Learning-Based Test Programming for Programmers

Alex Groce, Alan Fern, Martin Erwig, Jervis Pinto, Tim Bauer, and Amin Alipour

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

Abstract. While a diverse array of approaches to applying machine learning to testing has appeared in recent years, many efforts share three central challenges, two of which are not always obvious. First, learning-based testing relies on adapting the tests generated to the program being tested, based on the results of observed executions. This is the heart of a machine learning approach to test generation. A less obvious challenge in many approaches is that the learning techniques used may have been devised for problems that do not share all the assumptions and goals of software testing. Finally, the usability of approaches by programmers is a challenge that has often been neglected. Programmers may wish to maintain more control of test generation than a "push button" tool generally provides, without becoming experts in software testing theory or machine learning algorithms, and with access to the full power of the language in which the tested system is written. In this paper we consider these issues, in light of our experience with adaptation-based programming as a method for automated test generation.

1 Introduction

The combination of machine learning (ML) and software/hardware correctness is now well-established as a fruitful intersection. In some cases, learning is used to aid complete verification: in these approaches, the stochastic nature of most machine learning is incidental. The learning is either by nature complete (e.g., using Angluin's algorithm [1] to learn a bounded finite state machine [2–4]) or intended to help produce an effective abstraction for model checking [5–7][1]. While the learning in some of these cases may not be guaranteed to reach a correct answer, it is easy to determine if the current answer is satisfactory: namely, when an abstraction produces either a proof of correctness or a valid counterexample, the current learned hypothesis is in no need of further refinement. Additionally, in these settings, the "user" for machine learning is a essentially a model

[1] In some cases, abstraction learning uses SAT to solve for a "perfect" hypothesis over current information, rather than a traditional ML algorithm; the approach is nonetheless learning, as it features an inductive bias and may be modified in response to future "training set" examples. The connection between CEGAR [8] and active learning is intuitively clear.

checking algorithm; a human user of adaptive model checking or a learning-assisted Counterexample Guided Abstraction Refinement (CEGAR) [8] system need not even be aware that learning is taking place "underneath the hood", and is highly unlikely to have useful knowledge for improving the effectiveness of learning. Moreover, because a "correct" answer can be mechanically checked, there is typically no need for a user to assess the effectiveness of the learning algorithm.

Another large body of recent work at the intersection of ML and system reliability, however, has focused on learning to produce an effective test suite for a program, whether by genetic/evolutionary techniques [9, 10] or by reinforcement learning [11, 12]. In these cases, learning is obviously not expected to be complete (there is no final, "correct" test suite), and evaluating the effectiveness of testing techniques is notoriously difficult, since testing is typically applied to precisely those systems where complete verification is not feasible, the set of all faults for realistic systems is seldom if ever known, and coverage and other metrics of test suite quality are of varied and difficult-to-predict effectiveness [13–16]. In these cases, therefore, a human user often must assess the effectiveness of an "answer" provided by an ML algorithm, without a general mechanical means for evaluating its value. Moreover, even automated testing is usually much less of a "black box" approach than model checking, and the user is likely to want to or need to influence the choices made by a search or learning algorithm, or tune its heuristics or reward structure.

Efforts where the machine learning is directed towards testing a program, must therefore address a set of *three* potential core challenges:

1. First, learning-based testing is inspired by the idea of using machine learning to adapt the set of tests run to the Software Under Test (SUT): the learning problem is to choose new inputs based on the behavior observed on past inputs. Obviously, this problem is recognized and addressed by all learning/search-based testing approaches: proposing a solution to this problem defines the field! This problem is essentially equivalent to the core problem in ML for verification — the application of machine learning to a computational problem.
2. Second, in some cases, the learning techniques proposed may make assumptions that do not (quite) match the aims of software testing. In particular, for reinforcement-learning (RL) [17] based techniques, the goal of maximizing total coverage can lead to formulations of reward that do not naturally fit the usual RL assumptions.
3. Finally, programmers often want to experience the benefits of automated test generation without completely abandoning part of their test effort to a "black box" that is given, e.g., a set of method calls and input types, or a numeric input range, and produces tests. Effective random testing tools for complex system software, for example, are often highly engineered artifacts with special-casing, test case filtering, human-designed feedback, and hand-written specifications written in a standard programming language [18, 19]. In fact, random testing's popularity among programmers (even among those

