Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning

Sameer Reddy
University of California, Berkeley, USA
sameer.reddy@berkeley.edu

Caroline Lemieux
University of California, Berkeley, USA
clemieux@cs.berkeley.edu

Rohan Padhye
University of California, Berkeley, USA
rohanpadhye@cs.berkeley.edu

Koushik Sen
University of California, Berkeley, USA
ksen@cs.berkeley.edu

ABSTRACT

Property-based testing is a popular approach for validating the logic of a program. An effective property-based test quickly generates many diverse valid test inputs and runs them through a parameterized test driver. However, when the test driver requires strict validity constraints on the inputs, completely random input generation fails to generate enough valid inputs. Existing approaches to solving this problem rely on whitebox or greybox information collected by instrumenting the input generator and/or test driver. However, collecting such information reduces the speed at which tests can be executed. In this paper, we propose and study a black-box approach for generating valid test inputs. We first formalize the problem of guiding random input generators towards producing a diverse set of valid inputs. This formalization highlights the role of a *guide* which governs the space of choices within a random input generator. We then propose a solution based on reinforcement learning (RL), using a tabular, on-policy RL approach to guide the generator. We evaluate this approach, *RLCheck*, against pure random input generation as well as a state-of-the-art greybox evolutionary algorithm, on four real-world benchmarks. We find that in the same time budget, *RLCheck* generates an order of magnitude more diverse valid inputs than the baselines.

ACM Reference Format:
Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380399

1 INTRODUCTION

Property-based testing is a powerful method for testing programs expecting highly-structured inputs. Popularized by QuickCheck [17], the method has grown thanks to implementations in many different languages [1, 2, 7, 8, 40], including prominent languages such as Python [3], JavaScript [4], and Java [27]. Property-based testing allows users to specify a test as $\forall x \in \mathcal{X} : P(x) \Rightarrow Q(x)$, where $\mathcal{X}$ is a domain of inputs and P, Q are arbitrary predicates. The precondition $P(x)$ is some sort of *validity constraint* on x. A property testing tool quickly generates many inputs $x \in \mathcal{X}$ and runs them through a test driver. If the tool generates an x for which $P(x)$ holds but $Q(x)$ does not hold, then the test fails.

Using a property-based testing tool requires two main steps. First, the user needs to write a *parameterized test driver*, i.e., the programmatic representation of $P(x) \Rightarrow Q(x)$. Second, the user needs to specify the *generator* for $x \in \mathcal{X}$. A generator for $\mathcal{X}$ is a non-deterministic program returning inputs $x \in \mathcal{X}$. Testing frameworks typically provide generators for standard types (e.g. primitive types or a standard collections of primitives). If $\mathcal{X}$ is a user-defined type, the user may need to write their own generator.

For property-based testing to be effective, the generator must produce a *diverse* set of inputs $x \in \mathcal{X}$ satisfying the validity constraint $P(x)$. This is a central conflict in property-based testing. On the one hand, a simple generator is easier for the user to write, but not necessarily effective. On the other hand, a generator that produces diverse valid inputs is good for testing, but very tedious to write. Further, generators specialized to a particular validity function $P(x)$ cannot be reused to test other properties on $\mathcal{X}$ with different validity constraints, say $P'(x)$. We thus want to solve the following problem: given a generator G of inputs $x \in \mathcal{X}$ and a validity function $P(x)$, *automatically* guide the generator to produce a variety of inputs x satisfying $P(x)$.

One way to solve this problem is to do whitebox analysis [14, 22–24] of the generator and/or the implementations of $P(x)$ and $Q(x)$. A constraint solver can be used to generate inputs $x \in \mathcal{X}$ that are guaranteed to satisfy $P(x)$, which also exercise different code paths within the implementation of $Q(x)$ [42]. Another set of approaches, adopted in recent greybox fuzzing-inspired work [31, 39], is to collect code coverage during test execution. This information can be used in an evolutionary algorithm that can generate inputs that are likely satisfy $P(x)$, while optimizing to increase code coverage through $Q(x)$. Both sets of approaches require instrumenting the program under test; thus, they are limited in terms of performance. Purely whitebox approaches have trouble scaling to complex validity functions due to the path explosion problem [16], and even greybox fuzzing techniques require instrumentation that can lead to slowdowns, thereby reducing the rate at which tests can be executed. This is in conflict with the QuickCheck approach, in which properties can be *quickly* validated without instrumenting the test

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05.
https://doi.org/10.1145/3377811.3380399

program. In this paper, we address this gap by investigating a *black-box approach* for guiding a generator G to produce a large number of diverse valid inputs in a *very short amount of time*.

In this paper, we first formalize the problem of guiding the random choices made by a generator for effective property testing as the *diversifying guidance problem*. Second, we notice that the diversifying guidance problem is similar to problems solved by reinforcement learning: given a sequence of prior choices (state), what is the next choice (action) that the generator should make, in order to maximize the probability of generating a new x satisfying $P(x)$ (get high reward)? We thus explore whether reinforcement learning can solve the diversifying guidance problem. We propose an on-policy table-based approach which adapts its choices on-the-fly during testing time. We call our technique *RLCheck*.

We implement *RLCheck* in Java and evaluate it to state-of-the-art approaches Zest [39] and QuickCheck. *RLCheck*'s implementation is open-source and available at: https://github.com/sameerreddy13/rlcheck. We compare the techniques on four real-world benchmarks used in the original evaluation of Zest: two benchmarks that operate on schema-compliant XML inputs, and two that operate on valid JavaScript programs. We find that *RLCheck* generates 1.4×-40× more diverse valid inputs than state-of-the-art within the same tight time budget. All methods are competitive in terms of valid branch coverage, but the simple RL algorithm we explore in *RLCheck* may be biased to certain parts of the valid input space. We also find that a basic addition of greybox feedback to *RLCheck* does not produce improvements that are worth the instrumentation slowdown.

In summary, we make the following contributions:

- We formalize the problem of making a generator produce many valid inputs as the *diversifying guidance problem*.
- We propose a reinforcement learning approach to solve the diversifying guidance problem, named *RLCheck*.
- We evaluate *RLCheck* against the state-of-the art Zest tool [39] on its valid test-input generation ability.

2 A MOTIVATING EXAMPLE

Property-based testing tools [9, 17, 19, 21, 22, 40, 47] allow users to quickly test a program by running it with many inputs. Originally introduced for Haskell [17], property-based testing has since been ported to many other programming languages [1–4, 7, 8, 27, 40]. Using a property-based testing tool involves two main tasks.

First, the user must write a *parameterized test driver* which takes input x of some type $\mathcal{X}$ and runs test code checking a property $P(x) \Rightarrow Q(x)$. We say x is *valid* if it satisfies $P(x)$.

For example, in Figure 1, the test driver `test_insert` (Line 15) is given a binary tree `tree` and an integer `to_add` as input. If `tree` is a binary search tree (Line 16), the driver inserts `to_add` into `tree` (Line 17) and asserts that `tree` is still a binary search tree after the insert (Line 18). The **`assume`** at Line 16 terminates the test silently if `tree` is not a binary search tree. The **`assert`** at Line 18 is violated if `tree` is not a binary search tree after the insertion. Thus, the test driver implements $P(x) \Rightarrow Q(x)$ for the validity constraint $P(x)$ ="x is a binary search tree" and the post-condition $Q(x)$ ="after inserting `to_add`, x is a binary search tree"—by raising an assertion failure when $P(x) \Rightarrow Q(x)$ is falsified. In this example, the predicate $Q(x)$ is specified explicitly via assertions. $Q(x)$ can also be implicitly

```
1  from generators import generate_int
2
3  def generate_tree(depth=0):
4      value = random.Select([0, 1, ..., 10])
5      tree = BinaryTree(value)
6      if depth < MAX_DEPTH and
7              random.Select([True, False]):
8          tree.left = generate_tree(depth+1)
9      if depth < MAX_DEPTH and
10             random.Select([True, False]):
11         tree.right = generate_tree(depth+1)
12     return tree
13
14 @given(tree = generate_tree, to_add = generate_int)
15 def test_insert(tree, to_add):
16     assume(is_BST(tree))
17     BST_insert(tree, to_add)
18     assert(is_BST(tree))
```

Figure 1: Pseudocode for a property-based test. `generate_tree` generates a random binary tree, and `test_insert` tests whether inserting a given integer into a given tree preserves the binary search tree property. `random.Select(D)` returns a random value from `D`.

defined, e.g. $Q(x)$ may be the property that the program does not crash with a segfault when executed on x.

