Learning-based Controlled Concurrency Testing

Suvam Mukherjee
Microsoft Research, India
t-sumukh@microsoft.com

Pantazis Deligiannis
Microsoft Research, India
pdeligia@microsoft.com

Arpita Biswas
Indian Institute of Science, India
arpitab@iisc.ac.in

Akash Lal
Microsoft Research, India
akashl@microsoft.com

December 18, 2019

Concurrency bugs are notoriously hard to detect and reproduce. Controlled concurrency testing (CCT) techniques aim to offer a solution, where a *scheduler* explores the space of possible interleavings of a concurrent program looking for bugs. Since the set of possible interleavings is typically very large, these schedulers employ heuristics that prioritize the search to "interesting" subspaces. However, current heuristics are typically tuned to specific bug patterns, which limits their effectiveness in practice.

In this paper, we present QL, a learning-based CCT framework where the likelihood of an action being selected by the scheduler is influenced by earlier explorations. We leverage the classical Q-learning algorithm to explore the space of possible interleavings, allowing the exploration to adapt to the program under test, unlike previous techniques. We have implemented and evaluated QL on a set of microbenchmarks, complex protocols, as well as production cloud services. In our experiments, we found QL to consistently outperform the state-of-the-art in CCT.

1 Introduction

Testing concurrent programs for defects is extremely challenging. The difficulty stems from a combination of potentially exponentially large set of program behaviors due to thread interleavings, and the dependence of a bug on a specific, and rare, ordering of actions between concurrent workers.[1] The term *Heisenbug* has often been used to refer to concurrency bugs because they can be hard to find, diagnose and fix [1, 2]. Unfortunately, traditional techniques such as stress testing are unable to uncover many concurrency bugs. Such techniques offer little control over orderings among workers, thus fail to exercise a sufficient number of program behaviors, and are a poor certification for the correctness or reliability of a concurrent program. Consequently, concurrent programs often contain insidious defects that remain latent until put into production, leading to actual loss of business and customer trust [3, 4, 5].

Previous work on addressing this problem can broadly be classified into *stateful* and *stateless* techniques, depending on whether they rely on observing the state of the program or not. Stateful techniques require a precise representation of a program's current state during execution. Examples of successful tools in this space include SPIN [6] and ZING [7] that do enumerative exploration, as well as tools such as NUSMV [8, 9] that do symbolic exploration. These techniques, however, are generally applied to a *model* of the program. Each of the tools mentioned above have their own input representation for programs. Our own interest, however, is in testing real-world code, for which stateful tools cannot be directly applied because of the inability to snapshot the state of an executing program.

Stateless techniques overcome the requirement of recording program state, and thus, are a better starting point for testing real-world code. Such techniques require taking over the scheduling in a program. By controlling all scheduling decisions of which worker to run next, they can reliably explore the program state space. The exploration can be systematic (i.e., exhaustive in the limit) or randomized. Since the number of executions of a concurrent program (also called *interleavings*) is usually very large, schedulers leverage heuristics that prioritize searching "interesting"

[1] Concurrency can come in many forms: between tasks, threads, processes, actors, and so on. In this paper, we will use the term *worker* to refer to the unit of concurrency of a program.

subspaces of interleavings where bugs are likely to occur. For example, many bugs can be caught with executions that only have a few *context switches* [10] or a few *ordering constraints* [11] or a few number of *delays* [12], and so on. These heuristics have been effective at finding several classes of concurrency bugs in microbenchmarks and real-world scenarios [13, 14]. We refer to the general setup of applying a scheduler to (real-world) programs for the purpose of finding concurrency bugs as *controlled concurrency testing* (CCT).

