Lightweight Automated Testing with Adaptation-Based Programming

Alex Groce, Alan Fern, Jervis Pinto, Tim Bauer, Amin Alipour, Martin Erwig, Camden Lopez
School of Electrical Engineering and Computer Science
Oregon State University, Corvallis, OR
Email: alex,afern,pinto,bauertim,alipourm,erwig@eecs.oregonstate.edu,camden.lopez@gmail.com

***Abstract*—This paper considers the problem of testing a container class or other modestly-complex API-based software system. Past experimental evaluations have shown that for many such modules, random testing and shape abstraction based model checking are effective. These approaches have proven attractive due to a combination of minimal requirements for tool/language support, extremely high usability, and low overhead. These "lightweight" methods are therefore available for almost any programming language or environment, in contrast to model checkers and concolic testers. Unfortunately, for the cases where random testing and shape abstraction perform poorly, there have been few alternatives available with such wide applicability. This paper presents a generalizable approach based on reinforcement learning (RL), using adaptation-based programming (ABP) as an interface to make RL-based testing (almost) as easy to apply and adaptable to new languages and environments as random testing. We show how learned tests differ from random ones, and propose a model for why RL works in this unusual (by RL standards) setting, in the context of a detailed large-scale experimental evaluation of lightweight automated testing methods.**

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

I. Introduction

The problem of automatically generating tests for software systems is long-studied and generally accepted as critically important to the future of reliable systems. Manual testing is expensive and, alas, often ineffective for discovering subtle flaws in systems. While automatic generation of tests for large, complex systems remains difficult, highly automated methods for generating unit tests for small, critical modules have recently gained considerable traction [1], [2], [3]. As early as 2000, Haskell programmers began to incorporate lightweight highly automated random testing as a frequently used method for "quickly checking" a function's correctness, using the QuickCheck library [4].

Methods for automated unit testing can be categorized in many ways, but perhaps the most important from the point of view of practitioners, rather than researchers, is how easy it is to incorporate modern automated methods into their software development and QA practices. Concolic testing [5] and model checking are powerful in principle, but often impractical to apply. Model checking involves capturing state for backtracking [6] or automated abstraction, and concolic testing requires symbolic execution. The most mature model checkers even for popular languages such as C and Java [6], [7] are often difficult for experts to use on real-world code, and non-expert users often cannot overcome the challenges inherent in e.g., testing code that involves calls to native code in JPF, dynamic memory allocation in SPIN, or abstraction/search problems in any model checker [8].

A second problem with concolic testing and dedicated explicit-state model checking is that a large amount of research and development is required to build an effective, efficient symbolic execution engine or backtracking model checker. Even for Java, concolic engines tend to be semi-stable research projects hard for non-experts to find and install. Non-prototype dedicated model checkers or concolic testers do not appear to be available for such languages as Python, Perl, Ruby, Scala, OCaml, and Haskell. Of course, in the long run, popular languages will probably be served by popular methods; however, (1) code needs to be tested today, not in the long run and (2) the rate at which new languages are being introduced (and becoming popular) does not seem to be decreasing. Consider the case of a programmer in Python or OCaml, developing a new implementation of binomial heaps with custom modifications for her application. If she wants to use automated test generation to save time and find more bugs, what are her options?

A. The Value of Lightweight Methods

Fortunately, two useful automated testing methods are available for almost *any* language: random testing (RT) [9] and non-backtracking model checking with shape abstraction (SA) [3]. These methods require no support in the form of special or symbolic execution environments. In fact, most experienced programmers can code up a basic RT harness or SA "model checker" with ease in almost any language. We call such testing methods *lightweight automated testing methods*. Such methods must meet three basic requirements: (1) they must be easy enough to *implement* that they are essentially available for *all* programming languages and environments; (2) they must be easy enough to *use* that programmers can quickly code up test harnesses for small, moderate-complexity modules; and, (3) they must be fast enough that tests can be quickly generated and examined, to determine if testing is producing useful results. Experimental results show that even simple versions of RT and SA can be effective for many subject programs [2]. Unfortunately, "many" does not equal "all,": RT and SA perform poorly for some subjects, and the set of alternative lightweight methods is thus far almost empty. This paper shows that testing based on adaptation-based programming [10] (ABP) using reinforcement learning (RL) meets lightweightness requirements, and evaluates ABP's effectiveness experimentally. The contributions of this paper are: (1) an in-depth examination of ABP-based testing, showing how it satisfies the requirements for a lightweight method, (2) an experimental evaluation of lightweight methods using realistic fixed-duration testing, over more containers than previous studies, using three hard-to-complete coverage measures, (3) a case study applying ABP to a larger, non-container class example, and (4) examination of the statistical structure of ABP and random tests as a lead-in to (5) a consideration of *why* ABP testing is highly effective in many cases, despite our approach not satisfying the assumptions made by standard RL algorithms. We provide practical advice for the software developer aiming to use lightweight methods to improve module reliability at low cost, in any programming language.

II. Background: Adaptation-Based Programming

Adaptation-based programming (ABP) [11] is a an approach to programming that allows a programmer to exploit machine learning to "implement" difficult optimization algorithms. Rather than writing a function to compute a value, the programmer asks the ABP-library to "suggest" a value, given a context (a formulation of the state of the system/problem). The programmer rewards the ABP library based on how good the suggestion was. ABP acts as a friendly interface to a reinforcement learning (RL) [12] algorithm that attempts to optimize expected reward. ABP lets the programmer concentrate on problem definition, not the underlying machine learning algorithm.