Second, the user must specify how to generate random inputs for the test driver. This must be done by writing or specifying a *generator*, a non-deterministic function returns a random input of a given type in each of its executions. Property-based testing frameworks typically provide generators for basic types such as primitive types and predefined containers of primitives (e.g. `generate_int` in Figure 1). If the test function takes a user-defined data structure, such as the `tree` in Figure 1, Line 15, the user writes their own generator. For many types, writing a basic generator is fairly straightforward. In Figure 1, `generate_tree` generates a random binary tree by (1) choosing a value for the root node (Line 4), (2) choosing whether or not to add a left child (Line 7) and recursively calling `generate_tree` (Line 8), and (3) choosing whether or not to add a right child (Line 10) and recursively calling `generate_tree` (Line 11). We have deliberately kept this generator simple in order to have a small motivating example.

The user can now run `test_insert` on many different trees to try and validate $P(x) \Rightarrow Q(x)$: the **`assume`** in Line 16 effectively filters out invalid (non-BST) inputs. Unfortunately, this rejection sampling is not an effective strategy if $P(x)$ is too strict. If the generator has no knowledge of $P(x)$, it will, first of all, very rarely generate valid inputs. So, in a fixed time budget, very few valid inputs will be generated. The second issue is that the generator may not generate very diverse valid inputs. That is, the only valid inputs the generator has a non-negligible probability of generating may be *very small* valid inputs; these will not exercise a variety of behaviors in the code under test. For example, out of 1000 generated binary trees, the generator in Figure 1 only generates 20 binary search trees of size ≥ 3, and only one binary search tree of size 4 and 5, respectively. Overall, the generator has very low probability

of generating complex valid inputs, which greatly decreases the efficacy of the property-based test.

Fundamentally, the input generated by a generator is governed by the choices taken at various *choice points*. For example, in Figure 1, at Line 4, the generator makes a *choice* of which integer value to put in the current node, and it chooses to make a left or right child at Lines 7 and 10, respectively. Depending on the prior sequence of choices taken by the generator, only a subset of the possible choices at a particular choice point may result in a valid input. For example, if $P(x)$ is the binary search tree invariant, when generating the right child of a node with value n, the only values for the child node that can result in a valid BST are those greater than n. While narrowing the choice space in this manner is straightforward for BSTs, manually encoding these restrictions is tedious and error-prone for complex real-world validity functions.

Overall, we see that the problem of guiding G to produce many valid inputs boils down to the problem of narrowing the choice space at each choice point in the generator. We call this the *diversifying guidance problem*. We formalize this problem in Section 3 and propose a solution based on reinforcement learning in Section 4.

3 PROBLEM DEFINITION

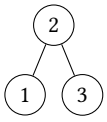
In property-based testing, a generator G is a non-deterministic program returning elements of a given space $\mathcal{X}$. For example, in Figure 1, $\mathcal{X}$ is the set of binary trees of depth up to `MAX_DEPTH` with nodes having integer values between 0–10, inclusive.

In particular, a generator G's non-determinism is entirely controlled by the values at returned at different *choice points* in the generator. A choice point p is a tuple (ℓ, C) where $\ell \in \mathbb{L}$ is a program location and $C \subseteq \mathcal{C}$ is a finite domain of choices. For example, there are three choice points in the generator in Figure 1:

- (Line 4, `[0, 1, ..., 10]`): the choice of node value;
- (Line 7, `[True, False]`): whether to generate a left child; and
- (Line 10, `[True, False]`): whether to generate a right child.

During execution, each time the generator reaches a choice point (ℓ, C), it makes a choice $c \in C$. Every execution of the generator, and thus, every value produced by the generator, corresponds to a sequence of choices made at these choice points, say $c_1, c_2, \ldots, c_n$.

For example, the execution through `generate_BST` in Figure 1 which produces the tree:

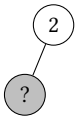


corresponds to the following sequence of choices $c_1, c_2, \ldots c_9$:

Choice Index	Choice Taken	Choice Point
c_1	`2`	(Line 4, `[0, 1, ..., 10]`)
c_2	`True`	(Line 7, `[True, False]`)
c_3	`1`	(Line 4, `[0, 1, ..., 10]`)
c_4	`False`	(Line 7, `[True, False]`)
c_5	`False`	(Line 10, `[True, False]`)
c_6	`True`	(Line 10, `[True, False]`)
c_7	`3`	(Line 4, `[0, 1, ..., 10]`)
c_8	`False`	(Line 7, `[True, False]`)
c_9	`False`	(Line 10, `[True, False]`)

As the generator executes, each time it reaches a choice point $p = (\ell, C)$, it will have already made some choices $c_1, c_2, \ldots c_k$. Traditional QuickCheck generators, like the one in Figure 1, will simply choose a random $c \in C$ at choice point p regardless of the prefix of choices $c_1, c_2, \ldots c_k$.

Going back to our running example, suppose the generator has reached the choice point choosing the value of the left child of 2, i.e. choosing what to put in the `?`:



That is, the generator has made the choices $[c_1 = 2, c_2 = \texttt{True}]$, and must now choose a value from 0–10 at the choice point in Line 4. The generator is equally likely to pick any number from this range. Since only 2 of the 11 numbers from 0–10 are smaller than 2, it has at most an 18% chance of producing a valid BST.

To increase the probability of generating valid inputs, the choice at this point should be made not randomly, but according to a *guide*. In particular, according to a guide which restricts the choice space to only those choices which will result in a binary search tree. First, we formalize the concept making choices according to a guide.

Definition 3.1 (Following a Guide)**.** We say that a generator G *follows* a guide $\gamma : \mathcal{C}^* \times P \times \mathbb{N} \to C$ if: during its t^{th} execution, given a sequence of past choices $\sigma = c_1, c_2, \ldots, c_k$, and the current choice point $p = (\ell, C)$, the generator G makes the choice $\gamma(\sigma, p, t)$.

Suppose we have a validity function $v : \mathcal{X} \to \{\texttt{True}, \texttt{False}\}$ which maps elements, output by the generator, to their validity. For example, `is_BST` is a validity function for the generator in Figure 1. The *validity guidance problem* is the problem of finding a guide that leads the generator to produce valid elements of $\mathcal{X}$:

Definition 3.2 (Validity Guidance Problem)**.** Let G be a generator producing elements in space $\mathcal{X}$. Let $v : \mathcal{X} \to \{\texttt{True}, \texttt{False}\}$ be a validity function. The *validity guidance problem* is the problem of creating a guide γ such that:

if G follows γ, then $v(x) = \texttt{True}$ for any $x \in \mathcal{X}$ generated by G.

Note that a solution to the validity guide problem is not necessarily useful for testing. In particular, the guide γ could simply hard-code a particular sequence of choices through the generator which results in a valid element $x \in \mathcal{X}$. Instead, we want to generate valid inputs with diverse *characteristics*. For example, we may want to generate unique valid inputs, or valid inputs of different lengths, or valid inputs that exercise different execution paths in the test program. We use the notation $\xi(x)$ to denote an input's characteristic of interest, such as identity, length, or code coverage.

Definition 3.3 (diversifying guidance problem)**.** Let G be a generator producing elements in space $\mathcal{X}$. Let $v : \mathcal{X} \to \{\texttt{True}, \texttt{False}\}$ be a validity function and ξ be a characteristic function. The *diversifying guidance problem* is the problem of creating a guide γ such that:

if G follows γ and $X_T \subseteq \mathcal{X}$ is the set of inputs generated by G after T executions, $|\{\xi(x) : x \in X_T \wedge v(x)\}|$ is maximized.

If ξ is the identity function, then a solution to the diversifying guidance problem is a guide which maximizes the number of unique valid inputs generated.