Our goal is to improve upon the state-of-the-art for CCT. Our technique leverages lessons from the area of *reinforcement learning* (RL) [15, 16], which also has been concerned with the problem of efficient state-space exploration. The general RL scenario comprises an *agent* interacting with its *environment*. Initially, the agent has no knowledge about the environment. At each step, the agent can only partially observe the state of the environment, and invoke an *action* based on some policy. As a result of this action, the environment transitions to a new state, and provides a *reward* signal to the agent. The objective of the agent is to select a sequence of actions that maximizes the expected reward. RL techniques have been applied to achieve spectacular successes in domains such as robotics [17, 18], game-playing (Go [19], Atari [20], Backgammon [21]), autonomous driving [22, 23, 24], business management [25, 26], transportation [27, 28], chemistry [29, 30] and many more.

We map the problem of CCT to the general RL scenario. In essence, the RL agent is the CCT scheduler and the RL environment is the program: the scheduler (agent) decides the action in the form of which worker to execute next, and the program (environment) executes the action by running the worker for one step, and then passing control back to the scheduler to choose the next action. RL techniques are robust to partial state observations, which implies that we only need to partially capture the state of an executing program, which is readily possible. In that sense, our technique is neither stateless nor stateful, but rather somewhere in between. The main contribution of this paper is a scheduler based on the classical Q-Learning algorithm [31, 32]. To the best of our knowledge, this scheduler is the first attempt at applying learning-based techniques to the problem of CCT.

How does the use of RL compare against the stateless exploration techniques described earlier? The latter build off empirical observations to define heuristics that optimize for a particular subspace of interleavings. This can, however, have its shortcomings when the heuristics fail to apply. The heuristics optimize for a particular bug pattern, but fall-off in effectiveness as soon as the bug escapes that pattern. This renders the testing to be brittle against new classes of bugs. For instance, a bug that is triggered by a combination of two patterns will not be found by a scheduler looking for just one of those patterns (see example in Table 1).

This problem of brittleness is further compounded in scenarios where the concurrency is not the only form of non-determinism in the program. Programs can have *data* non-determinism as well (we refer to concurrency as a form of *control* non-determinism). Data non-determinism is used to generate unconstrained scalar values, for example, to model user input or choices made outside the control of the application under test. In these cases, heuristics dictating how to resolve the non-determinism may not even exist: the resolution of data non-determinism may be program specific (see more in Section 2).

Lastly, existing schedulers do not *learn* from the explorations performed in previous iterations. We show how RL does not exhibit the falling-off behavior for complex bug patterns, and systematically learns from exploration done previously, even in the presence of data non-determinism.

It is worth noting that RL, in general, has two phases. The first phase is concerned with learning a strategy for the agent through exploration, and in the second phase the agent simply applies the learnt strategy to navigate the environment. With CCT, we limit our attention to the first phase because our objective is simply to explore the state space of the program, and stop as soon as a bug is discovered.

We implemented our RL-based scheduler (which we denote as `QL`) in `P#` [33], an open-source industrial-strength framework for building and testing concurrent applications and distributed services. We evaluated `QL` on a wide range of `P#` applications, including production distributed services spanning tens of thousands of LOC, complex protocols, and multithreaded programs. Our results show that `QL` outperforms state-of-the-art CCT techniques in terms of bug finding ability. In some scenarios, `QL` was the only scheduler that was able to expose a particular bug. Moreover, in many cases, its frequency of triggering bugs was higher compared to other schedulers.

Our work is a starting point for the application of RL-based search to CCT. We anticipate that many further improvements are possible by tuning knobs such as the reward function, partial state observations, etc. For this purpose, we make our implementation and (non-proprietary) benchmarks available open-source[2].

The main contributions of this paper are as follows:

[2] `https://github.com/pdeligia/psharp-ql`

1. We provide the first mapping of the problem of testing a program for concurrency bugs onto the general reinforcement learning scenario (Section 4).
2. We provide an implementation of the RL-based scheduling strategy on `P#`, an industrial-strength CCT framework (Section 5).
3. We perform a thorough experimental evaluation of our RL-based scheduler on a wide range of applications, including production distributed services (Section 6).