who might be more inclined to prove programs correct, such as Haskell users [20]) is likely due to the combination of ease of use, effectiveness, and control that it provides programmers. The "interface" to random testing's core test-generation approach is simply a method for producing a pseudo-random number, which is provided as a standard library by all popular programming languages. The gap between test programming and "normal" programming in random testing essentially vanishes, and the only "tool" that must be understood is, typically, a small set of API calls.

In this paper, we discuss our experiences with these three challenges, in the context of one ML-based approach to software testing based on adaptation-based programming (ABP) [21]. Rather than new technical contributions or experimental results, the primary aim of this paper is to bring the second and third challenges of ML-based testing to the attention of other researchers and practitioners. We first briefly introduce ABP-based testing [12], then discuss the problems of adapting testing to learning (Section 2), adapting learning to testing (Section 3), and adapting test programming to programmers (Section 5), primarily focusing on the latter two, less widely recognized, challenges.

1.1 Adaptation-Based Programming

Adaptation-based programming (ABP) [21, 22] is a novel approach to programming that allows a programmer to exploit reinforcement learning (RL) [17] to "implement" difficult algorithms. Rather than writing code to compute a value, the programmer simply asks an ABP-library to "suggest" a value, given a context (the context is the formulation of the current state of the system). The programmer then rewards the ABP library based on how good the suggestion is. The ABP-library uses a reinforcement learning algorithm to attempt to optimize expected reward.

RL is an approach to the problem of learning controllers that maximize expected reward in controllable stochastic transition systems. 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 [23, 24]. 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 and rewards.

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 string, e.g. `'X-XO-OO-X'`), and provide a positive reward if the move proposed resulted in a win (see Figure 1 for pseudo-code). Each game would constitute one "episode" of learning,

```
playGame():
   ABP.beginEpisode();
   while (!gameOver()) {
      context = boardState();
      move(ABP.suggest(boardState()));
      if (victory())
         ABP.reward();
      opponentMove();
   ABP.endEpisode()
```

Fig. 1. Pseudo-code for ABP-style Tic-Tac-Toe

since moves in previous games have no influence on 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 RL) should improve its play; for a simple game like tic-tac-toe this might only take a small number of iterations. A key point is that the programmer need not be aware of the concept of SMDPs underlying this adaptation to the reward function. The programmer only needs to be able to generate a good description of the current state and a reasonable evaluation of choices made.

The ABP library referred to in this paper, available for download on the web [25], makes use of a popular reinforcement learning algorithm called SARSA(λ) [17]. At the heart of SARSA(λ) 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 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. exploited) with probability $(1 - \epsilon)$ while an action is chosen randomly (i.e. explored) 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. Our 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.

```
import abp.*;
...
public enum TestOp implements java.io.Serializable
    INSERT,REMOVE,FIND;
    public static final Set<TestOp> AllVals =
      unmodifiableSet(EnumSet.allOf(TestOp.class));
...
  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 context/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();
      Object r1, r2;
      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();
  }
```

Fig. 2. Adaptation-Based Programming: a Simple Example

1.2 ABP-Based Testing

The key insight of ABP-based testing is that a programmer can take a similar approach to generating tests for a program with a clear API or other stateful input-definition. Rather than selecting moves in a game, she lets the ABP library select methods to call and parameters for the selected method calls for the program being tested (the SUT). In practice, the programmer essentially writes a random testing harness, replacing calls to a pseudo-random number generator 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 container initialization until we begin a new test on a new container) constitutes an episode. Figure 2 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. 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; conceptually, the ABP library's interface is only slightly more complex than that of a typical pseudo-random number generator. Note that 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 pure random testing, we might expect poor results when the size of the domain is too large). The key question is now: what can the programmer reasonably 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 chooses randomly, effectively duplicating random testing. 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 ensures that testing is likely to improve over time but that exploration is never abandoned.