4 METHOD

In this section we describe our proposed solution to diversifying guidance problem. In particular, our proposed guide uses reinforcement learning to makes its choices. We begin with a basic background on Monte Carlo Control [46], the table-based reinforcement learning method used in this paper. We then describe how it can be used to solve the diversifying guidance problem.

4.1 Reinforcement Learning

We begin by defining a version of the problem solved by reinforcement learning which is relevant to our task at hand. We use a slightly nontraditional notation for consistency with the previous and next sections. What we call *choices* are typically called *actions*, and what we call a *learner* is typically called an *agent*.

We assume an learner in some environment. The learner can perceive the *state* s of the environment, where s is in some set of states $\mathcal{S}$. At the first point in time, the learner is at an initial state $s_0 \in \mathcal{S}$. At each point in time, the learner can make a choice $c \in C$ which will bring it to some new state $s' \in \mathcal{S}$. Eventually, the agent gets into some terminal state $s_{term} \in \mathcal{S}$, indicating the end of an *episode*. An episode is the sequence of *(state, choice)* pairs made from the beginning of time up to the terminal state, i.e.:

$$e = (s_0, c_0), (s_1, c_1), \ldots (s_T, c_T)$$

Where the choice c_T in state s_T brings the learner to the terminal state s_{term}. Finally, we assume we are given a reward r for a given episode e. A larger reward is better.

The problem to solve is the following. Given a state space $\mathcal{S}$, choices C, and reward r, find a policy π which maximizes the expected reward to the learner. That is, find a π such that if the learner, at each state $s \in \mathcal{S}$, makes the choice $c = \pi(s)$, then the expected reward $\mathbb{E}_{\pi,e}[r]$ from the resulting episode e is maximized.

4.1.1 Monte Carlo Control. One approach to solving the policy-learning problem above is by on-policy *Monte Carlo Control* [46]. The technique is *on-policy* because the policy the learner is optimizing is the same one it is using to control its actions. Thus, a Monte Carlo Control learner L defines both a policy π, where $\pi(s)$ outputs a choice c for the given state s, as well as an update procedure that improves π after each episode.

Algorithm 1 shows pseudocode for a Monte Carlo Control (MCC) learner L. In the algorithm, we subscript the choice space, state space, and Q and *counts* with L to emphasize these are independent for each MCC learner. We will drop the subscript L when talking about a single learner. The basic idea is as follows.

We are trying to learn a policy π for state space $\mathcal{S}$ and choices C. The policy is ε-greedy: with probability ε it makes random choices (Line 7), otherwise it makes the choices that will maximize the value function, Q (Line 9).

The value function $Q[s, c]$ models the expected reward at the end of the episode from the choice c in state s. It is initialized to 0 for each (s, c) pair (Line 4), so the first episode follows a totally random policy. $Q[s, c]$ is exactly the *average rewards* seen for each episode e containing (s, c). Thus, at the end of each episode e, for each $(s, c) \in e$ (Line 15), the running average for the rewards observed with action (s, c) is updated to include the new reward r (Line 16).

Algorithm 1 A Monte Carlo Control learner L. Implements a policy π_L and an update function $\textsc{update}_L$ which updates π_L towards the optimal policy after each episode.

Input: choice space C_L, state space $\mathcal{S}_L$, and ε_L

1: $e_L \leftarrow []$ ▷ initialize episode
2: **for** $(s, c) \in \mathcal{S}_L \times C_L$ **do**
3: $counts_L[s, c] \leftarrow 0$
4: $Q_L[s, c] \leftarrow 0$
5: **procedure** π(**state** s)
6: **if** uniformRandom( ) $< \varepsilon$ **then**
7: $c \leftarrow$ random(C)
8: **else**
9: $c \leftarrow \arg\max_{c \in C_L} Q_L[s, c]$ ▷ break ties arbitrarily
10: $e_L \leftarrow$append($e_L, (s, c)$)
11: **return** *choice*
12: **procedure** update(**reward** r)
13: $T \leftarrow$len(e_L)
14: **for** $0 \leq t < T$ **do**
15: $s, c \leftarrow e_L[t]$
16: $Q_L[s, c] \leftarrow \frac{r + Q_L[s,c] \cdot (counts_L[s,c])}{counts_L[s,c]+1}$ ▷ update avg. reward
17: $counts_L[s, c] \leftarrow counts_L[s, c] + 1$
$e_L \leftarrow []$

If the reward function producing r is *stationary* (i.e., fixed), then it can be shown that this update procedure always improves the policy. That is, if π is the original policy, and π' is the policy after the update, the expected reward from a learner following π' is greater than or equal to the expected reward from a learner following π. Sutton and Barto [46] provide a proof.

We draw attention to some specifics of our implementation, that diverge from what may appear in textbooks.

4.1.2 Algorithmic Changes. Firstly, we update an episode with a single reward r which is distributed to all state action pairs. This is because, as will be seen in later sections, we only observe rewards at the end of an episode i.e there are no intermediate rewards provided in our method. Secondly we do not use a discount factor on the reward r. This is because the sequence of choices in an input generation, do not lend themselves to a natural absolute ordering. We cannot assume later decisions are more important than earlier ones, which the discount factor implicitly does.

4.2 *RLCheck*: MCC with Diversity Reward

We now return to our problem space of generating inputs with a generator G. Notice that the *guides* we defined in Definition 3.1 have a similar function to the learners in Section 4.1: given some state (σ, p, t) make a choice c.

This leads to the natural idea of implementing a guide as an MCC learner, rewarding the learner with some $r(x)$ after the generator produces input x. However, note that for the guide, at each choice point $p = (\ell, C)$, only a subset of choices $C \subseteq C$ can be taken. Further, each choice point has a unique task: for example, choosing whether to generate a left child (Figure 3, Line 9) or a right child (Figure 3, Line 13). Thus, it is natural to define a separate learner

L_p for each choice point p, and call UPDATE_{L_p} once for each learner after every execution of the generator.

Finally, in Section 3, we defined a guide using a sequence $\sigma \in C^*$ to influence its actions, while in Section 4.1, we assumed a finite set of states $\mathcal{S}$. Thus, we need a state abstraction function:

Definition 4.1 (State Abstraction Function)**.** A *state abstraction function* $A : C^* \to \mathcal{S}$ for a generator G is a deterministic function mapping an arbitrary-length choice sequence σ to a finite state space $\mathcal{S}$. A can rely on G to retrieve, for any $c_i \in \sigma$, the choice point p at which c_i was made.

We will return to the state abstraction function in Section 4.3. We can now define a Monte Carlo Control Guide.

Definition 4.2 (Monte Carlo Control Guide)**.** Assume a generator G producing inputs in $\mathcal{X}$, a state abstraction function A, and a reward function $r : \mathcal{X} \to \mathbb{R}$. A Monte Carlo Control Guide γ consists of a set of Monte Carlo control *learners*, $\{L_p\}$. Each learner L_p is associated with a choice point $p = (\ell, C)$ in G.

Let $\pi^{(t)}_{L_p}$ be L_p's policy after $t-1$ calls to UPDATE_{L_p} (ref. Algorithm 1). Then γ is:

$$\gamma(\sigma, p, t) = \pi^{(t)}_{L_p}(A(\sigma)).$$

Finally, after G produces an input x, the guide γ calls $\text{UPDATE}_{L_p}(r(x))$ for each of its learners L_p.

Now, to use a Monte Carlo Control guide (MCC guide) to solve the diversifying guidance problem, only (1) the state abstraction function A (ref. Section 4.3) and (2) the reward function r need to be specified. We construct a reward function as follows.

Let v be the validity function and ξ the characteristic function of interest. If X be the set of inputs previously generated by G, then let $\Xi = \{\xi(x') : x' \in X\}$ be the set of characteristics of all the previously generated inputs. Then the reward function r is:

$$r(x) = \begin{cases} R_{unique} \text{ if } v(x) \wedge \xi(x) \notin \Xi \\ R_{valid} \text{ if } v(x) \wedge \xi(x) \in \Xi \\ R_{invalid} \text{ if } \neg v(x) \end{cases} \quad (1)$$

Our technique, ***RLCheck***, is thus: make a generator G follow an MCC Guide with the reward function r above.