RL is an approach to the problem of learning controllers that maximize expected reward in controllable stochastic transition systems. Informally, such a system can be imagined as a graph of control points with rewards possibly observed on transitions. Each control node has an associated set of actions that influence (perhaps only probabilistically) the transition taken. An optimum controller for such a system is one that selects actions at all control points such that total reward is maximized. Program-like structures annotated with control points are isomorphic to Semi-Markov Decision Processes (SMDPs), widely used models of controllable stochastic systems [13], [14]. The details of SMDP theory are not essential to understand ABP: what is important is that there are well-known RL algorithms for learning policies (action choices based on a context indicating the control point) for SMDPs based on repeated interactions.

As an example, to program tic-tac-toe in ABP, a programmer would allow the ABP library to suggest a move (e.g. a number 1-9 indicating a board position) based on the current board state (perhaps a simple string, e.g. `'X-XO-OO-X'`), and provide a positive reward if the moves suggested eventually resulted in a win (illegal moves might be negatively rewarded). 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 (the library's encapsulation of all it has learned about the problem using the RL algorithm) should improve its behavior. The programmer need not even be aware of the concept of SMDPs underlying this adaptation to the reward function.

III. ABP-Based Testing

The key insight of ABP-based testing is that a programmer can take a similar approach to generating tests. Rather than selecting moves in a game, she lets the ABP library select methods to call and parameters for 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 the RNG with calls to the ABP library's `suggest` method, using, e.g., a string representation (via `toString`) of the SUT's current state as a context. Each test sequence from initialization to end constitutes an episode. Figure 1 shows an example ABP test harness for a `SplayTree` class, using a binary search tree (a simpler to implement library with equivalent functionality) as an oracle. In general, ABP-based testing supports all the oracle methods that might be used in RT or SA. Notice that the ABP-based testing harness is just a standard Java program, making calls to a library implemented in Java. No special compilation or execution environment is involved; the ABP library's interface is only slightly more complex than that of a typical pseudo-random number generator. The use of methods with a single integer parameter is simply an accident of the example; an Adaptive (action variable) can be based on any finite type (though, as in RT or SA, we might expect poor results when the domain is too large). The key question is: what can the programmer use as a reward, in order to "encourage" the adaptive process to thoroughly test the SplayTree code?

The example provides a concrete clue to the general answer. After each test step, the harness checks to see if the current SUT state has been previously seen during testing. 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 coverage instrumentation to not only record statement/branch/path coverage, but to signal an appropriate reward for new coverage. This gives the ABP's adaptive process an optimization goal that the programmer can hope will correlate with effective testing, with little additional complexity over that required in computing coverage in the first place. Initially, in the absence of experience, ABP

```
import abp.*;
...
  AdaptiveProcess test = AdaptiveProcess.init();
  HashSet<String> states = new HashSet<String>(); // Store all states visited
  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++) {
    SUT = new SplayTree(); // Create an empty container at beginning of each test case
    Oracle = new BinarySearchTree(); // Empty oracle container
    String context = SUT.toString(); // The state is simply a linearization of the SplayTree
    for (int j = 0; j < M; j++) {
      TestOp o = opChoice.suggest(context, TestOp.AllVals); // Used just like pseudo-random number generator
      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)); // Behavior should match
      context = SUT.toString(); // Update the context
      if (!states.contains(context)) { // Is this a new state?
        states.add(context); test.reward(1000); } // Good work, AdaptiveProcess test, you found a new state!
    }
    test.endEpisode();
```

Figure 1. Adaptation-Based Programming in a Test Harness

A simple test harness for a SplayTree class, rewarding test operations that produce previously unseen states for the class. This code assumes `toString` produces a simple linearization of the SplayTree contents (e.g., `(0,(1,3,4),2)`). The `AllVals` fields of the enum classes `TestOp` and `TestVal` are sets with all values of the enum types.

chooses randomly, effectively duplicating RT. However, after the adaptive process has observed a few rewards, the learned policy will, with high probability (about 90% of the time), take the actions with maximum predicted reward, and only choose randomly 10% of the time. This alternation between exploiting what has been learned and exploring with random actions arises from the RL algorithm.

The ABP library used in this paper (http://groups.engr.oregonstate.edu/abp), makes use of a popular reinforcement learning algorithm called SARSA(λ) [12]. At the heart of the algorithm is the notion of a Q value defined as follows: at adaptive A, the Q value of context c and action a ($Q_A(c, a)$) is the expected sum of rewards seen by executing a in c and following the optimal policy thereafter. Learning these Q values allows us to pick actions optimally since the best action is simply the one with the largest Q value. The SARSA(λ) algorithm learns these Q values from experience. This is done by executing the learning algorithm for a number of episodes during which it updates the Q values at every (context,action) pair that is encountered. The algorithm follows an ϵ-greedy explore-exploit policy which means that the best action is chosen (i.e. exploit) with probability $(1-\epsilon)$ while an action is chosen randomly (i.e. explore) with the remaining ϵ probability. The library uses a small (typical) value of 0.1 for ϵ. Finally, the value of λ ($\in [0, 1]$) controls the extent to which a particular action is given credit for future rewards. A large value of λ updates an action's Q value with rewards that occur long after the action is taken whereas a small value of λ only updates the Q value with rewards seen immediately after the action is taken. The ABP library sets λ to the moderately high value of 0.75, allowing test coverage that only results from a complex combination of operations to be effectively taken into account.

Note that in some sense our coverage-based approach to rewards is "abusing" the basis of RL: the objective function is changing with each episode, in that the probabilities of reward for certain actions in certain states is decreasing with time. 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 the ABP library from learning a policy that, over time, improves test suite coverage. In Section VI we consider a possible explanation for the success of ABP-based testing that takes this non-stationary reward structure[1] into account.