Note that in some sense this 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. This approach to reward derives from typical methods for evaluating software test suites: for any coverage metric, the "score" for a suite is typically based on treating the suite as a "hitting set" for the coverage targets: in typical usage, if suite A takes 100% of all program branches precisely once each and suite B covers 90% of all branches, but takes each branch 10 times, we simply say that suite A "has better branch coverage." Even using a set of coverages (including path, branch, and statement) as in our framework only complicates this essential fact: repeated exploration is not considered valuable, in and of itself. In the usual RL setting, e.g., game playing or planning, reaching a goal in future episodes is just as good as reaching it the first time — e.g., there is no penalty for winning a game in the same state as in a previous game. This property of rewards is known as stationarity. Experimental results [12] indicate that this unusual reward structure does not prevent the ABP library from learning a policy that, over time, improves test suite coverage. Informally, we can think of this setting as playing a game against an opponent who never "falls for" the same trick twice — but exploring strategies similar to those that recently proved successful increases the chance of finding a new way to win.

Out experimental results indicate that ABP can be effective for testing, at least for container classes and an HTML parser, even with no tuning of the RL algorithm to the problem, and no programmer tuning of the contexts used or reward structure beyond a naive combination of string linearizations and "off-the-shelf" coverage metrics.

2 Adapting Testing to Learning

The previous section of this paper presents one approach to the problem of adapting testing to learning. Many other approaches are possible, but all are essentially applications of some learning or search algorithm to the problem of test generation, and in this sense typical of much applied machine learning research. The nature and importance of this problem is widely understood. We do believe that one aspect of this problem (related to the generally difficult problem of evaluating test suites) may merit further attention, but must delay the discussion of this idea (in Section 4) until after we place it in the context of adapting learning to testing.

3 Adapting Learning to Testing

ML-based testing has mostly applied off-the-shelf techniques to the problem of software testing. While using ML as a "black-box" is a good start, it unfortunately treats software testing as "just another domain." This is in contrast to much work in the ML community, where learning algorithms are often developed to leverage the structure of a problem, particularly for those with significant applications. Such a learning algorithm can be expected to perform far better than a more general one. The problem of testing software is of sufficient importance to warrant a learning algorithm explicitly designed for it. Therefore, in this section,

we attempt to identify the specific characteristics of software testing, viewed as a class of learning problems. We use the gained insight to propose extensions to the existing ABP-based testing framework followed by sketching an outline of RL algorithms that might be better suited for our purpose.

3.1 Assessment of ABP-based Software Testing

In section 1.2, we briefly described how the current use of the ABP system violates one of the fundamental assumptions of the underlying RL algorithm, namely, the stationarity of the reward signal. The assumption of stationarity means that the expected reward we get for being in a particular context c does not depend on how many times we have visited a context. However, this is clearly violated in the software testing setting, where the reward for being in a context, will typically decrease each time the context is re-visited. At the extreme, if a context corresponds exactly to a program state, then the reward would often be zero after the first visit, depending on the kind of coverage considered.

However, despite the non-stationarity, positive experimental results indicate that the learner can still use the feedback to improve testing performance. In order to understand this behavior, it is useful to note that each context c used in software testing actually represents an entire class of program states. It appears that this fact leads to a useful form of generalization that the ABP system is able to exploit. In particular, if a context is visited for the first time and results in a positive reward, then the ABP system will tend to estimate that the context will have a positive reward in the future. This is a good assumption when that context describes a set of unique but similar program states that will each generate rewards on the first visit. In this situation, the ABP system will tend to bias the exploration of the program executions toward such promising contexts.

$$Q_{t+1}(c, a) = Q_t(c, a) + \alpha z(r_t) \tag{1}$$

Of course, after visiting a context many times, we can expect that the positive rewards will become rare. Unfortunately, this is where the ABP system will run into difficulty. The value estimates maintained by the ABP system, which are used to select its actions, are averages of observed reward sequences (see Equation 1). It can take significant time for this average to track the change in reward for a context and in the meantime the ABP system will continually explore the now exhausted context, wasting program executions. Eventually the system will learn that the previously attractive contexts are exhausted and then explore more promising areas. It is easy to imagine situations where this type of behavior can lead an ABP-based system to perform worse than random testing. In particular, this will happen when the early positive impact of ABP's biased exploration does not counteract the later inefficiencies resulting from slowly realizing a context is no longer rewarding. In what follows, we propose a number of simple extensions to the current ABP system that might allow it to better leverage this insight.