The rest of this paper is organized as follows. In Section 2, we provide a high-level overview of our learning-based scheduling strategy. We cover background material on controlled concurrency testing and reinforcement learning in Section 3, and then present the `QL` exploration strategy in Section 4. We discuss the implementation details of `QL` in the `P#` framework in Section 5 and discuss our experimental evaluation in Section 6. We present related work in Section 7 and conclude in Section 8.

2 Overview

This section provides an overview of the key benefits of our learning-based scheduler. We consider a message-passing model of concurrency in our examples. That is, programs can have one or more workers executing concurrently. Each worker has its own local state and communicates with other workers via messages. When the program reaches a "bad" state, it automatically stops and raises an error.

The goal of CCT is to find some execution of a given program that raises an error. CCT typically works by serializing the execution of the program, allowing only one worker to execute at a given point. More precisely, CCT is parameterized by a *scheduler* that is called at each step during the program's execution. The scheduler must pick one *action* among the set of all enabled actions at that point to execute next. For simplicity, assume that a worker can have at most one enabled action, which implies that picking an enabled action is the same as picking an enabled worker. Section 3 presents our formal model that also considers the possibility of each worker having multiple enabled actions in order to allow for data non-determinism.

We focus this section on two particular schedulers that have been used commonly in prior literature. The first is a pure random scheduler. This scheduler, whenever it has to make a scheduling decision, it picks one worker uniformly at random from the set of all enabled workers. The second scheduler is *probabilistic concurrency testing* (PCT) [11]. In PCT, each worker is assigned a unique priority. The scheduler always selects the highest-priority worker, except that at a few randomly chosen points during the program's execution, it changes (decreases) the priority of the highest-priority worker. The two schedulers are formalized in Section 3. We use additional schedulers in our evaluation (Section 6).

Learning-based scheduler Our main contribution is `QL`, a learning-based scheduler. As mentioned in the introduction, we consider the scheduler to be an *agent* that is interested in exploring the *environment*, i.e., the program under test. The agent can (partially) observe the configuration of the environment which, in our case, is the program state. Whenever the agent asks the environment to execute a particular action, it gets feedback in the form of a *reward*. The agent attempts to make decisions that maximizes its reward.

`QL` employs an adaptive learning algorithm called Q-Learning [31]. For each state-action pair, `QL` associates a (real-valued) quality metric called *q-value*. When `QL` observes that the program is in state s, it picks an action a from the set of all enabled actions with probability that is governed by the q-value of (s, a). After one run of the program finishes, `QL` updates the q-values of each state-action pair that it observed during the run, according to the rewards that it received, and uses the updated q-values for choosing actions in the next run.

The goal of `QL` is to explore the program state space, covering as many diverse set of executions as it can. In other words, the scheduler should maximize coverage. For this purpose, we set the reward to always be a fixed *negative* value (-1), which has the effect of *disincentivizing* the scheduler from visiting states that it has seen before: the more times a state has been visited before, the higher the likelihood of `QL` of staying away from it in future runs. The choice remains probabilistic and the probabilities never go to 0. This is essential because the scheduler only gets to observe the program state partially.

Adaptive learning has important ramifications for exploration. Unlike stateful strategies, `QL` can operate on abstractions of program configurations, thereby allowing it to scale to production-sized codebases (see Section 6). Unlike stateless strategies, `QL` adapts its decisions based on program executions and not due to hard-wired rules.

Power of state observations Consider the program in Figure 1, with two workers A and B continually sending messages, denoted 0 and 1 respectively, to a third worker C.

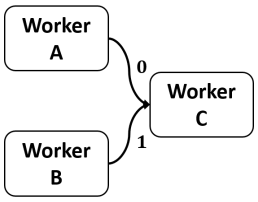

Figure 1: Simple example with three workers A, B and C.

