from the previous literature on test input generation; in particular 13 subjects are taken from the work of Sharma et al. [2] which combines subjects from several other studies. Two additional popular container classes (a splay tree and a chaining hash table) were added for this paper, both from standard textbook implementations. A test case largely follows the form: `SUT = new Container(); SUT.`$m_1(i_1)$`; ...;`

`SUT.`$m_M(i_M)$`;`, where $\forall n : 0 \leq i_n < N$. In some cases (e.g., heaps), methods require more than one input parameter, or use of a vector to track nodes, but a test case is still a sequence of method calls on the SUT. M (test case length) and N (input range) in all experiments are set at 200 and 20, a "good" value tuned by experiments with random testing. The coverage of a test suite is the *union of coverage for all test cases*.

Random testing been recognized as an effective method for testing API-based programs such as container classes [7], [11]. Here, a random test consists of M randomly selected method calls with inputs chosen randomly from N integers. Exploration based on shape abstraction is performed as in the exposition of Sharma et al. [2].

The framework for ABP-based testing is almost identical to that used for random testing. The random selection of methods and input parameters is replaced with the choice method of an adaptive process, with the additional parameter of a *context*. The adaptive process is rewarded each time a test results in coverage of a new branch, statement, path, or shape. In order to demonstrate that effective coverage does not depend on explicit rewards, the framework does not reward predicate coverage. Experiments were performed with a variety of obvious contexts, ranging from fully concrete to the shape abstractions used in previous model checking efforts, in some cases augmented with basic coverage information. The results reported below are based on a configuration pairing a shape abstraction and a *count* of the current test case's branch and statement coverage. While this configuration did not always perform best, it was consistently effective and provides a fair baseline for comparison, with no tuning of ABP to the particular SUT.

The test suites generated by each method are evaluated in terms of coverage metrics. All SUTs are automatically instrumented for branch and statement coverage by CodeCover (*http://codecover.org*). Since methods obtained the same (excellent) branch and statement coverage for almost all SUTs, evaluation is based on three much more difficult-to-obtain coverages: path, shape, and predicate coverage. Shape coverage, to our knowledge not used in previous evaluations of testing approaches, simply applies the underlying rationale of shape abstraction as a coverage metric: it is desirable to cover many different shapes of a container, ignoring the contents of the structure. This paper adopts the same predicate coverage code as used in other studies [2]. Metrics are likely correlated, but are independent, with no subsumption relationships. Rather than base an evaluation on the effectiveness of fixed-size test suites produced by each method, we allow each method to test each subject for a fixed amount of time, on the same hardware, using a common framework. Evaluation is based on coverage obtained by each testing method for three time budgets. A budget of only 30 seconds represents a reasonable "quick check" for errors, e.g. after every compile. Budgets of 30 minutes and 1 hour show how much improvement in coverage can result from more in-depth testing. A 6GB heap was used in all experiments. Experiments were duplicated 5 times, with different seeds, showing *no overlap in rankings with different seeds*: e.g, if ABP-based testing performed better than random testing, it performed better for all seeds.

Basing experimental results on real-time poses an obvious danger: if one method is implemented more efficiently than others, it will have a major advantage. Because it is so easy to implement, this may produce results that favor random testing. In order to adjust for the additional overhead of replay in shape abstraction, all such experiments were allowed time equal to twice the limit. This rough estimate of costs is based on a comparison with exhaustive testing (using the same replay mechanism) and random testing. No adjustment was applied for the inefficiency of the current ABP library, in order to show that, even with a primitive prototype using slow Java strings to represent contexts, ABP-based testing is competitive with or superior to random testing and shape abstraction for interesting container classes.