3.2 Dealing with Non-Stationarity

One approach to dealing with non-stationarity is to use a more refined context. For example, if we augment a given context with the counts of how many times we've visited the context, then the reward signal will appear to be much more stationary. However, there is a serious drawback to this "solution." The number of possible contexts would increase substantially and this would reduce the ABP system's ability to generalize, which was one of our main hypothesized reasons for the ABP system's current success. Thus, simply increasing the scope of the context does not appear to address the fundamental issue.

Another approach would be to adjust the way that the system updates the Q-values $Q_A(c, a)$, which estimate the value of taking action a in context c for adaptive A. Recall that these values are used to select an action to execute in a given context. Currently, after each program trajectory, these values are updated as a moving average of past rewards and the newly observed reward. As mentioned above, this averaging process can be quite slow with respect to realizing that a previously good Q-value is now bad. A solution would be to place more weight on the newly observed rewards in the update equations. This is certainly a reasonable engineering approach to the problem, that deserves further investigation. However, there is no clear principle for selecting the particular weighting scheme, which is likely to vary from problem to problem, leading to robustness concerns.

A more principled approach would be to explicitly define a non-stationary reward model that makes sense for software testing and to modify the RL algorithms to take that model into account. In particular, this reward model should encode the notion that subsequent visits to a newly discovered context are likely to produce a reward pattern that tends to decrease toward. Working out the technical details of this approach is an interesting research problem and would suggest new update mechanisms that would actively try to estimate non-stationary changes and correct Q-value estimates accordingly.

The last observation raises an interesting question: Is learning online, as we do here, the best way to apply RL to software testing? A fundamentally different approach is training offline on a diverse set of programs which has the potential of improving generalization between contexts. Furthermore, training offline produces a useful policy that can be applied off-the-shelf to testing a SUT which is desirable since its performance can be carefully evaluated before deployment and optimized for efficiency. This approach requires a carefully engineered context encoding for an SUT (perhaps by ML experts) which seems feasible. For instance, we may include features that compute counts of the number of new states seen from a given $(context, action)$ pair, number of visits since the last reward was seen, and so on. It opens the door to using efficient feature vector representations which typically achieve better generalization compared to the current tabular approach. If required, we may even have different contexts corresponding to fundamental differences in search spaces.

3.3 Monte-Carlo Tree Search

The ABP approach has so far focused on controlling exploration by biasing random walks according to continually adapting Q-value estimates. There are, however, other approaches for exploration developed in the machine learning (and more generally, the AI) field, that also deserve attention. One particularly promising class of algorithms is known as Monte-Carlo Tree Search (MCTS) [26], which has demonstrated tremendous success in recent years, most famously for its major advances in computer Go [27]. In the context of software testing, MCTS can be viewed as a way of building a tree of program executions in a way that is biased toward more promising areas of the tree. Each iteration of MCTS would correspond to selecting a program execution, where the actions at adaptives are selected in a way that attempts to balance exploration with exploitation of actions that look more promising based on past executions. One of the key contributors to the recent success of MCTS is the use of modern rules for managing this explore/exploit tradeoff in a theoretically rigorous way that works well in practice.

MCTS seems well-suited for testing software since any good adaptive method of testing software within a time budget should perform a careful exploration of an unknown search space. It is easy to modify the existing ABP library so that MCTS could be run under the hood rather than RL, with no noticeable difference to the tester with respect to writing the adaptive test program. However, like RL, MCTS also assumes the search problem involves stationary rewards. Thus, an interesting research direction is to consider variants of MCTS that capture its strengths while taking into account the non-stationary nature of the software testing search problems. As for RL, there are a variety of starting points for doing this, the most promising of which is to explicitly build a model of non-stationary reward into an MCTS algorithm, which continually tries to estimate the non-stationarity and account for that in its explore/exploit behavior.