Worker C has a constant n-length string $\eta \in \{0,1\}^*$, as well as a counter m that is initialized to 0. If the i-th message received by C (which is 0 if sent by A or 1 if sent by B), matches the i-th character of η, then m is incremented, else it is set to -1 and is never updated again. The program reaches a bad configuration if $m = n$. Note that given any string η, there is exactly one way of scheduling between A and B so that C raises an error after n messages.

We measure the effectiveness of each scheduler as the B-% value: the percentage of buggy program runs in a sufficiently large number of runs. Table 1 shows the results for Random, PCT and QL, for different choices of the string η.

	B-%		
String	**Random**	**PCT**	QL
$\eta_1 = 0^9 1$	0.08	0.97	10.3
$\eta_2 = (01)^5$	0.09	✗	10.4
$\eta_3 = (01)^3 0^3 1$	0.09	✗	10.3

Table 1: B-% for Random, PCT (priority-change budget of 3) and QL scheduling strategies for the program in Figure 1.

The Random scheduler has similar B-% values for the various strings. It is easy to calculate this result analytically: for any string η of length n in the above example, Random has $\frac{1}{2^n}$ chance of producing that string, as it must choose between workers A and B exactly according to η. Although B-% does not change with the string, the effectiveness of finding the bug is poor because of the exponential dependence on the string length. The PCT scheduler biases search for certain types of strings. As seen in Table 1, when the string matches the PCT heuristics, its effectiveness is much better than Random, however, it has no chance otherwise.

QL is able to expose the bug for each of the strings with a much higher B-% compared to the other two schedulers. QL benefits greatly from observing the state of the program as it performs exploration. For this example, we set it up to observe just the value of counter m of worker C. QL optimizes for coverage, and because the counter is set to -1 on a wrong scheduling choice, it is forced to learn scheduling decisions that keep incrementing m, which leads to the bug.

Resilience to bug patterns Different schedulers have their own strengths: they are more likely to find bugs of a certain pattern over others. Consider a slight extension to the Figure 1 example where we add another worker D that sends a *single* message (denoted 2) to C and then halts. In this case, Random is much more likely to find the string $\eta_4 = 2(0^9 1)$ (B-% of 0.05%) compared to the string $\eta_5 = (0^9 1)2$ (B-% of 0.003%). The reason is that Random is much more likely to schedule D earlier in the execution rather than later. PCT works much better at scheduling D late (B-% of 0.02% for η_5). However, this means that a string like $\eta_6 = (01)^5 2$ falls out of scope for both Random and PCT: it requires D to be scheduled late, and it requires a prefix that PCT cannot schedule. QL does not demonstrate such behavior: it has a high B-% for each of these strings (at least 1.7%).

Optimizing for coverage The QL scheduler attempts to learn scheduling decisions that increase coverage. That is, it tries to uncover new states that it has not observed before. It does not directly attempt to learn scheduling decisions that reveal the bug. The bug-finding ability is a by-product of increased coverage. Consider the "calculator" example illustrated in Figure 2.

The worker `Calculator` maintains a counter that is initialized to 0. Each of the other workers sends exactly 10 identical messages to `Calculator`. In response to a message sent by worker `Add`, `Calculator` increments its counter by 1. Similarly, in response to a message sent by `Subtract`, `Multiply`, `Divide`, and `Reset`, the `Calculator` worker will subtract 1, multiply by 2, divide (integer division) by 2, and reset the counter to 0, respectively. In this example, there are a large number of concurrent executions, however, their effect is such that many of them result in the same `Calculator` state. Figure 3 shows the coverage that each scheduler can achieve as we increase the number of program runs. Here, coverage is measured in terms of the number of distinct counter values generated across all runs. QL dramatically

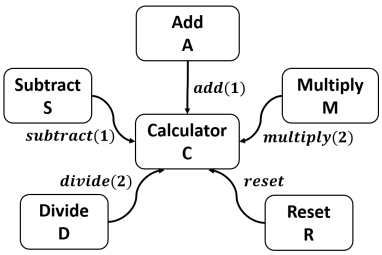

Figure 2: A simple calculator example.

outperforms both Random and PCT: its state observations (which is the `Calculator` counter value) allows it to push the program in many different corner cases that other schedulers do not.