Table I shows the results of testing a large number of Java containers, summarizing over 12 straight days of computation. In the table, ✓ indicates that a particular method obtained the highest coverage obtained by any method for that SUT and amount of testing time. In some cases, more than one method tied for best coverage; ✓✓ indicates *unique best* coverage. × indicates *unique worst* coverage. For all metrics other than predicates, maximum coverage increased considerably with more time spent testing. The final three columns of the results show the actual maximum coverage value obtained by any method, and the difference (Δ) between best and second-best coverage value. Due to space budgets, we have omitted the 30 second results except in cases where random testing did not perform best, and omitted results for LinkedList and NodeCachingLinkedList as the relative performance of methods was so similar to that shown in SinglyLinkedList.

The density of ✓and ✓✓symbols conveys a simple summary of the results: random testing performed well on a wide array of test subjects, ABP-based testing performed second best, and shape abstraction performed least well for most SUTs. Each method performed best on at least one coverage metric, for at least eight SUT/time combinations. ABP-based testing performed especially well on BinomialHeap, FibHeap, and HeapArray, suggesting its strength lies in testing SUTs requiring complex input sequences, a weakness of random testing observed in previous work [1], [2]. However, random testing was the *best* method for testing FibonacciHeap. Moreover, ABP-based testing also generally did better than random testing for linked lists, the *simplest* of the containers considered. In general, the experimental data does not support a claim for dominance for either random testing or ABP-based testing. Both methods are competitive, and results vary widely and unpredictably with SUT. It is best to use both methods if there is sufficient test budget. For very small budgets it may be best to only use random testing, as it generally performed best for the 30 second test budget. Shape abstraction appears to be less effective overall, but for some SUTs it was the most effective method for obtaining path coverage. This experimental data clearly supports ABP-based testing, even with an inefficient library implementation and no

TABLE I
COVERAGE RESULTS

SUT + Time	ABP			Random			Shape Abstraction			Max Coverage(Δ vs. 2nd Best)		
	PA	SH	PR	PA	SH	PR	PA	SH	PR	PA	SH	PR
AvlTree 30m			×	✓✓	✓✓	✓	×	×	✓	428(96)	16925(12005)	104(1)
AvlTree 1h			✓	✓✓	✓✓	✓	×	×	✓	442(99)	22668(15320)	104(0)
BinomialHeap 30s		✓✓		✓✓		✓✓	×	×	×	233(54)	27(11)	304(8)
BinomialHeap 30m	✓✓	✓✓	✓✓		×	×	×			1378(523)	39(12)	327(12)
BinomialHeap 1h	✓✓	✓✓	✓	×	×	×			✓	1735(620)	38(3)	327(11)
BinTree 30m			✓	✓✓	✓✓	✓	×	×	✓	4487(3094)	122921(90605)	157(0)
BinTree 1h			✓	✓✓	✓✓	✓	×	×	✓	5657(3105)	224828(143751)	157(0)
ChainedHashTable 30m	✓✓		✓		✓✓	✓	×	×	✓	342(299)	386689(301362)	6(0)
ChainedHashTable 1h	✓✓		✓		✓✓	✓	×	×	✓	323(281)	787968(344720)	6(0)
FibHeap 30s	×	✓✓		✓✓		✓✓		×	×	620(324)	629(526)	115(6)
FibHeap 30m	✓✓	✓✓	✓		×	✓	×		×	24584(16506)	2519(1933)	118(14)
FibHeap 1h	✓✓	✓✓	✓	×	×	✓			×	51608(37025)	3571(2637)	118(14)
FibonacciHeap 30m				✓✓	✓✓	✓✓	×	×	×	18522(14659)	5047(1343)	506(266)
FibonacciHeap 1h				✓✓	✓✓	✓✓	×	×	×	32099(24586)	8723(1614)	538(265)
HeapArray 30s	✓✓	✓✓				✓✓	×	×	×	283(33)	344(256)	69(9)
HeapArray 30m	✓✓	✓✓	✓		×	✓	×		×	3107(2486)	7991(6815)	71(6)
HeapArray 1h	✓✓	✓✓	✓		×	✓	×		×	4532(3799)	13097(11921)	71(2)
IntAvlTreeMap 30m			✓	✓✓	✓✓	✓	×	×	×	624(111)	3113(526)	225(10)
IntAvlTreeMap 1h			✓	✓✓	✓✓	✓	×	×	×	638(51)	3937(326)	225(0)
IntRedBlackTree 30m				✓✓	✓✓	✓✓	×	×	×	708(173)	6943(2814)	379(4)
IntRedBlackTree 1h				✓✓	✓✓	✓✓	×	×	×	755(139)	8713(2830)	379(2)
SinglyLinkedList 30s	×	✓✓	×	✓✓				×	✓✓	699(81)	189(155)	67(4)
SinglyLinkedList 30m		✓✓	✓	×	×	✓	✓✓		✓	3875(2240)	187(83)	67(0)
SinglyLinkedList 1h		✓✓	×	×	×	✓	✓✓		✓	4999(2337)	192(61)	67(1)
SplayTree 30m			✓	✓✓	✓✓	✓	×	×	×	2408(957)	50870(22885)	262(3)
SplayTree 1h			✓	✓✓	✓✓	✓	×	×	×	2969(1109)	79223(29803)	262(3)
TreeMap 30m			×	✓✓	✓✓	✓✓	×	×		671(114)	5154(1006)	358(2)
TreeMap 1h				✓✓	✓✓	✓✓	×	×		697(82)	6637(872)	358(2)
TreeSet 30m			×	✓✓	✓✓	✓	×	×	✓	707(105)	6668(1429)	334(2)
TreeSet 1h			✓	✓✓	✓✓	✓	×	×	✓	805(112)	8666(20)	334(0)