4 Adapting Testing to Learning, Revisited

One mitigation of the problem of non-stationary reward is to abandon the typical software testing evaluation of test suites as hitting sets. While re-visits of coverage entities should almost certainly be de-valued according to some discount function, considering one execution of a branch in a test suite to be just as effective for testing as multiple executions is not particularly intuitive. Certainly, when evaluating randomly-generated suites in terms of fault detection, test engineers prefer suites that detect a fault multiple times to suites that only detect a fault once, on the grounds that the later method has a high probability of not detecting the fault at all [28]. Model checking heuristics based on structural coverage have used such a discounted (rather than binary) approach successfully [29, 30]. Note that this approach, to our knowledge not applied in learning-based approaches to date, only reduces the non-stationarity of the reward, rather than completely removing it. We do not believe that considering a suite that executes

one branch 1,000 times "just as good as" a suite that executes 1,000 branches once is wise, so some discount for revisits is clearly required.

5 Adapting Test Programming to Programmers

It seems rather obvious that a program can be tested only after it has been written. This view can easily lead to the assumption that test cases for a program also have to be created after it has been been written. This perspective leads to a decoupling of testing from programming, which has the danger of making testing seem more like an optional part, something that can be left out. In a sense, this is the point of view taken by testing approaches that take as input a program, its input structure, and possibly a specification, and output a set of tests, whether this generation is based on machine learning or some other technique.

That testing can be integrated well with programming has been impressively demonstrated by the QuickCheck tool for Haskell [20]. QuickCheck provides an easy way for a programmer to generate random values of almost any predefined or user-defined type. The programmer implements tests by writing Haskell functions that represent properties to be checked. These properties can then be tested using the automatically generated data. The fact that tests for Haskell code are written as Haskell functions as well as the fact that automatic test data generation is also expressed within the program to be tested leads to a testing system that is tightly integrated into the language that is to be tested.

Specifically, QuickCheck is a *domain-specific language* for testing. A domain-specific language (DSL) offers notations and abstractions that are designed to work in a specific application domain [31]. DSLs can be implemented in quite different ways. Most importantly, we can distinguish between *external* and *internal* DSLs. An external DSL is implemented as a stand-alone product, which means that it has complete control over the syntax of the DSL, which is one of the major advantages of external DSLs. On the other hand, the implementation of an external DSL is usually quite complex and often difficult to adapt. In contrast, an internal DSL is implemented as an extension of an existing language (called the *host language*) and uses constructs of the host language as part of its syntax. Internal DSLs are also called domain-specific *embedded* languages (DSELs) [32]. DSELs are easier to implement and adapt since they can reuse much of the infrastructure of the host language. For example, all functionality for arithmetic or string processing is immediately available whereas these have to be reimplemented in an external DSL. QuickCheck is a DSEL in Haskell and it depends crucially on the fact that it has direct access to Haskell code. It is hard to imagine a version of QuickCheck implemented as an external DSL, which would essentially have to re-implement a significant part of, if not all of, the Haskell language.

We find ourselves in a quite similar situation for ABP. While QuickCheck is a tool for deriving properties of a program, ABP is a tool for changing programs. It is in a sense a metaprogramming tool. But much like QuickCheck, ABP needs

access to the program it is supposed to adapt — which is precisely what the Java ABP library provides, a language embedded in Java that offers constructs to produce adaptive Java programs.