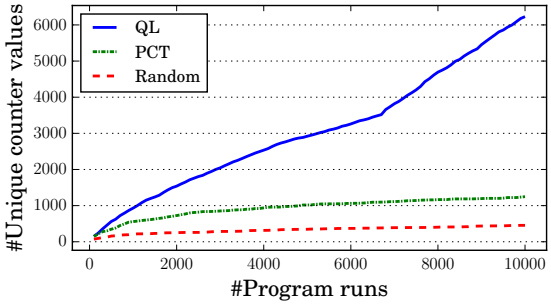

Figure 3: Measuring coverage in the calculator example.

Picking the right state abstraction The above examples show that learning from state observations provides an effective alternative to generic (program-independent) scheduling heuristics. This leaves open the question of what constitutes a good state observation? Even for these simple examples, if we make the observation too imprecise (e.g., not observing any state), then `QL` deteriorates to Random-like performance because there is nothing to learn. If we make the observation to be too precise (e.g., observing the string of messages received at C) then the effectiveness reduces because the learning slows down in a large state space. Our evaluation shows two interesting trends. First, a generic state observation (that is readily available from a CCT tool) is enough to make `QL` very effective in practice for a large class of benchmarks. Second, the state observation can be tuned on a per-program basis to further improve its effectiveness, offering users a means of improving the effectiveness of testing for the programs they care about most.

Data non-determinism So far we have only considered the choice of which worker to execute next. However, programs can have non-determinism within a worker as well. A common example is the use of a Boolean $\star$ operator to model choices made outside the control of the program (i.e., the program is expected to work correctly irrespective of how this choice is made). The following snippets demonstrate how the $\star$ operator can be used to model timeouts (that may or may not fire) or error conditions (e.g., calling an external routine may return one of two different exceptions).

```
if ( * )
{
  timeout();
}
```

```
if ( * )
{
    throw Exception1();
}
else
{
    throw Exception2();
}
```

Prior work on stateless schedulers does not account for such data non-determinism. The common practice is to resolve $\star$ uniformly at random: there are no heuristics that determine if one return value should be preferred over the other. With partial state observations, `QL` can do much better. We demonstrate this fact with a simple example. Consider rewriting the program of Figure 1: replace workers A and B with a single worker W, as shown in Figure 4. The worker W makes a non-deterministic choice (in a loop) and either sends 0 to C or sends 1. Semantically, this program is not different from the one in Figure 1. However, PCT's performance regresses because the scheduling between workers is immaterial: only the non-deterministic choice matters, for which sampling uniformly-at-random is the only option

available. Interestingly, QL retains its performance, performing just as well as it did for the program of Figure 1 because the scheduler treats all non-determinism in the same manner.

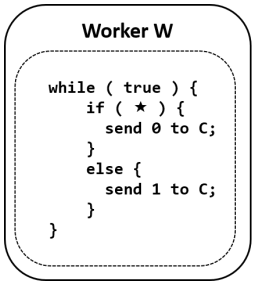

Figure 4: A program with data non-determinism.

3 Preliminaries

This section presents background material on CCT and reinforcement learning and sets up the notation that we follow in the rest of the paper.

3.1 Controlled Concurrency Testing

3.1.1 Programming Model

A program $\mathcal{P}$ is a tuple $\langle \mathcal{W}, \Sigma, \sigma_{init}, A, \Theta \rangle$, where $\mathcal{W}$ denotes a (finite) set of concurrently executing *workers* and Σ denotes the set of program *configurations*, with $\sigma_{init} \in \Sigma$ being the initial configuration. The set of *actions* that the program can execute is denoted by A. We use the meta-variables σ, a and w to range over the sets Σ, A and $\mathcal{W}$ respectively. Let $\omega : A \mapsto \mathcal{W}$ be an onto function that maps each action to the unique worker which executes it. The function $\Theta : A \mapsto 2^{\Sigma \times \Sigma}$ denotes the set of *transitions* associated with a. We write $\sigma \xrightarrow{a} \sigma'$ iff $(\sigma, \sigma') \in \Theta(a)$. Further, we assume that transitions associated with an action are deterministic. That is, for each action a, if $\sigma_1 \xrightarrow{a} \sigma_2$ and $\sigma_1 \xrightarrow{a} \sigma_3$ then $\sigma_2 = \sigma_3$.