customization in rewards or context, as *competitive with the container testing methods previously considered most effective*. Given that the implementation effort for ABP-based testing is little more than that required for random testing this makes a very compelling case for adding ABP-based testing to the arsenal of effective test input generation methods.

IV. FUTURE WORK

This paper presents only a very preliminary investigation of this approach; on the empirical side, obvious next efforts are application to more complex systems (e.g. file systems) that are already tested using random testing, comparisons of mutation testing results, and longer run-times. More fundamentally, an investigation of why the ABP based approach succeeds in many cases, and how, without requiring programmers to have RL expertise, RL algorithms can be better exploited (or modified) is critical to making full use of this new technique.

ACKNOWLEDGEMENTS

The author would like to thank Martin Erwig, Alan Fern, Jervis Pinto, Tim Bauer, Darko Marinov, Milos Gligoric, Willem Visser, Chaoqiang Zhang, Shalini Shamasunder, Amin Alipour, and Jamie Andrews.

REFERENCES

[1] W. Visser, C. Păsăreanu, and R. Pelanek, "Test input generation for Java containers using state matching," in *International Symposium on Software Testing and Analysis*, 2006, pp. 37–48.

[2] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov, "Testing container classes: Random or systematic?" in *Fundamental Approaches to Software Engineering*, 2011, to appear.

[3] J. Andrews, F. Li, and T. Menzies, "Nighthawk: A two-level genetic-random unit test data generator," in *Automated Software Engineering*, 2007, pp. 144–153.

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

[5] "JPF," http://babelfish.arc.nasa.gov/trac/jpf.

[6] "CREST," http://code.coogle.com/p/crest.

[7] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*, 2007, pp. 75–84.

[8] T. Bauer, M. Erwig, A. Fern, and J. Pinto, "Adaptation-based programming in Java," in *Workshop on Partial Evaluation and Program Manipulation*, 2011, pp. 81–90.

[9] R. Sutton and A. Barto, *Reinforcement Learning: an Introduction*. MIT Press, 1998.

[10] M. Veanes, P. Roy, and C. Campbell, "Online testing with reinforcement learning," in *Formal Approaches to Software Testing and Runtime Verification*, 2006, pp. 240–253.

[11] A. Arcuri, M. Z. Z. Iqbal, and L. C. Briand, "Formal analysis of the effectiveness and predictability of random testing," in *International Symposium on Software Testing and Analysis*, 2010, pp. 219–230.