A. ABP as Lightweight Automated Testing

ABP satisfies our requirements for a lightweight automated test generation method. SARSA(λ) is a fairly easy algorithm to implement in an afternoon in almost any language — e.g., Sutton's well known C++ and LISP implementations (http://webdocs.cs.ualberta.ca/~sutton/tiles.html) are each less than 500 lines, including quite numerous comments. Crucially, RL requires no program analysis for the SUT's language. ABP is also roughly as easy to use as RT, with the same few pitfalls (range of random values), with the one addition that a suitable context must be devised. We believe that for the kind of container-like modules where lightweight methods are most effective, a string linearization is likely to prove effective and is extremely easy to code. We show in our experimental results that ABP can generate

[1]A stationary reward is one where the probability distribution over the (typically random) reward seen due to performing the same action in the same state is time-invariant.

useful test suites in as little as 30 seconds, providing immediate feedback to a user. ABP is therefore lightweight; is it also effective?

IV. Experimental Methodology

A. Subjects and Test Cases

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 (one from Cormen, Leiserson, Rivest and Stein's second edition, and one from Weiss' *Data Structures and Algorithms Analysis in Java*.

For all methods, a test case largely follows a simple 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), some methods require two input parameters, or the maintenance of a node vector from which an input is selected (and deleted nodes are removed), but in these cases a test case is still essentially simply a sequence of method calls on the same container object. The parameters M (the length of a test case) and N (the range of input parameters) in all experiments are set at 200 and 20, respectively. These values derive from those used in previous experiments with some of these SUTs [3], [2], tuned for this paper to a "good" value experimentally. For most container classes, 20 different input values are much more than sufficient to expose all realistic faults, with the exception of overflows, which are most effectively handled by dedicated unit tests. In all cases, the coverage of a test suite is the total coverage of all test cases executed, rather than the maximum achieved by one test or set of tests.

B. Testing Methods

Random testing (RT) has recently been recognized as an effective and easy to use method for testing API-based SUTs such as container classes [1], [15]. In this paper RT simply means selecting methods and parameters using a pseudo-random number generator.

The framework for ABP-based testing (ABP) is almost identical to that used for RT. Random selection of methods/parameters is replaced with a call to the `suggest` method of an adaptive process, with the additional parameter of a *context*. The adaptive process is rewarded for coverage of a new branch, statement, path, or shape[2]. Experiments were performed with a variety of contexts, defined by two aspects: the representation of the state, and the representation of the currently executing test case's coverage. The container state was in some cases represented as fully concrete (the result of `toString`), in some cases as a shape abstraction, and in other cases as a shape abstraction annotated with information on contained values (which might or might not define actual positions, depending on the container). Coverage options included no coverage information, counts of coverage (# of statements/branches covered), or full coverage information as which statements and branches the current test case (not suite) had executed. These simple contexts were not developed using any special knowledge of ABP, RL, or SUTs. They are simply standard representations familiar to the typical programmer, or standard, very easily implemented (by a string crawl on `toString` results) abstraction approaches. The results reported below are, except when noted, based on the configuration pairing a shape abstraction without membership information and a count of the current test case's branch and statement coverage, which we call ShapeCover. While this configuration did not always perform best of the ABP approaches, it was consistently effective, very often best, and provides a fair comparison with other approaches.

Visser et al. introduced an explicit-state model checking approach to test sequence generation based on shape abstraction (SA) [3], which Sharma et al. later exposited in a form that did not depend on a model checking framework [2]. Essentially, this approach performs a *BFS over test sequences*, up to length M, with parameter range N, pruning the search tree based on the equivalence of the shape of the SUT to any previously explored shape: only children where the SUT's shape has not been previously visited will be expanded in each generation of the search. In practice, the M was irrelevant in these experiments, as no exploration managed to reach depth 200 of the search tree.

C. Evaluation Methods

The test suites generated by each method are evaluated in terms of coverage metric. First, all SUTs are automatically instrumented for branch and statement coverage by CodeCover, an open source tool. All methods obtained the same (complete) branch and statement coverage for these SUTs. Evaluation is therefore based on three much more difficult-to-obtain coverages: path, shape, and predicate-complete test coverage (PCT). Complete coverage of all paths and shapes is often computationally intractable for even simple SUTs with relatively small bounds; complete PCT coverage is often more tractable, but still much more difficult than complete branch or statement coverage. All coverage metrics are independent: no subsumption relationships hold. Path coverage (PA) is computed by modifying CodeCover instrumentation to add the current statement to a vector which is cleared before each test step, allowing us to record unique paths starting from top-level methods. The path information is exact: no approximations are required, as the limit N on items in the containers also imposes a bound on most loops. The desirability of high path coverage is generally

[2]We used essentially arbitrary rewards; the exact positive magnitudes were experimentally determined to not be significant factors in performance.

agreed upon, and is a motivating factor for much work in directed testing [5]. Shape coverage (SH), a measure that to our knowledge has not been used in previous evaluations of testing approaches, simply applies the underlying rationale of the SA approach as a coverage metric: it is desirable to cover many different shapes of a container, ignoring the exact contents of the structures. Shape coverage is trivial to compute given that we have code to compute the shape abstraction — we simply maintain a set of all visited shapes: coverage is the size of this set. The final coverage measure is predicate-complete test coverage (PCT) [16] used in the major previous comparisons of lightweight methods [3], [2]. PCT measures how many combinations of *all* program predicates are covered at *all* program points: it essentially captures what portions of a Boolean abstraction have been explored, and is not to be confused with simple decision coverage. This paper adopts the same PCT coverage as used by Sharma et al., except for the new SUTs, where a similar instrumentation has been applied.