Note that this reward function is *nonstationary*, that is, it is not fixed across time. If $X = \emptyset$, then generating any $x \in \mathcal{X}$ such that $v(x)$ holds will result in the reward R_{unique}; re-generating the same x in the next step will only result in the reward R_{valid}. This means the assumptions underlying the classic proof of policy improvement do not hold [46]. Thus, *RLCheck*'s guide is not guaranteed to improve to an optimal policy. Instead, it practices a form of online learning, adjusting its policy over time.

4.3 State Abstraction

A key element in enabling MCC to solve the diversifying guidance problem is the state abstraction function (Definition 4.1), which determines the current state given a sequence of past choices. The choice of A impacts the ability of the MCC guide to learn an effective policy. On one extreme, if A collapses all sequences into the same abstract state (e.g., $A(\sigma) = 0$), then a learner L_p essentially attempts to find a single best choice $c \in C_{L_p}$ for choice point p, regardless

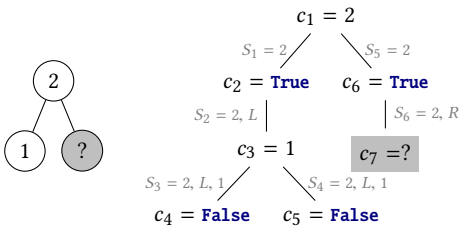


Figure 2: A partially-generated binary tree (left) and its corresponding choice sequence arranged by influence (right).

of state. On the other extreme, if A is the identity function (i.e., $A(\sigma) = \sigma$), then the state space is infinite; so for every previously unseen sequence of choices σ, the learner's policy is random.

The ideal A is the abstraction function that maximizes expected reward. However, computing such an A is not tractable, since it requires inverting an arbitrary validity function $v(x)$. Instead, we apply the following heuristic: in many input generators, a good representation for the state S_n after making the n^{th} choice c_n is some function of a past subsequence of choices that *influence* the choice c_n. The meaning of influence depends on the type of input being generated and the nature of the validity function.

For example, Figure 2 shows a partially generated binary tree on the left. On the right, we show the choices made in the binary-tree generator (ref. Fig. 1) leading to this partial tree ($c_1 = 2, c_2 =$ `True`, $c_3 = 1, c_4 =$ `False`, $c_5 =$ `False`, $c_6 =$ `True`), arranged by *influence*, where a choice in the construction of a child node is *influenced* by choices constructing its parent node.

With this influence heuristic, the best value for the next choice c_7, which determines the value assigned to the right child, should depend on the choice c_1, which decided that the root node had value 2, as well as the choice c_6, which made the decision to insert a right child. The best value for this choice c_7 does not necessarily depend on choices c_2–c_5, which were involved in the creation of the left sub-tree. Therefore, the state S_6, in which the choice c_7 is to be made, can be represented as a sequence $[f_v(c_1), f_r(c_6)]$. Here, f_v is a function associated c_1's choice point (the node-value choice point at Line 4 of Fig. 1) and f_r is a function associated with c_6's choice point (the right-child choice point at Line 10 of Fig. 1). In Figure 2, the state S_6 after applying these functions is `[2, R]`; we will define the functions f_v and f_r for this figure later in this section.

An additional consideration when representing state as a sequence derived from past choices is that such sequences can become very long. We need to restrict the state space to being finite. Again, a reasonable heuristic is to use a trimmed representation of the sequence, which incorporates information from up to the last w choices that influence the current choice. w is a fixed integer that determines the size of a sliding *window*.

We can build a state abstraction function that follows these considerations in the following manner. First, build a choice abstraction function f_p for each choice point p, which maps each c to an abstract choice. Then, for $\sigma = c_1, c_2, \ldots, c_n$, build $S_n = A(\sigma)$ so that:

$$S_n = \begin{cases} \emptyset \text{ if } \sigma = \emptyset \\ tail_w(S_k :: f_p(c_n)) \text{ for some } k < n \text{ otherwise,} \end{cases}$$

```
1  def concat_tail(state, value):
2    return (state + [value])[-WINDOW_SIZE:]
3
4  def gen_tree(state, depth=0):
5    value = guide.Select([0, ..., 10], state, idx=1)
6    state = concat_tail(state, value)
7    tree = BinaryTree(value)
8    if depth < MAX_DEPTH and \
9          guide.Select([True, False], state, idx=2):
10     left_state = concat_tail(state, "L")
11     tree.left = gen_tree(left_state,  depth+1)
12   if depth < MAX_DEPTH and \
13         guide.Select([True, False], state, idx=3):
14     right_state = concat_tail(state, "R")
15     tree.right = gen_tree(right_state, depth+1)
16   return tree
```

Figure 3: Pseudo-code for a binary tree generator which follows guide and builds a tree-based state abstraction.

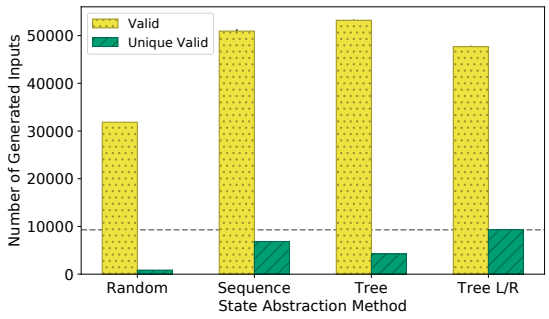


Figure 4: Number of (unique) valid inputs generated, by state abstraction. "Random" is a no-RL baseline.

where :: is the concatenation operator and $tail_w(s)$ takes the last w elements of s. Assume c_n was taken at choice point p.

We can build both very basic and very complex state abstractions in this manner.

For example, we can get $A(\sigma) = c_{n-w+1}, \ldots, c_{n-1}, c_n$ by taking $f_p = id$ for all and choosing $k = n - 1$ always. This would be a simple sliding window of the last w choices.

The states S_1-S_6 that annotate the edges in Figure 2 are derived using the choice point abstraction functions $f_v(c) = c$ for the value choice point, $f_r(c) = R$ for the right child choice point, and $f_l(x) = L$ for the left child choice point. The k is chosen as $k =$"largest $k < n$ which is a choice from the parent node". While programatically deriving this k from a choice sequence σ is tedious, it is quite easy to do inline in the generator. The generator Figure 3 shows a modified version of the generator from Figure 1, which updates an explicit state value at each to compute exactly this state abstraction function (Lines 6, 9, 13); it also uses guides to select arbitrary values (Lines 5, 9, 13).

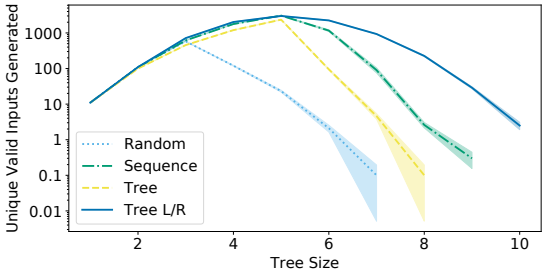


Figure 5: Distribution of unique valid tree sizes, by state abstraction. "Random" is a no-RL baseline.

4.3.1 Case study. We evaluate the effect the state abstraction function has on the ability of *RLCheck* to produce unique valid inputs for the BST example. We evaluate three state abstraction functions:

- *Sequence*, the sliding window abstraction which retains choices from the sibling nodes, i.e. $A(\sigma) = c_{n-w+1}, \ldots, c_{n-1}, c_n$.
- *Tree L/R*, the abstraction function illustrated in Figure 2 and implemented in Figure 3.
- *Tree*, which chooses k like Tree L/R but has $f_p = id$ for all choice points, and thus produces the same state for the left and right subtree of a node.

For example, taking $w = 4$ and the choices to be abstracted $c_1, \ldots, c_6$ from Figure 2: *Sequence* will give `[1, False, False, True]`, *Tree* state will give `[2, True]`, and *Tree L/R* will give `[2, "R"]`.

We evaluate each of these abstraction techniques for generating BSTs with maximum depth 4 (i.e., 4 links), with $\varepsilon = 0.25$ and rewards (Eq. 1) $R_{invalid} = -1, R_{valid} = 0$, and $R_{unique} = 20$. We set $w = 4$ for the abstraction function: since there are at least two elements in the state for each parent node, this means the learners cannot simply memorize the decisions for the full depth of the tree.