We define some helper functions that will be used in the rest of the paper. The function $IsError(\sigma)$ returns true if the configuration σ represents an erroneous configuration, and returns false otherwise. We define the functions $enabled(\sigma)$, $enabled^w(\sigma)$ and $\vartheta(\sigma)$ as follows:

$$
\begin{aligned}
enabled(\sigma) &\stackrel{\text{def}}{=} \{a \in A \mid \exists \sigma' \in \Sigma : \ \sigma \xrightarrow{a} \sigma'\} \\
enabled^w(\sigma) &\stackrel{\text{def}}{=} \{a \in enabled(\sigma) \mid \omega(a) = w\} \\
\vartheta(\sigma) &\stackrel{\text{def}}{=} \{w \in \mathcal{W} \mid enabled^w(\sigma) \neq \emptyset\}
\end{aligned}
$$

The function $enabled(\sigma)$ returns the set of all actions that $\mathcal{P}$ can execute when it is at the configuration σ, while $enabled^w(\sigma)$ returns those actions in $enabled(\sigma)$ that can be executed by a worker w. Lastly, $\vartheta(\sigma)$ returns all workers that have at least one enabled action in the configuration σ. With this notation, $|\vartheta(\sigma)| > 1$ represents *control* non-determinism: there is a choice between which worker will take the next step. When $|enabled^w(\sigma)| > 1$, it represents *data* non-determinism: the worker w itself can have multiple enabled actions.

3.1.2 Schedulers

A *schedule* ℓ of length N is defined to be a sequence of program transitions starting from the initial configuration:

$$
\ell \stackrel{\text{def}}{=} \left\langle \sigma_{init} \xrightarrow{a_1} \sigma_1 \xrightarrow{a_2} \ldots \xrightarrow{a_N} \sigma_N \right\rangle
$$

We use $|\ell|$ to denote the length of a schedule, and write $\ell \xrightarrow{a'} \sigma'$ to denote a schedule that extends ℓ with the single (valid) transition $\sigma_N \xrightarrow{a'} \sigma'$. We also refer to the transition $\sigma_{i-1} \xrightarrow{a_i} \sigma_i$ as the i-th step of the schedule. We call a

Algorithm 1: Generic Exploration Algorithm.

Input: Scheduler SCH
Input: Program $\mathcal{P}$, Max-Steps M, Max-Iterations N

```
1  foreach i ∈ {1, ..., N} do
2      σ ← σ_init, ℓ ← ⟨σ_init⟩
3      SCH.PrepareNext()
4      foreach j ∈ {1, ..., M} do
5          if IsError(σ) ∨ enabled(σ) = φ then
6              break
7          end
8          a ← SCH.GetNext(enabled(σ))
9          σ' ← Execute(P, σ, a)
10         ℓ ← (ℓ -a-> σ')
11         σ ← σ'
12     end
13     if IsError(σ) then
14         return ℓ
15     end
16 end
17 return ⟨⟩
```

schedule *buggy* if its final configuration represents an error. Note that because actions are deterministic, a schedule can equivalently be represented by its sequence of actions.

For a given program $\mathcal{P}$, CCT aims to explore possible schedules of $\mathcal{P}$ using the generic exploration algorithm shown in Algorithm 1. The algorithm is parameterized by a scheduler SCH. It accepts bounds M and N on the length of each schedule, and the total number of schedules to explore, respectively. Algorithm 1 returns a buggy schedule if discovered, else it returns an empty sequence.