D. Fixed Duration Testing

Rather than basing an evaluation on the effectiveness of fixed-*size* test suites produced by each method, it is preferable to allow each method to test each subject for a fixed amount of time (including both generation and execution), on the same hardware, using a common framework for all coverage costs and SUT execution. This more realistically represents actual testing practice: when using automated test generation methods, a test engineer will usually have a certain budget of *time* available for testing, and hope for the best in terms of coverage and fault detection within this budget [17]. Evaluation is therefore based on total coverage obtained by each testing method, for each SUT, given 30 seconds, 30 minutes, and 1 hour in which to generate tests. A budget of only 30 seconds represents a reasonable "quick check" [4] for simple errors, such as might be performed after every compile. Budgets of 30 minutes and 1 hour show how much improvement in coverage can result from more in-depth testing, and match the kind of "over lunch" budget that is also critical for lightweight testing. A maximum heap size of 6GB was used in all experiments.

For ABP, experiments actually included 6 different possible contexts for each SUT and time budget. Generating results for 10 methods and 15 SUTs therefore required well over 150 hours of computation at the one hour budget. This prevented a useful investigation of mutant-kill rates for the chosen methods, given the added costs of executing tests over a sufficiently large set of mutants. Because results depended on random values, each 30 second and 30 minute experiment was actually performed 5 or more times, with different seeds. There was *no overlap in the rankings of methods using different seeds*: e.g, if ABP performed better than RT, it performed better for all seed values for both methods. The results shown therefore all come from using a single seed for a particular time budget (different initial seeds are used for each budget to avoid simply overlapping the earlier testing). In general, seed had so little impact even for 30 second budgets that running with one seed was sufficient to establish results for larger testing budgets where there was insufficient time to repeat experiments.

Basing experimental results on fixed durations poses a danger: if one method is implemented more efficiently than others, it will have a major advantage. Given that it is difficult to implement RT particularly inefficiently, this may produce results that favor RT. However, for lightweight methods this is appropriate, as programmers will often have to implement these methods themselves. Programmers of varying abilities are likely to all implement RT in a fairly effective fashion. Implementing SA, on the other hand, requires considerably more expertise and effort, including a correct clone method, if the cost of replay is to be avoided. Similarly, programmers are likely to implement the simplest hash-based version of, e.g, SARSA(λ). Nonetheless, in order to adjust for overhead of replay as opposed to state storage exploration in SA, all experiments used real time equal to *twice the reported time* for SA, based on a comparison with exhaustive testing and RT. No adjustment was used for ABP.

V. Experimental Results

A. Ranking the Methods

Table I shows the results of testing a large number of Java programs with all three compared techniques, for three (four in some cases, as discussed below) different testing budgets. These results summarize 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. $\times$ indicates *unique worst* coverage. For all metrics other than PCT (which often reached best coverage as quickly as 30 seconds), maximum coverage increased significantly with more time spent testing. The final three columns show the actual maximum coverage obtained by any method, and the difference (Δ) between the maximum and second-best coverage value. The density of ✓and ✓✓symbols under the three sets of columns for the three methods conveys a simple summary of the results: RT performed well on a set of SUTs similar to those reported in past work [3], [2]. ABP performed best on a slightly smaller set of SUTs, and SA performed least well. Each method performed best for at least one metric, for at least eight SUT/time combinations. SUT/time combinations in bold (25 of 55 combinations, covering 9 of 15 SUTs) indicate that ABP performed best for this configuration for at least one metric. ABP performed at least as well for PCT coverage as other metrics, even without an explicit reward. Looking more closely at the results, we can observe that ABP performed especially well on BinomialHeap, FibHeap, and HeapArray,