Results. Figures 4 and 5 show the results for our experiments. In each experiment we let each technique generate 100,000 trees. The results show the averages and standard errors over 10 trials. We compare to a baseline, *Random*, which just runs the generator from Figure 1. Figure 4 illustrates that no matter the state abstraction function chosen, *RLCheck* generates many more valid and unique valid inputs than the random baseline; *Tree L/R* generates 10× more unique valid inputs than random. Within the abstraction techniques, *Tree* generates the fewest unique valid inputs. *Sequence* appears to be better able to distinguish whether it is generating a left or right child than *Tree*, probably because the *Tree* state is identical for the left and right child choice points.

Tree L/R generates the fewest valid inputs, but the most unique valid inputs, 36% more than *Sequence*. These unique valid inputs are also more complex those generated with other state abstractions. Figure 5 shows, for each technique, the average number of unique valid trees generated of each size. Note the log scale. The tree size is the number of nodes in the tree. We see that *Tree L/R* is consistently able to generate orders of magnitude more trees of sizes > 5 than the other techniques. Since we reward uniqueness, the *RLCheck* is encouraged to generate larger trees as it exhausts the space of smaller trees. These results suggest that *Tree L/R* has

enough information to generate valid trees, and then combine these successes into more unique valid trees.

Overall, we see that even with a naïve state abstraction function, *RLCheck* generates nearly an order of magnitude more unique valid inputs than the random baseline. However, a well-constructed influence-style state abstraction yields more diverse valid inputs.

5 EVALUATION

In this section we evaluate how *RLCheck*, our MCC-based solution to the diversifying guidance problem, performs. In particular, we focus on the following research questions:

- RQ1 Does *RLCheck* quickly find many diverse valid inputs for real-world benchmarks compared to state-of-the-art?
- RQ2 Does *RLCheck* find valid inputs covering many different behaviors for real-world benchmarks?
- RQ3 Does adding coverage feedback improve the ability of *RLCheck* to generate diverse valid inputs for real-world benchmarks?

Implementation. To answer these research questions, we implemented Algorithm 1 in Java, and *RLCheck* on top of the open-source JQF [38] platform. JQF provides a mechanism for customizing input generation for QuickCheck-style property tests.

Baseline Techniques. We compare *RLCheck* to two different methods: (1) junit-quickcheck [27], or simply QuickCheck, the baseline generator-based testing technique which calls the generator with a randomized guide; and (2) Zest [39], also built on top of JQF, which uses an evolutionary algorithm based on coverage and validity feedback to "guide" input generators. Unlike *RLCheck* and QuickCheck, Zest is a *greybox* technique: it relies on program instrumentation to get code coverage from each test execution.

Benchmarks. We compare the techniques on four real-world Java benchmarks used in the original evaluation of Zest [39]: Apache Ant, Apache Maven, Google Closure Compiler, and Mozilla Rhino. These benchmarks rely on two generators: Ant and Maven use an XML generator, whereas Closure and Rhino use a generator for JavaScript ASTs. The validity functions for each of these four benchmarks is distinct: Ant expects a valid `build.xml` configuration, Maven expects a valid `pom.xml` configuration, the Closure expects an ES6-compliant JavaScript program that can be optimized, and Rhino expects a JavaScript program that can be statically translated to Java bytecode. Overall, Ant has the strictest validity function and Rhino has the least strict validity function.

Design Choices. In our main evaluation, we simply use identity as the characteristic function ξ to which inputs get R_{unique}. Thus, *RLCheck* simply tries to maximize the number of unique valid inputs. This allows us to run *RLCheck* at full speed without instrumentation, and generate more inputs in a fixed time budget. In Section 5.3 we compare this choice to a greybox version of *RLCheck*, where $\xi(x)$ takes into account the branch coverage achieved by input x.

We instantiate our reward function (Eq. 1) with $R_{unique} = 20$, $R_{valid} = 0$ and $R_{invalid} = -1$. This incentivizes *RLCheck* to prioritize exploration of new unique valid inputs, while penalizing strategies that lead to producing invalid inputs. Additionally, we set $\varepsilon = 0.25$ in Algorithm 1, which allows *RLCheck* to explore at random with reasonably high probability.

We first modified the base generators provided by JQF for XML and JavaScript to transform choice points with infinite domains to finite domains. These are the generators we use for evaluation of Zest and QuickCheck. We then built guide-based generators with the same choice points as these base generators. For the guide-based generators, we built the state abstraction inline, like it is built in Figure 3. For each benchmark, the state abstraction function is similar to that in Figure 3 as it maintains abstractions of the parent choices. We set $w = 5$ for the state window size. The generator code is available at https://github.com/sameerreddy13/rlcheck.

Experiments. We sought to answer our research questions in a property-based testing context, where we expect to be able to run the test generator for a short amount of time. Thus, we chose 5 minutes as a timeout. To account for variability in the results, we ran 10 trials for each technique. The experiments in Section 5.1 and 5.2 were run on GCP Compute Engine using a single VM instance with 8vCPUs and 30 GB RAM. The experiments in Section 5.3 were run on a machine with 16GB RAM and an AMD Ryzen 7 1700 CPU.

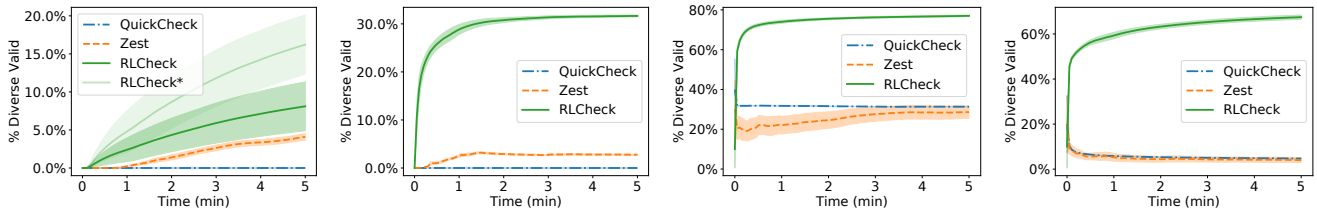
5.1 Generating Diverse Valid Inputs

To answer RQ1, we need to measure whether *RLCheck* generates a higher number of unique, valid inputs compared to our baselines. On these large-scale benchmarks, where the test driver does non-trivial work, simple uniqueness, at the byte or string level, is not the most relevant measure of input diversity.

What we are interested in is inputs with diverse coverage. So, we measure inputs with different *traces*, a commonly-used metric for input coverage diversity in the fuzz testing literature [50] (sometimes these traces are called "paths", but this is a misnomer). The trace of an input x is a set of pairs (b, c) where b is a branch and c is the number of times that branches is executed by x, bucketed to base-2 orders of magnitude. Let $\xi(x)$ give the trace of of x. If x_1 takes the path A, B, A, then $\xi(x_1) = \{(A, 2), (B, 1)\}$. If x_2 takes the path A, A, A, B, then A is hit the same base-2 order-of-magnitude times, so $\xi(x_2) = \{(A, 2), (B, 1)\}$. We call valid inputs with different traces *diverse valid* inputs.

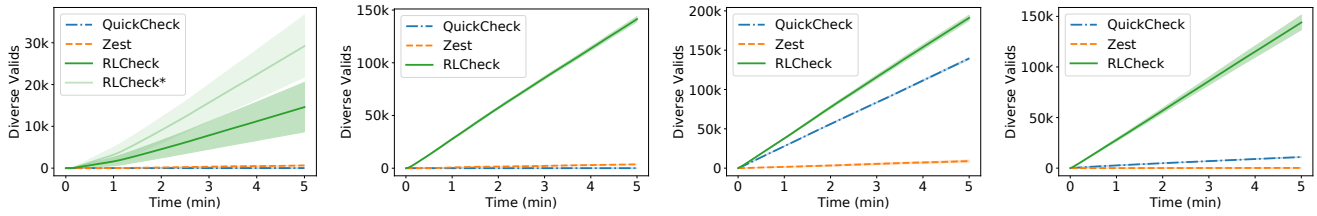
The results are shown in Figures 6 and 7. Figure 6 shows, at each time, the *percentage* of all generated inputs that are diverse valid inputs. For techniques that are only able to generate a fixed number of diverse valid inputs, this percentage would steadily decrease over time. In Figures 6c and 6d, we see an abrupt decrease at the beginning of fuzzing for Zest and QuickCheck, and for Closure we see a continuing decrease in the percentage over time for these techniques. In Figures 6b, 6c, and 6d see that *RLCheck* quickly converges to a high percentage of diverse valid inputs being generated, and maintains this until timeout.