The combination of ABP with testing further leverages the integration into the host language and makes it possible to base the adaptation process of test cases on information obtained directly from the program to be tested during the testing process itself. In particular, in contrast to test-generation approaches that operate as external tools, ABP-based testing can "talk to" the program being tested (and its host language) with great ease, letting programmers assert as much or as little control over the testing process as with random testing. This gives programmer some extremely useful abilities:

- A context in ABP can be anything that can be computed by the SUT or by auxiliary functions in the host language. In our container class experiments, "system state" was produced by simply calling `toString`, abstracting the result with a simple string-processing function written in Java, and merging in some single-test coverage results.
- The reward in ABP can, again, be computed by arbitrary code. It can make complex context-sensitive evaluations of test fitness easy by directly inspecting system state. There is no need to express desirable properties of the system's behavior in any language other than that of the implementation itself.
- Similarly, if programmers wish to introduce new coverage metrics that generalize to more than one program, they can program these instrumentations in Java itself, making use of reflection. Our own implementation uses automatic instrumentation to compute path coverage based on Java-coded branch and statement coverage provided by CodeCover [33].
- A programmer can "override" ABP when needed — if certain behaviors in rare states are known to lead to known faults, for example, a hand-coded choice function can be used in place of the ABP suggestion.
- Contracts/properties can be implemented directly in SUT terms, without the need to learn a new property language.
- ABP-based testing, as noted before, "looks like" simple random testing (or generalized unit testing) to a large extent, and does not require a programmer to leave the "comfort of home" by changing languages or running an external tool.

In a sense, ABP-based testing has some similarities to the approach to checking C code introduced in version 4.0 of the SPIN model checker [34, 35]. The ability to directly call C code, check properties, and bias model checking exploration based on C-language constructs made it much easier to model check large, complex C programs in SPIN [36, 37]. C served as a "DSEL" for SPIN's PROMELA language; in our case, we are similarly describing a search problem, with the added advantage of not requiring programmers to switch between two languages (SPIN and PROMELA): testing and tested code are both written in Java, and have the same language of discourse.

In short, the realization of ABP as a DSEL is crucial in feeding test-relevant program information into the adaptation process, and it is this language design decision which contributes significantly to the style of ABP-based testing that makes it into an interesting new opportunity for programmers.

6 Conclusions

While the use of machine learning (and related AI approaches) in testing has already proved fruitful, we believe that the full potential of this combination can only be reached when research efforts move beyond formulating testing problems as machine learning problems to consider two additional aspects that distinguish learning-based test generation from the use of machine learning in model checking:

- First, while off-the-shelf ML/AI approaches may work well for testing, many of the most effective uses of ML involve leveraging the structure of a unique problem domain with new machine learning algorithms specially suited for the nature of the problem at hand.
- Second, the adaptation of learning-based testing by users outside the research community may be greatly speeded by placing test generation in a context that such users already understand: namely, the language in which they are developing the Software Under Test. Such an approach not only makes using ML to produce tests more appealing to programmers; it also gives test generation systems access to programmer knowledge and the full power of the implementation language, which may improve the quality of the tests generated.

In particular, the second point brings us to the title of this paper. We have come to believe that it may be fruitful to think of learning-based test generation not so much as "generation" which implies a completely automatic process without human control but as *test programming* where a human test engineer/domain expert makes use of algorithmic techniques to ease the task of *programming* a highly effective method for generating tests. ABP systems (in conjunction with automated coverage tools) can in this light be seen simply as libraries, albeit more sophisticated and powerful than most, for helping programmers write programs to accomplish their tasks. It may be possible to (mostly) remove the human programmer from testing; we do not know if it is altogether wise.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75** (1987) 87–106
2. Cobleigh, J., Giannakopoulou, D., Păsăreanu, C.: Learning assumptions for compositional verification. In: Tools and Algorithms for the Construction and Analysis of Systems. (2003) 331–346

3. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. In: Tools and Algorithms for the Construction and Analysis of Systems. (2002) 357–370
4. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: FORTE. (1999) 225–240
5. Brady, B., Bryant, R.E., Seshia, S.A.: Learning conditional abstractions. In: Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD). (October 2011) 116–124
6. Gupta, A., Clarke, E.M.: Reconsidering CEGAR: Learning good abstractions without refinement. In: International Conference on Computer Design. (2005) 591–598
7. Chaki, S., Clarke, E.M., Groce, A., Strichman, O.: Predicate abstraction with minimum predicates. In: Advanced Research Working Conference on Correct Hardware Design and Verification Methods. (2003) 19–34
8. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Computer-Aided Verification. (2000) 154–169
9. McMinn, P.: Search-based software test data generation: A survey. Software testing, verification, and reliability **14** (2004) 105–156
10. Andrews, J., Li, F., Menzies, T.: Nighthawk: A two-level genetic-random unit test data generator. In: Automated Software Engineering. (2007) 144–153
11. Veanes, M., Roy, P., Campbell, C.: Online testing with reinforcement learning. In: Formal Approaches to Software Testing and Runtime Verification. (2006) 240–253
12. Groce, A.: Coverage rewarded: Test input generation via adaptation-based programming. In: IEEE/ACM International Conference on Automated Software Engineering. (2011) 380–383
13. Frankl, P.G., Weiss, S.N.: An experimental comparison of the effectiveness of branch testing and data flow testing. IEEE Transactions on Software Engineering **19** (1993) 774–787
14. Frankl, P.G., Iakounenko, O.: Further empirical studies of test effectiveness. In: International Symposium on Foundations of Software Engineering. (1998) 153–162
15. Lyu, M.R., Huang, Z., Sze, S.K.S., Cai, X.: An empirical study on testing and fault tolerance for software reliability engineering. In: International Symposium on Software Reliability Engineering. (2003) 119–126
16. Cai, X., Lyu, M.R.: The effect of code coverage on fault detection under different testing profiles. In: International Workshop on Advances in Model-Based Testing. (2005) 1–7
17. Sutton, R., Barto, A.: Reinforcement Learning: an Introduction. MIT Press (1998)
18. Groce, A., Holzmann, G., Joshi, R.: Randomized differential testing as a prelude to formal verification. In: International Conference on Software Engineering. (2007) 621–631
19. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Programming Language Design and Implementation. (2011) 283–294
20. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of haskell programs. In: International Conference on Functional Programming. (2000) 268–279
21. Bauer, T., Erwig, M., Fern, A., Pinto, J.: Adaptation-based programming in Java. In: ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. (2011) 81–90
22. Pinto, J., Fern, A., Bauer, T., Erwig, M.: Robust learning for adaptive programs by leveraging program structure. In: International Conference on Machine Learning and Applications. (2010) 943–948
23. Andre, D., Russel, S.: State abstraction for programmable reinforcement learning agents. In: National Conference on Artificial Intelligence. (2002)

24. Mahadevan, S.: Agent reward reinforcement learning: Foundations, algorithms, and empirical results. Machine Learning **22**(1) (1996) 159–195
25. Fern, A., Pinto, J., Bauer, T.: Adapatation-based programming library in Java. http://groups.engr.oregonstate.edu/abp/
26. Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in Games **4**(1) (2012) 1–43
27. Gelly, S., Silver, D.: Achieving master level play in 9× 9 computer go. In: Proceedings of the AAAI on Artificial Intelligence. (2008) 1537–1540
28. Andrews, J.H., Groce, A., Weston, M., Xu, R.G.: Random test run length and effectiveness. In: Automated Software Engineering. (2008) 19–28
29. Groce, A., Visser, W.: Model checking Java programs using structural heuristics. In: International Symposium on Software Testing and Analysis. (2002) 12–21
30. Groce, A., Visser, W.: Heuristics for model checking Java programs. Software Tools for Technology Transfer **6(4)** (2004) 260–276
31. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010)
32. Hudak, P.: Building Domain-Specific Embedded Languages. ACM Computing Surveys **28**(4es) (1996) 196–196
33. : Codecover - an open-source glass-box testing tool. http://codecover.org/
34. Holzmann, G., Joshi, R.: Model-driven software verification. In: SPIN Workshop on Model Checking of Software. (2004) 76–91
35. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
36. Groce, A., Holzmann, G., Joshi, R., Xu, R.G.: Putting flight software through the paces with testing, model checking, and constraint-solving. In: International Workshop on Constraints in Formal Verification. (2008) 1–15
37. Holzmann, G., Joshi, R., Groce, A.: Model driven code checking. Automated Software Engineering **15**(3-4) (2008) 283–297