Table I
COVERAGE RESULTS

SUT + Time	ABP			RT			SA			Max Coverage(Δ vs. 2nd Best)		
	PA	SH	PCT	PA	SH	PCT	PA	SH	PCT	PA	SH	PCT
AvlTree 30s			×	✓✓	✓✓	✓	×	×	✓	314(144)	1073(722)	104(5)
AvlTree 30m			×1	✓✓	✓✓	✓	×	×	✓	428(96)	16925(12005)	104(1)
AvlTree 1h			✓	✓✓	✓✓	✓	×	×	✓	442(99)	22668(15320)	104(0)
AvlTree 2h			✓	✓✓	✓✓	✓	×	×	✓	446(100)	27353(17980)	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)
BinomialHeap 2h	✓✓		✓	×	×	×		✓✓	✓	2528(780)	46(3)	327(11)
BinTree 30s			×	✓✓	✓✓	✓	×	×	✓	1070(794)	3104(2111)	157(2)
BinTree 30m			✓	✓✓	✓✓	✓	×	×	✓	4487(3094)	122921(90605)	157(0)
BinTree 1h			✓	✓✓	✓✓	✓	×	×	✓	5657(3105)	224828(143751)	157(0)
BinTree 2h			✓	✓✓	✓✓	✓	×	×	×	6897(4237)	385042(262435)	157(0)
ChainedHashTable 30s			✓	✓✓	✓✓	✓	×	×	✓	28(10)	6700(1240)	6(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)
FibHeap 2h	✓✓	✓✓	✓	×	×	✓			✓	97481(72300)	4484(2943)	118(0)
FibonacciHeap 30s				✓✓	✓✓	✓✓	×	×	×	843(644)	247(54)	301(196)
FibonacciHeap 30m				✓✓	✓✓	✓✓	×	×	×	18522(14659)	5047(1343)	506(266)
FibonacciHeap 1h				✓✓	✓✓	✓✓	×	×	×	32099(24586)	8723(1614)	538(265)
FibonacciHeap 2h				✓✓	✓✓	✓✓	×	×	×	54995(39879)	15066(678)	569(256)
HeapArray 30s	✓✓	✓✓				✓✓	×	×	×	283(33)	344(256)	69(9)
HeapArray 30m	✓✓	✓✓	✓		×	✓	×		×	3107(2486)	7991(6815)	71(6)
HeapArray 1h	✓✓	✓✓	✓		×	✓	×		×	4532(3799)	13097(11921)	71(2)
HeapArray 2h	✓✓	✓✓	✓		×	✓	×		×	6808(5975)	26253(22399)	71(1)
IntAvlTreeMap 30s				✓✓	✓✓	✓✓	×	×	×	385(173)	527(152)	225(14)
IntAvlTreeMap 30m			✓	✓✓	✓✓	✓	×	×	×	624(111)	3113(526)	225(10)
IntAvlTreeMap 1h			✓	✓✓	✓✓	✓	×	×	×	638(51)	3937(326)	225(0)
IntAvlTreeMap 2h			✓	✓✓	✓✓	✓	×	×	×	641(12)	4823(247)	225(2)
IntRedBlackTree 30s				✓✓	✓✓	✓✓	×	×	×	513(300)	1079(720)	378(54)
IntRedBlackTree 30m				✓✓	✓✓	✓✓	×	×	×	708(173)	6943(2814)	379(4)
IntRedBlackTree 1h			2	✓✓	✓✓	✓✓	×	×	×	755(139)	8713(2830)	379(2)
IntRedBlackTree 2h				✓✓	✓✓	✓✓	×	×	×	836(205)	10945(2716)	379(2)
LinkedList 30s	×	✓✓	✓	✓✓		✓		×	✓	1012(132)	191(147)	11(0)
LinkedList 30m		✓✓	✓	×	×	✓	✓✓		✓	5539(1862)	186(41)	11(0)
LinkedList 1h		✓✓	✓	×	×	✓	✓✓		✓	7063(2193)	187(5)	11(0)
NCLinkedList 30s	×	✓✓	3	✓✓		✓✓		×	×	1005(203)	185(141)	21(7)
NCLinkedList 30m		✓✓	✓	×	×	✓	✓✓		×	5296(1757)	187(46)	21(7)
NCLinkedList 1h		✓✓	✓	×	×	✓	✓✓		×	6869(1683)	190(13)	21(7)
SLinkedList 30s	×	✓✓	×	✓✓				×	✓✓	699(81)	189(155)	67(4)
SLinkedList 30m		✓✓	✓	×	×	✓	✓✓		✓	3875(2240)	187(83)	67(0)
SLinkedList 1h		✓✓	×4	×	×	✓	✓✓		✓	4999(2337)	192(61)	67(1)
SplayTree 30s			×	✓✓	✓✓	✓✓	×	×		602(408)	2361(1681)	262(18)
SplayTree 30m			✓	✓✓	✓✓	✓	×	×	×	2408(957)	50870(22885)	262(3)
SplayTree 1h			✓	✓✓	✓✓	✓	×	×	×	2969(1109)	79223(29803)	262(3)
SplayTree 2h	×		✓	✓✓	✓✓	✓		×	×	3571(948)	124948(30218)	262(3)
TreeMap 30s	×		×	✓✓	✓✓	✓✓		×		461(238)	811(522)	355(22)
TreeMap 30m			×	✓✓	✓✓	✓✓	×	×		671(114)	5154(1006)	358(2)
TreeMap 1h				✓✓	✓✓	✓✓	×	×		697(82)	6637(872)	358(2)
TreeMap 2h		✓✓		✓✓		✓✓	×	×		798(117)	8733(69)	358(2)
TreeSet 30s			×	✓✓	✓✓	✓✓	×	×		537(292)	1172(747)	334(28)
TreeSet 30m	5		×	✓✓	✓✓	✓	×	×	✓	707(105)	6668(1429)	334(2)
TreeSet 1h		6	✓	✓✓	✓✓	✓	×	×	✓	805(112)	8666(20)	334(0)
TreeSet 2h		✓✓	✓	✓✓		✓	×	×	✓	871(80)	11797(916)	334(0)

Notes: **(1)** ShapeMembersCover and ShapeMembers cover 104 PCT values, tying Random; ShapeCover only covers 103. **(2)** ConcreteCover, Concrete, and ShapeMembersCover match Random at 279 PCT values, ShapeCover only reaches 277. **(3)** ShapeMembers covers 21 PCT values, tying with Random. ShapeCover only covers 16. **(4)** ShapeCover alone covers only 66 PCT values, while all other methods cover 67. **(5)** Shape has the best path coverage (724 paths), while ShapeCover only obtains 602 paths. Random performs best if we only consider ShapeCover for ABP. **(6)** Shape covers 8740 shapes, improving on Random, but ShapeCover only covers 8646, slightly lower than Random. Names of some LinkedList classes abbreviated in order to fit table.

suggesting its strength lies in testing SUTs requiring more complex input sequences. However, RT was by far the best method for testing FibonacciHeap. ABP also generally did better than RT for linked lists, arguably the *simplest* of the containers considered. Linked lists were the only subjects where SA also proved partially superior to RT and ABP.

It is going to be extremely difficult for *any* method to beat RT at path and shape coverage for some SUTs: consider only the insertion of a random permutation of 20 objects into an AVL, binary, red-black, or splay tree. *Most permutations will produce different paths and shapes*. RT, having *essentially no overhead*, will produce permutations as rapidly as possible. Adding `find` and `delete` operations does not change this fact. Learning or state storage is highly unlikely to improve on the coverage/second ratio of RT here. The more complex approaches do eventually match the *PCT* coverage of RT in most cases, with ABP only failing to reach maximum PCT coverage in 3 of 15 SUTs.