RLCheck also generates a large *quantity* of diverse valid inputs. Figure 7 shows the total number of diverse valid inputs generated by each technique: we see that *RLCheck* generates multiple order of magnitude more diverse valid inputs compared to our baselines. The exception is on Rhino (Figure 7), *RLCheck* only has a 1.4× increase over QuickCheck. Rhino's validity function is relatively easy to satisfy: most JavaScript ASTs are considered valid inputs for translation; therefore, speed is the main factor in generating valid inputs for this benchmark. Consequently, the blackbox techniques



(a) Ant (*: at least 1 valid) (b) Maven (c) Rhino (d) Closure

Figure 6: Percent of total generated inputs which are diverse valids (i.e. have different traces). Higher is better.



(a) Ant (*: at least 1 valid) (b) Maven (c) Rhino (d) Closure

Figure 7: Number of diverse valid inputs (i.e. inputs with different traces) generated by each technique. Higher is better.

RLCheck and QuickCheck outperform the instrumentation-based Zest technique on the Rhino benchmark.

On both metrics, the increase in Ant is less pronounced, and very variable. The variation in percentage for Ant is quite wide because it is hard to get a first valid input for *RLCheck* (and QuickCheck), and in some cases *RLCheck* did not get this within the five-minute time budget. For an understanding of the effect on the results, *RLCheck** shows the results for only those runs that find at least one valid input. The mean value for *RLCheck** is much higher, but the high standard errors remain due to the fact that these runs find the first valid input being at different times. For such extremely strict validity functions, *RLCheck* has difficulty finding a first valid input compared to coverage-guided techniques. This is a limitation of *RLCheck*: a good policy can only be found after some valid inputs have been discovered.

For completeness, we also ran longer experiments of 1 hour, to see if Zest or QuickCheck would catch up to *RLCheck*. In 1 hour, *RLCheck* generates between 5-15× more diverse valid inputs than Zest on all benchmarks and outperforms QuickCheck on all benchmarks. Furthermore, *RLCheck* continues to generate a higher percentage of generated diverse valid inputs after one hour. In particular, the large improvements that are seen in Figures 6 are all maintained at roughly the same rate except for Rhino. In the case of Rhino, Zest improves its percentage of diverse valid inputs from 40% to 67% after one hour, while *RLCheck* continues to generate 78% diverse valid inputs throughout.

RQ1: *RLCheck* quickly converges to generating a high percentage of diverse valid inputs, and on most benchmarks generates orders of magnitude more diverse valid inputs than our baselines.

5.2 Covering Different Valid Behaviors

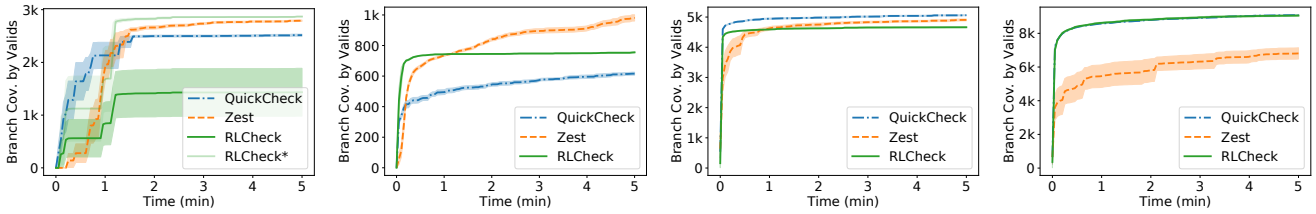
Section 5.1 shows that *RLCheck* generates many more diverse valid inputs than the baselines, i.e. solves the diversifying guidance problem more effectively. A natural question is whether the valid inputs generated by each method cover the same set of input behaviors (RQ2). For this, we can compare the cumulative branch coverage achieved by the valid inputs generated by each technique.

Figure 8 shows the coverage achieved by all valid inputs for each benchmark until timeout. The results are much more mixed than the results in Section 5.1. On the Closure benchmark (Fig. 8d), QuickCheck and *RLCheck* achieve the same amount of branch coverage by valid inputs. On Rhino (Fig. 8c) QuickCheck dominates slightly. On Maven (Fig. 8b), *RLCheck* takes an early lead in coverage but Zest's coverage-guided algorithm surpasses it at timeout.

On Ant (Figure 8a), *RLCheck* appears to perform poorly, but this is mostly an artifact of *RLCheck*'s bad luck in finding a first valid input. Again, for comparison's sake, *RLCheck** shows the results for only those runs that generate valid inputs: we see that *RLCheck*'s branch coverage is slightly above Zest's on these runs.

The overall clearest trend from Figure 8 is that *RLCheck*'s branch coverage seems to quickly peak and then flatten compared to the other techniques. This suggests that our MCC-based algorithm, while it is exploring diverse valid inputs, may still be tuned too much towards exploiting the knowledge from the first set of valid inputs it generates. We discuss in Section 6 some possible avenues to explore in terms of the RL algorithm.

RQ2: No method achieves the highest branch coverage on all benchmarks. *RLCheck*'s plateauing branch coverage suggests that it may be learning to generate diverse inputs with similar features rather than discovering new behavior.



(a) Ant **(b) Maven** **(c) Rhino** **(d) Closure**

Figure 8: Number of branches covered by valid inputs generated by each technique. Higher is better

5.3 Greybox Information

Given that *RLCheck* is able to attain its objective as defined by the diversifying guidance problem—generating large numbers of unique valid inputs—(Section 5.1), but does not achieve the highest branch coverage over all benchmarks (Section 5.2), a natural question is to ask whether choosing a different ξ, one that is coverage-aware, could help increase the diversity of behaviors discovered. This is what we seek to answer in RQ3.

For this experiment, we re-ran *RLCheck* both blackbox, i.e. with $\xi_{bb} = id$, and with greybox information, using $\xi_{gb}(x)$ = "the set of all branches covered by the input x". Thus, Greybox *RLCheck* is rewarded when it discovers a valid input that covers a distinct set of branches compared to all generated inputs. Note that this does not reward the guide more for generating an input which covers a wholly-uncovered branch, compared to an input that covers a new combination of already-seen branches. Again, we ran each method for 10 trials, timing out at 5 minutes.

Figures 9 shows the number of diverse valid inputs (valid inputs with distinct traces) generated the the blackbox and greybox versions of *RLCheck*, and Figure 10 shows the branch coverage by valid inputs for these two versions. We see universally across all benchmarks and both metrics that Blackbox *RLCheck* outperforms Greybox *RLCheck*. This suggests that the slowdown incurred by instrumentation is not worth the increased information *RLCheck* gets in the greybox setting. The difference is less striking for branch coverage than number of diverse valid inputs generated, because fewer inputs are required to get the same cumulative branch coverage.

We see much lower variation in Ant in this experiment because on all 10 runs, Blackbox *RLCheck* was able to generate at least one valid input for Ant. We chose random seeds at random in both experiments, so this is simply a quirk of experimentation.

> RQ3: Adding greybox feedback to *RLCheck* in terms of the characteristic function ξ causes a large slowdown, but no huge gains in number of valid inputs or coverage achieved. Overall, *RLCheck* performs best as a black-box technique.