Algorithm 1 iteratively explores multiple schedules of $\mathcal{P}$, up to bound N (line 1). In each iteration, the scheduler is informed via a call to $PrepareNext$ to prepare for executing a new schedule (line 3). Each schedule consists of at most M steps, after which the schedule is aborted and a new one is attempted. In each step, the scheduler SCH is asked to pick an action from the set of all enabled actions via a call to $GetNext$ (line 8). The selected action is then executed (line 9): given σ and a, $Execute(P, \sigma, a)$ returns σ' such that $\sigma \xrightarrow{a} \sigma'$. The process continues until a bug is found or the algorithm hits the bound N on the number of explored schedules.

The scheduler SCH controls the exploration strategy. We describe two different scheduler instantiations next.

Random A purely randomized exploration strategy can be obtained by setting the $PrepareNext$ function to `skip`, and making the $GetNext$ function return an action chosen uniformly at random from its given set of enabled actions $enabled(\sigma)$. Prior work has noted that such a simple strategy is still effective at finding bugs in practice [14].

Probabilistic concurrency testing (PCT) The PCT scheduler described here is an adaptation of the original algorithm [11] to our setting. PCT is a priority-based scheduler that is parameterized by a given bound D, called the priority-change-point budget. We assume that the program has a fixed number of workers: $|\mathcal{W}| = W$. The $PrepareNext$ method of the scheduler (randomly) assigns a unique initial priority to each worker in the range $\{D, D+1, \ldots, D+W\}$. It also constructs a set $Z = \{z_1, \ldots, z_{D-1}\}$ of $D-1$ distinct numbers chosen uniformly at random from the set $\{1, \ldots, M\}$. Assume that Z is sorted, so that $z_j \leq z_{j+1}$ for each $j \leq D-2$.

The idea behind PCT is to choose the highest-priority worker at each step. Priorities remain fixed, except at priority-change points when the scheduler shifts priorities. More precisely, when $GetNext$ is called for choosing the i^{th} action, it first picks the highest-priority worker w. Next, it checks if i equals z_j for some $z_j \in Z$. If so, it decreases the priority of w to j. (Note that $j \leq D-1$, so it is the least priority among all currently-assigned priorities.) It then re-picks the highest-priority worker w'. Finally, it returns an enabled action of w' chosen uniformly at random from its set of all enabled actions. (This last part deals with data non-determinism.)

Schedulers need not always be probabilistic or randomized. It is also possible to define a DFS-like scheduler that is guaranteed to explore all schedules of the program in the limit, although such systematic schedulers tend not to work well in practice [14]. There are many other schedulers defined in prior literature [34, 14, 12, 10].

Delay bounding (DB) The DB scheduler [12], Algorithm 1 requires an additional argument D, which bounds the number of times the strategy deviates from an underlying deterministic scheduling strategy[3], which we assume to be round-robin (RR). We assume that the program has a fixed number of workers: $|\mathcal{W}| = W$. Each deviation incurs a cost, in the form of one or more units of *delay*, and D limits the total accumulated delays. The RR strategy maintains a map $\mathcal{W} \to \mathbb{N}$ assigning to each worker a unique natural number which serves as its identifier. We abuse notation slightly to use $\omega(a)$ to denote the identifier associated with the worker executing action a. For a given configuration σ, the $GetNext$ for RR selects a worker w according to a round-robin ordering based on the identifiers, and returns an action a drawn uniformly at random from $enabled^w(s)$. For any two workers with identifiers x and y, we use $\Delta(x, y)$ to denote the round-robin distance d between x and y, that is $y = (x + d) \bmod W$. For a schedule ℓ of length M and a worker with id x, $delays(s, x)$ denotes the total number of enabled workers which are bypassed when the worker with identifier x is scheduled after $\omega(a_M)$:

$$delays(\ell, x) \stackrel{\text{def}}{=} |\{y \ : \ 0 \leq y \leq \Delta(\omega(a_M), x) \wedge \omega(a_M) + x \bmod M \in \vartheta(s_M)\}|$$