B. *The Effect of Testing Budget*

One obvious question to consider is the improvement gained by additional test budget. A large increase in coverage between the 30 second test budget and the 30 minute test budget would be expected. The budget itself is 60 times larger, and for ABP, the adaptive process initially has no information to guide its test generation. Nonetheless, the increase in the best testing method's PCT coverage only averaged a factor of 1.05. For most of our SUTs, at least one method reached the maximum observed PCT coverage after only 30 seconds of testing. Path coverage, however, increased by an average factor of 7.97, and shape coverage by an average factor of 14.07 — far less than the 60-fold increase in budget: the easiest paths and shapes to cover will be explored very early in testing. When the test budget was doubled from 30 minutes to 1 hour, both path and shape coverage increased by an average factor of 1.3. Averages, here, are somewhat misleading, given the wide variance in rates of coverage increase. For example, with the linked lists, there are only a small number of possible shapes, most of which were covered by ABP in the first 30 seconds. In contrast, RT almost doubled its coverage of the vast space of binary search tree shapes when testing time was increased from 30 minutes to 1 hour — coverage increased by a factor of 1.83. ABP actually more than doubled its shape coverage of binary search trees for the same budget increase: coverage improved by a factor of 2.51, showing that RL had actually managed to improve the quality of tests enough to overcome the problem of having already covered the easiest targets. Nonetheless, RT improved on ABP's 1 hour shape coverage by over 143,000 shapes. It appears possible that, given a large enough testing budget, ABP would eventually improve on random testing. In contrast, in most cases where ABP was better than RT, the coverage increase factor for ABP was *larger* than the improvement for RT: ABP was increasing its lead (e.g., for heap arrays, the respective factors of 30 minute to 1 hour improvement in path/shape coverage were 1.45/1.64 for ABP and 1.18/1.39 for RT). The few cases where this relationship did not hold appear to be cases (shape coverage for BinomialHeap and the linked list structures mentioned above) where ABP-based testing had (almost) reached maximum attainable coverage with a small budget.

While coverage often increased significantly from 30 minutes to 1 hour, the best testing method *never changed* as a result of the additional half hour of testing (in some cases, an additional method became tied on PCT). In order to see if another increase in budget would result in any changes, additional experiments, using only the ShapeCover configuration for ABP, were performed with a 2 hour testing budget, for those SUTs where RT was outperforming ABP on both path and shape coverage, or vice-versa, with a 1 hour budget. For the TreeMap and TreeSet classes, this additional doubling of test budget allowed ABP to improve its shape coverage sufficiently to better that obtained by RT. For BinomialHeap, SA discovered 3 more shapes than ABP. For all cases where ABP-based testing was superior at 1 hour, the difference between ABP and RT increased: ABP's learning widened the gap.

C. *Single-Test Case Rewards*

A modification of the ABP approach considered above is to base the reward on coverage per test-case rather than cumulative coverage by suite: i.e., to optimize for the "best" length M test case, rather than for the best suite. This approach properly respects episode boundaries: one learning interaction does not change the probability of reward in a future interaction. This approach not only always lowered the effectiveness of ABP in producing good whole-suite coverage, but *frequently lowered the best single test case coverage produced*. At least for the SUTs studied in this paper, a "universal" reward approach seems to be superior to per-test-case rewards, even if producing a single very effective test case is the goal of testing.

D. *Testing An HTML Parser*

As a preliminary experiment in applied lightweight methods to systems larger than typical container classes, we applied ABP, RT and SA to an HTML parser included in the Lobo project (http://lobobrowser.org), a pure-Java web browser specialized to support Rich Internet Application (RIA) languages. The parser code itself (our testing target) is about 975 lines of code, supported by a much larger HTML document class and other infrastructure. We measured coverage only over the code we were targeting, the scanning and parsing engine. The choices in this experiment were no longer API calls but tokens, including 15 core tags, brackets, etc., obtained by a simple static scan of the parsing code, as is common in RT practice when randomness at the level of complete fuzz is unlikely to be productive. In order to show how well ABP can work when essentially no effort is put into context definition, we used the actual string to be parsed as our context, and at each "step" of testing extended the string by one choice and re-parsed.

In our experiments, covering maximum "document" sizes of 15, 25, 50, 100, and 200, test times of 5 minutes, 30 minutes, 1 hour, and 2 hour, and a number of random seeds, ABP and RT were essentially tied in terms of statement and branch coverage, the winner in each experiment varying seemingly at random; SA, due to a large branching factor and low shape equivalence, never generated long enough tests to find interesting behavior, and is ignored in the following results. Even with 2 hours of testing, the best method (by one branch and four statements guarded by that branch) changed with the chosen random seeds and test length, with no clearly superior method. In our container class results, RT usually worked better than other methods for very small test budgets (30s); here, ABP improved by 8 branches and 8 statements for some seeds and test lengths. In all experiments, even when ABP missed the "hard to hit" branch, it improved on RT for path and shape coverage by a factor of at least 2x (e.g., 48,871 paths vs. 18,421 and 51,550 shapes vs. 18,496). Even more surprisingly, in more than three quarters of our experiments ABP executed more tests than RT — presumably purely random input tends to produce more inputs that take a long time to parse (this "slowness" was not correlated with branch/statement wins). It is hard to generalize from one limited case study, but it appears that ABP can explore some behaviors that RT does not explore, even for a slightly more real-world program.

E. Threats to Validity

One threat to construct validity is the use of fixed duration testing. The test code (other than actual generation) and execution environment (actual machine, OS, load, etc.) were held constant, but the implementations are possibly inefficient. The primary threat to external validity is obviously that these results are almost completely from container classes, and difficult to extrapolate to other kinds of programs.

VI. Explaining ABP's Effectiveness

Consider two test suites for the BinomialHeap class, each produced with a 20 minute test budget. RT produces 1,357 test cases of length 200, while ABP-based testing can only execute 659 test cases due to the overhead of learning and other factors. Nonetheless, the ABP and RT both cover 179 statements and 44 branches, and ABP improves on RT in all other metrics: 1,112 vs. 704 paths, 40 vs. 25 shapes, and 327 vs. 310 PCT values.