6 DISCUSSION

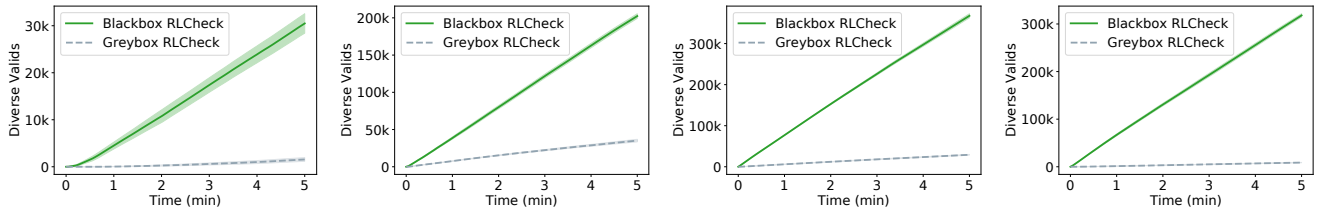
Performance. Tabular methods such as ours do not scale well for large choice or state spaces. Let S and C denote state and choice space sizes, respectively. The Monte Carlo control algorithm requires $O(SC)$ space overall, and requires $O(C)$ time for evaluating the policy function π. This is because all the algorithmic decision-making is backed by a large Q-table with $S \times C$ entries. Because of these constraints we had to restrict our state and choice spaces for *RLCheck*. For example, in our JavaScript implementation, when selecting integer values, we restricted our choice space to the range $\{0, 1, \ldots, 10\}$ rather than a larger range like $\{0, 1, \ldots, 100\}$. This was necessary to generate inputs in a reasonable amount of time with this generator. Function approximation methods, such as replacing the Q-table with a neural network, may be necessary for dealing with larger, more complex, state and choice spaces.

Increasing Exploration. In Section 5.2 we observed that the branch coverage achieved by *RLCheck*-generated valid inputs tends to quickly plateau, even for benchmarks where the other methods could achieve higher branch coverage (Figs 8b, 8c). This suggests that even with a high ε, MCC may still be too exploitative for the diversifying guidance problem. One approach to increase exploration would be to allow the learners to "forget" old episodes so choices made early in the testing session that are not necessary to input validity do not persist throughout the session. Curiosity-based approaches, which strongly encourage exploration and avoid revisiting states [41], may also be applicable.

Fine-tuning. In our experiments we heuristically chose the ε and k values, and then kept them fixed across benchmarks. We noted the importance of a large ε and modest k value for both generating unique valid inputs and doing so quickly. We also chose the reward function heuristically and in our design process we noticed how that this choice significantly affected performance, and particularly the distribution of invalid, valid, and unique valid inputs generated. It may be valuable to fine tune these hyperparameters and reward functions for different benchmarks.

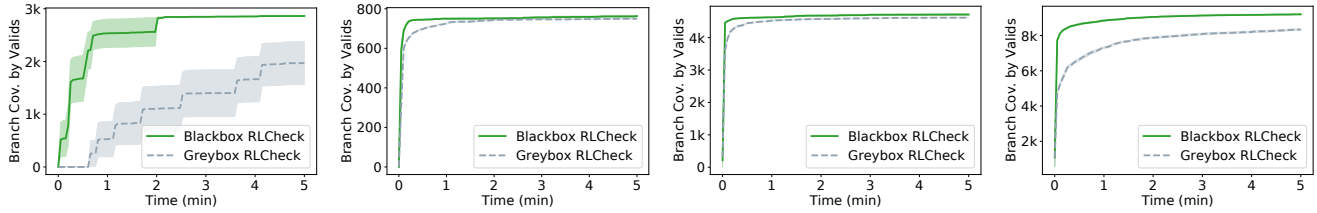
Bootstrapping. In Section 5.1 we saw that *RLCheck* had difficulty generating a first valid input for very strict validity functions (Ant). This limitation could be overcome by allowing *RLCheck* to be *bootstrapped*, i.e. given a sequence of choices that produces a valid input at the beginning of testing. This choice sequence could be user-provided, as long as there exists a relatively short sequence of choices resulting in a valid input.

Relevance of Diverse Valid Inputs for Testing. In aswering RQ1, we established that *RLCheck* was able to generate orders-of-magnitude more diverse valid inputs. A natural question is whether this metric is relevant for testing goals such as bug finding. While we did not conduct an extensive study of bug-finding ability as part of our research questions, we did look at *RLCheck*'s bug-finding ability on our benchmark programs as compared to QuickCheck in Zest.



(a) Ant (b) Maven (c) Rhino (d) Closure

Figure 9: Number of diverse valid inputs generated by each technique. Higher is better.



(a) Ant (b) Maven (c) Rhino (d) Closure

Figure 10: Number of branches covered by valid inputs generated by each technique. Higher is better.

During our evaluation runs, the techniques found a subset of the bugs described in the Zest paper [39]. Table 1 lists, for each bug that was discovered during out evaluation runs, the average time to discovery (TTD) and reliability (percent of runs on which the bug was found) for each method. Bugs are deduplicated, as done in the Zest paper, by exception type.

We see that on the Ant, where RLCheck found 1000× more diverse valid inputs than QuickCheck, it found bug (#1) 4× faster and 5× more often than QuickCheck. It was also faster than Zest. On Closure, where RLCheck found 60× more diverse valid inputs than Zest, it was also 20× faster at finding fault (#2). In contrast, on Rhino, RLCheck only found 1.4× more unique valid inputs than QuickCheck. In fact, as shown in Figure 6c, over 30% of generator-generated inputs already satisfied the validity function. Thus, we see that the plain generator-based approach (QuickCheck) had the best fault discovery of the three methods. This benchmark is representative of situations where the generator is already fairly well-tuned for the validity function of the program under test.

Overall, we believe these results suggest, but are not conclusive in showing, that order-of-magnitude increases in input diversity result in better fault discovery.

7 RELATED WORK

The problem of automatically generating test inputs that satisfy some criteria has been studied for over four decades. Symbolic execution [18, 28] as well as its dynamic [15] and concolic [24, 44] variants attempt to generate test inputs that reach program points of interest by collecting and solving symbolic constraints on inputs. Despite numerous advances in improving the precision and performance of these techniques [10, 12, 22, 29, 45], the path explosion problem [16] remains a key challenge that limits scalability to large complex constraints.

Table 1: Average time to discovery (TTD) and Reliability (Rel.)—the percentage of runs on which the bug was found—for bugs found by each technique during our experiments. Bugs are deduplicated by benchmark and exception type. Dash "-" indicates bug was not found.

	RLCheck		QuickCheck		Zest	
Bug ID	TTD	Rel.	TTD	Rel.	TTD	Rel.
Ant, (#1)	41s	50%	178s	10%	123s	90%
Closure, (#2)	1s	100%	1.2s	100%	23s	60%
Rhino, (#3)	95s	90%	62s	70%	276s	10%
Rhino, (#4)	11s	100%	1s	100%	30s	100%
Rhino, (#5)	-	-	3s	100%	80s	100%
Rhino, (#6)	-	-	96s	20%	-	-

Fuzz testing [36] is a popular method to generate byte-sequence test inputs. The key idea is to generate a huge quantity of test inputs at random, without incurring much cost for each individual input. Input validity requirements can be addressed either via user-provided input format specifications [5] or by mutating existing valid inputs [6]. Coverage-guided fuzzing, popularized by tools such as AFL [49], improves the effectiveness of mutation-based fuzz testing by instrumenting the program under test and incorporating feedback in the form of code coverage achieved by each test execution; the feedback is used to perform an evolutionary search for test inputs that cover various behaviors in the test program.

Search-based software testing [25, 26, 34, 48] generates inputs which optimize some objective function by using optimization techniques such as hill-climbing or simulated annealing. These

techniques work well when the objective function is a smooth function of the input characteristics.

QuickCheck [17] introduced the idea of formulating tests as properties $\forall x : P(x) \Rightarrow Q(x)$, which could be validated to some degree by randomly generating many instances of x that satisfy $P(x)$. Of course, the main challenge is in ensuring that $P(x)$ is satisfied often enough. Some researchers have attempted to write generators that produce diverse valid inputs by construction, such as for testing compilers [35, 47], but solutions turn out to be highly complex domain-specific implementations. For some domains, effective generators can also be automatically synthesized [32, 37]. Targeted property-based testing [33] biases hand-written input generators for numeric utility values. Domain-specific languages such as Luck [30] enable strong coupling of generators and validity predicates. In contrast, we address the problem of biasing arbitrary input generators towards producing inputs that satisfy arbitrary validity functions, without any prior domain knowledge.