What difference in the suites produces these coverage differences? Test cases for BinomialHeap consist of calls to `insert`, `delete`, `extractMin`, and `decreaseKeyValue`. The `insert` and `delete` methods each take one integer parameter (limited to 20 values in our test configuration), the `extractMin` method takes no parameters, and `decreaseKeyValue` requires two integer parameters. Both suites cover almost the same range of API calls: RT calls each possible combination of method and input value at least once; ABP also calls `insert` and `delete` (and `extractMin`) with all possible values, but only calls `decreaseKeyValue` with 315 of the 400 possible inputs. The distribution of calls, however, is very different. The RT suite is, of course, essentially evenly distributed — roughly 1/4 of calls are to `extractMin`, with the 40 possible `insert` and `delete` calls each appearing at a rate of 1.19%-1.29% of the time, and `decreaseKeyValue` calls all around 0.07%. Table II shows that the ABP suite is radically different in distribution, with almost half of all calls made to `insert(0)`, and the 4 most frequent calls accounting for over 75% of all calls. Can we then view ABP as "RT with an automatic biasing of probabilities"? This is not the case: the frequency of `insert` is highest in the first tests executed, and lowest in the final tests. ABP degrades its bias towards behavior as it repeatedly explores it without obtaining new coverage. Unlike the Nighthawk genetic algorithm approach, ABP-based testing does not seek a most-fit mix of method calls, but continually shifts its bias as it gains (or fails to gain) reward.

Table II
CALL DISTRIBUTION, ABP TEST SUITE FOR BINOMIALHEAP

Call	**% of All Calls**
Complete Test Suite	
`insert(0)`	46.4%
`delete(0)`	10.7%
`decreaseKeyValue(0,0)`	10.1
`extractMin()`	7.9%
`insert(1)`	1.5%
`delete(1)`	0.27%
`decreaseKeyValue(14,14)`	0.24%
`decreaseKeyValue(18,15)` (+ 70 others)	0.0015%
`decreaseKeyValue(6,1)` (+ 130 others)	0.00076%
1st Quarter of Test Suite	
`insert(0)`	60.4%
`decreaseKeyValue(0,0)`	7.69%
`delete(0)`	6.66%
`extractMin()`	5.88%
4th Quarter of Test Suite	
`insert(0)`	34.6%
`decreaseKeyValue(0,0)`	13.6%
`delete(0)`	12.65%
`extractMin()`	9.55%

Moreover, ABP testing is not stateless. More than 83% (551 of 659) of ABP test cases begin with a call to `insert(0)`. The frequency of such first-step calls is 86% in the first quarter of tests, falls to 80% and 82% in the next two quarters, but returns to 87% in the final tests. It does not matter which value is given to `insert` (though we would expect it to be one also frequently given to `delete` and `decreaseKeyValue`), but it is true that *future coverage is maximized in a test that begins with* `insert`. The 10% chance of exploration ensures that the ABP suite does call each of the other functions occasionally in the first step of a test case, thus covering the functions on an empty heap.

Groce and Visser [18] proposed the use of *structural heuristics* in software model checking as an alternative to the more common practice of heuristics derived from a property to be checked. A prominently featured heuristic in this work was the *branch counting heuristic* which gave heuristic priority to (1) states covering an uncovered branch followed by (2) states reached without taking any branches followed by (3) states covering an already-taken branch, in order of how many times the branch has been taken (fewer visits = higher priority). The idea was to focus model checking on portions of a state space that had been less frequently explored, which proved useful for finding a flaw in a real-time operating system kernel. We believe ABP and structural heuristic model checking both bias exploration towards parts of a state space that have been covered least and naturally degrade the bias towards certain actions or states as the state is repeatedly explored. ABP-based testing is thus in a sense one hybrid between random (backtrack-free) search (RT) and heuristically guided backtracking search. The middle ground is RL, which is backtrack free, but updates a local estimate of action value based on feedback. In a stationary reward, finite, setting, all three methods are expected to converge to the same complete exploration. ABP has some key advantages over structural heuristic model checking. ABP is lightweight, not requiring backtracking. ABP uses RL to maximize a reward based on a complex mix of diverse metrics (or custom rewards) rather than using a fixed heuristic based on a simple coverage metric. The explore-exploit paradigm, by imitating RT 10% of the time, avoids local minima that may easily trap model checking. Pure testing in place of backtracking model checking also allows us to exploit abstract state-space representations in the context without missing some states due to unsoundness. The failure of rewards based on single-test coverage observed above is likely an artifact of losing the benefits of decaying reward: a 10% exploration rate is not enough to overcome the tendency to repeatedly execute any "good" test. We speculate that ShapeCover performs so well partly because the coverage count allows the policy to combine this essential global decay with a single-test decay of expected reward (e.g., calling `insert` less towards the end of each test).

VII. Related Work

The problem of generating test input sequences for software systems is long-standing and widely studied. Recent work on lightweight methods has focused on random testing [9] and shape abstraction [3], [2]. The one previous work on using reinforcement learning specifically in software testing, to our knowledge, is that of Veanes et al. [19], which considered only model-based online testing of reactive systems, and provided limited experimental results on a single toy robot model. More importantly, the reward considered was based on a planning-type problem, with the robot collecting cans (a standard example in the RL literature [12]). The present paper builds on our initial brief introduction of the idea of using coverage metrics as a basis for reward [10] in RL for testing. The general idea of "learning" tests has also been studied. Andrews et al. [20] used genetic algorithms to generate data for random unit testing in the Nighthawk tool. ABP is similar to Nighthawk in that both approaches learn *how to construct test cases* rather than learning an ideal set of test cases. The primary algorithmic difference (beyond the use of RL vs. genetic algorithms) is that ABP learns what method to call and what input to provide based on a context, while Nighthawk learns overall method weights to optimize random test generation.

The critical difference between ABP and search based [21] testing is that search-based or evolutionary testing (by genetic programs or other means) considers the fitness of individual tests, and has as its goal a set of good tests. ABP is more like aggressive random testing, in that each individual test is not expected to be high-quality; the goal is to produce many tests quickly to cover an SUT's behaviors, not to generate an efficient small test suite for regression. The policy in ABP is *not* a recipe for good tests, but an evolving description of a "search frontier", biased against producing coverage of already-covered behavior. Another critical difference between the present work and previous efforts is the focus on a learning-based method that is (nearly) as easy to apply as RT, allowing programmers to essentially "drop it in" when RT is ineffective.

Adaptive random testing (ART) [22] modifies traditional RT by sampling tests and only executing the test most "distant," as determined by a distance metric over inputs, from all previously executed tests. The choice of a distance metric in ART is a burden on a test engineer, similar to our requirement that the test engineer choose a context. It seems likely that in many cases devising a distance metric will be harder than devising a context. ART has been shown to be only effective under fairly limited circumstances for real world programs [23], in part due to the overhead of computing distance metrics. While ABP also imposes an overhead on testing, it is able to provide benefits that go beyond avoiding oversampling, by actually learning structure. ART has usually been applied to numeric problems where distance metrics are fairly natural, rather than API calls in lengthy sequences, the natural domain for ABP.

VIII. Advice to Practitioners and Future Work

In general, the experimental data does not support a claim for dominance for either RT or ABP. Both methods are useful for different subjects, with ABP better for 7 of our 15 subjects. It seems reasonable to propose using *both* methods if sufficient test budget is available; the gain in terms of testing for running both RT and ABP with a 30 minute budget, at least once, then using the best of the two methods in the future, surely outweighs the cost of 30 minutes of compute time, and the same harness can easily be used for both methods. For *very* small test budgets RT is best: it almost always outperformed other methods for a 30 second budget, and other studies have shown that RT bests even model checking for small budgets [24]. When testing small library-style SUTs like container classes, we would choose ABP as our *first* test case generation method in cases where we expect that RT will tend to redundant tests, or where inputs are more complex than a call plus one parameter (e.g., heaps), unless context is hard to define. ABP appears to be a good replacement for the previously recommended SA approach in most cases where RT is not particularly effective: whenever there is a suitable shape to use as a basis for abstraction, that shape looks likely to be better used as an ABP context. In short, ABP extends the set of truly lightweight automated testing methods, making improved reliability available at low cost for programmers working in environments without concolic testing or model checking.

We plan to apply ABP to more complex SUTs, given its success on the HTML parser. Appropriate contexts for ABP in more complex software systems pose an interesting challenge, where the literature of unsound abstractions in model checking may provide insight. Unfortunately, the unbiased comparison made possible by automated context representations for container classes will be impossible: context quality is likely to be a key factor for complex SUTs. However, "toString" based contexts such as those used here are likely to be effective for many of the small, simple (but hard to code correctly) modules where lightweight automated testing is most desirable. While our current approach to ABP-based testing clearly "works," it is likely that context and reward tuning will be useful, just as feedback can considerably improve the effectiveness of RT [1]. Most importantly, it is highly likely that the RL algorithm used by ABP can be tailored to the structure of the test input generation problem, incorporating the non-stationary reward into learning, allowing programmers to obtain better results simply by using learning *designed* for software testing.

Acknowledgment

The authors would like to thank Darko Marinov, Milos Gligoric, Jamie Andrews, and Willem Visser for helpful discussions related to this work. A portion of this research was funded by NSF grant CCF–1054786.

References

[1] 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.

[2] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov, "Testing container classes: Random or systematic?" in *FASE*, 2011, pp. 262–277.

[3] W. Visser, C. Păsăreanu, and R. Pelanek, "Test input generation for Java containers using state matching," in *ISSTA*, 2006, pp. 37–48.

[4] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of haskell programs," in *ICFP*, 2000, pp. 268–279.

[5] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Programming Language Design and Implementation*, 2005, pp. 213–223.

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

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

[8] A. Groce, G. Holzmann, R. Joshi, and R.-G. Xu, "Putting flight software through the paces with testing, model checking, and constraint-solving," in *International Workshop on Constraints in Formal Verification*, 2008, pp. 1–15.

[9] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*. Wiley, 1994, pp. 970–978.

[10] A. Groce, "Coverage rewarded: Test input generation via adaptation-based programming," in *IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 380–383.

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

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

[13] D. Andre and S. Russel, "State abstraction for programmable reinforcement learning agents," in *National Conference on Artificial Intelligence*, 2002.

[14] S. Mahadevan, "Agent reward reinforcement learning: Foundations, algorithms, and empirical results," *Machine Learning*, vol. 22, no. 1, pp. 159–195, 1996.

[15] 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.

[16] T. Ball, "A theory of predicate-complete test coverage and generation," in *FMCO*, 2004, pp. 1–22.

[17] J. H. Andrews, A. Groce, M. Weston, and R.-G. Xu, "Random test run length and effectiveness," in *Automated Software Engineering*, 2008, pp. 19–28.

[18] A. Groce and W. Visser, "Heuristics for model checking Java programs," *Software Tools for Technology Transfer*, vol. 6(4), pp. 260–276, 2004.

[19] 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.

[20] 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.

[21] P. McMinn, "Search-based software test data generation: A survey," *Software testing, verification, and reliability*, vol. 14, pp. 105–156, 2004.

[22] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Adv. in Computer Science*, 2004, pp. 320–329.

[23] A. Arcuri and L. Briand, "Adaptive random testing: An illusion of effectiveness," in *International Symposium on Software Testing and Analysis*, 2011, pp. 265–275.

[24] A. Groce and R. Joshi, "Random testing and model checking: Building a common framework for nondeterministic exploration," in *Workshop on Dynamic Analysis*, 2008, pp. 22–28.