Recently, techniques such as Zest [39], Crowbar [20], and FuzzChick [31] have combined ideas from coverage-guided mutation-based fuzzing with generative techniques such as QuickCheck. Although code coverage guidance helps discover new program behaviors, it comes at the cost of program instrumentation, which significantly reduces the number of test inputs that can be executed per unit time. In contrast, we address the problem of generating valid test inputs when considering the program as a black box.

There has been increasing interest in using machine learning to improve fuzz testing; Saavedra et al. [43] provide a survey. Böttinger et al. [13] propose a deep reinforcement learning approach to fuzz testing. This work uses the reinforcement learning agent to, given a subsequence of the input file as state, perform a mutation action on that subsequence. Instead of learning to mutate serialized input strings directly, *RLCheck* employs reinforcement learning on generators for highly structured inputs.

ALisp [11] is a language for specifying reinforcement learning problems in a hierarchical manner. Similarly to our work, these hierarchical learning programs contain choices points where an agent learns the best choice to take given the current state. ALisp splits value functions into three functions: value of the current action, current subroutine, and overall execution. It provides functionality for the user to specify which parts of the state each of these values depends on, which bears some similarity to our notion of context. The input-generator domain has some key differences from the hierarchical learning domain. Notably, we get reward only at the end of the episode, when an input is generated. Thus, it is unclear whether RLCheck value functions can be effectively split into ALisp's three components.

8 CONCLUSION

In this paper we investigated a reinforcement learning approach to guiding input generators to generate more valid inputs for property-based testing. We began by formalizing the problem of generating many unique valid inputs for property-based testing as the diversifying guidance problem. We proposed *RLCheck*, where generators follow a Monte Carlo Control (MCC)-based guide to generate inputs. We found that *RLCheck* has great performance in terms of generating many diverse valid inputs on real-world benchmarks. However, MCC seems to be prone to overfitting to a certain space of valid inputs. We believe more exploration-oriented RL approaches could be better suited to provide the guidance in *RLCheck*.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback, which helped us improve the paper. This research is supported in part by gifts from Samsung, Facebook, and Fujitsu, as well as by NSF grants CCF-1900968, CCF-1908870, and CNS-1817122.

REFERENCES

[1] 2019. Eris: Porting of QuickCheck to PHP. https://github.com/giorgiosironi/eris. Accessed January 28, 2019.
[2] 2019. FsCheck: Random testing for .NET. https://hypothesis.works/. Accessed January 28, 2019.
[3] 2019. Hypothesis for Python. https://hypothesis.works/. Accessed January 28, 2019.
[4] 2019. JSVerify: Property-based testing for JavaScript. https://github.com/jsverify/jsverify. Accessed January 28, 2019.
[5] 2019. PeachFuzzer. https://www.peach.tech. Accessed August 21, 2019.
[6] 2019. Radamsa: a general-purpose fuzzer. https://gitlab.com/akihe/radamsa. Accessed August 21, 2019.
[7] 2019. ScalaCheck: Property-based testing for Scala. https://www.scalacheck.org/. Accessed January 28, 2019.
[8] 2019. test.check: QuickCheck for Clojure. https://github.com/clojure/test.check. Accessed January 28, 2019.
[9] Cláudio Amaral, Mário Florido, and Vítor Santos Costa. 2014. PrologCheck–property-based testing in Prolog. In *International Symposium on Functional and Logic Programming*. Springer, 1–17.
[10] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-driven compositional symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 367–381.
[11] David Andre and Stuart J. Russell. 2002. State Abstraction for Programmable Reinforcement Learning Agents. In *Eighteenth National Conference on Artificial Intelligence*. American Association for Artificial Intelligence, USA, 119–125.
[12] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1083–1094.
[13] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. 2018. Deep Reinforcement Fuzzing. *CoRR* abs/1801.04589 (2018). arXiv:1801.04589 http://arxiv.org/abs/1801.04589
[14] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, USA, 123–133. https://doi.org/10.1145/566172.566191
[15] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*.
[16] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.
[17] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*.
[18] Lori A. Clarke. 1976. A program testing system. In *Proc. of the 1976 annual conference*. 488–491.
[19] David Coppit and Jiexin Lian. 2005. Yagg: An Easy-to-use Generator for Structured Test Inputs. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. ACM, New York, NY, USA, 356–359. https://doi.org/10.1145/1101908.1101969
[20] Stephen Dolan. 2017. Property fuzzing for OCaml. https://github.com/stedolan/crowbar. Accessed Jul 23, 2019.
[21] Roy Emek, Itai Jaeger, Yehuda Naveh, Gadi Bergman, Guy Aloni, Yoav Katz, Monica Farkash, Igor Dozoretz, and Alex Goldin. 2002. X-Gen: A random test-case generator for systems and SoCs. In *High-Level Design Validation and Test Workshop, 2002. Seventh IEEE International*. IEEE, 145–150.
[22] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. 225–234. https://doi.org/10.1145/1806799.1806835

[23] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*.

[24] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*.

[25] Mark Harman. 2007. The current state and future of search based software engineering. In *2007 Future of Software Engineering*. IEEE Computer Society, 342–357.

[26] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* 43, 14 (2001), 833–839.

[27] Paul Holser. 2014. junit-quickcheck: Property-based testing, JUnit-style. https://pholser.github.io/junit-quickcheck. Accessed August 21, 2019.

[28] James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19 (July 1976), 385–394. Issue 7.

[29] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. In *Acm Sigplan Notices*, Vol. 47. ACM, 193–204.

[30] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: A Language for Property-based Generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 114–129. https://doi.org/10.1145/3009837.3009868

[31] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. [n.d.]. Coverage Guided, Property Based Testing. *Proc. ACM Program. Lang.* 2, OOPSLA ([n. d.]).

[32] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2017. Generating Good Generators for Inductive Relations. *Proc. ACM Program. Lang.* 2, POPL, Article 45 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158133

[33] Andreas Löscher and Konstantinos Sagonas. 2017. Targeted Property-based Testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 46–56. https://doi.org/10.1145/3092703.3092711

[34] Phil McMinn. 2011. Search-Based Software Testing: Past, Present and Future. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW '11)*. IEEE Computer Society, Washington, DC, USA, 153–163. https://doi.org/10.1109/ICSTW.2011.100

[35] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-driven QuickChecking of Compilers. *Proc. ACM Program. Lang.* 1, ICFP (2017). http://doi.acm.org/10.1145/3110259

[36] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279

[37] Agustín Mista, Alejandro Russo, and John Hughes. 2018. Branching Processes for QuickCheck Generators. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (Haskell 2018)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/3242744.3242747

[38] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-guided Property-based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. ACM, New York, NY, USA, 398–401. https://doi.org/10.1145/3293882.3339002

[39] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. https://doi.org/10.1145/3293882.3330576

[40] Manolis Papadakis and Konstantinos Sagonas. 2011. A PropEr Integration of Types and Function Specifications with Property-based Testing. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang (Erlang '11)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/2034654.2034663

[41] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. 2017. Curiosity-driven Exploration by Self-supervised Prediction. In *ICML*.

[42] Talia Ringer, Dan Grossman, Daniel Schwartz-Narbonne, and Serdar Tasiran. 2017. A Solver-aided Language for Test Input Generation. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 91 (Oct. 2017), 24 pages. https://doi.org/10.1145/3133915

[43] Gary J. Saavedra, Kathryn N. Rodhouse, Daniel M. Dunlavy, and W. Philip Kegelmeyer. 2019. A Review of Machine Learning Applications in Fuzzing. *CoRR* abs/1906.11133 (2019). arXiv:1906.11133 http://arxiv.org/abs/1906.11133

[44] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*.

[45] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 842–853.

[46] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcment Learning: An Introduction*. MIT Press. http://www.incompleteideas.net/book/ebook/node53.html

[47] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*.

[48] Shin Yoo and Mark Harman. 2007. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 140–150.

[49] Michał Zalewski. 2014. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl. Accessed January 11, 2019.

[50] Michał Zalewski. 2014. American Fuzzy Lop Technical Details. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed Aug 2019.