The total amount of delay d_{tot} is recursively defined as

$$\begin{aligned} d_{tot}(\ell) &\stackrel{\text{def}}{=} 0 && \text{if } |\ell| = 0 \text{ or } |\ell| = 1 \\ &\stackrel{\text{def}}{=} d_{tot}(\ell') + delays(\ell', \omega(a')) && \text{if } \ell = \ell' \xrightarrow{a'} s' \end{aligned}$$

The DB exploration algorithm maintains the total delay d incurred so far (and is initialized to 0 in $PrepareNext$). The $GetNext$ function either returns an action a according to the RR strategy, or deviates from it—in which case the delays incurred (computed using the definition for d_{tot}) are added to d. If $d > D$, the $GetNext$ strictly adheres to the RR strategy.

3.2 Reinforcement Learning

The Reinforcement Learning (RL) [35, 36, 15] problem, outlined in Figure 5, comprises an agent interacting with an environment, about which it has no prior knowledge. At each step, the agent takes an action, which causes the environment to undergo a state transition. The agent then observes the new state of the environment, and receives feedback in the form of a reward or penalty. The goal of the agent is to learn a sequence of actions that maximizes its expected reward.

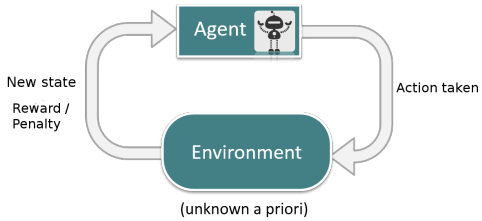

Figure 5: The Reinforcement Learning problem.

The environment is unknown a priori, i.e., the effect of executing an action is not known. This makes the RL problem hard, but also generally applicable. In the RL literature, it is common to model the environment as a Markov Decision Process (MDP) over the partial state observation. An MDP is a stochastic state-transition model, comprising the following components:

1. A set of states $\mathcal{S}$, representing partial observations of the environment's actual state.
2. A set of actions A that the agent can instruct the environment to execute.
3. Transition probabilities $\mathcal{T}(s, a, s')$, which denote the probability that the environment transitions from a state $s \in \mathcal{S}$ to $s' \in \mathcal{S}$, on taking action $a \in A$.
4. Reward $\mathcal{R}(s, a)$ obtained when the agent takes an action a from the state s.

[3]Our discussion of DB is based on the description in [14].

Assume that the environment is in some state $s \in \mathcal{S}$, and the agent instructs it to execute a sequence of actions $(a_1, a_2, \ldots)$, denoted as $\langle a_t \rangle$. Let s_{i+1} denote the state of the environment after executing the action a_{i+1} at state s_i. Then, the *expected discounted reward*, for s and $\langle a_t \rangle$ is defined as

$$\mathcal{V}(s, \langle a_t \rangle) \stackrel{\text{def}}{=} \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i \mathcal{R}(s_i, a_{i+1}) \;\middle|\; s_0 = s\right]$$

The parameter $\gamma \in (0, 1]$ is called the *discount factor*, and is used to strike a balance between immediate rewards and long-term rewards. The *optimal* expected discounted reward for a state s can be written as:

$$\begin{aligned} \mathcal{V}^*(s) &\stackrel{\text{def}}{=} \max_{\langle a_t \rangle} \mathcal{V}(s, \langle a_t \rangle) \\ &= \max_{a \in A} \left(\mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') \mathcal{V}^*(s') \right) \end{aligned}$$

The Q-function $Q^*(s, a)$ denotes the expected value obtained by taking action a at state s, and is written as

$$Q^*(s, a) \stackrel{\text{def}}{=} \left(\mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') \mathcal{V}^*(s') \right) \tag{1}$$

The agent makes use of a *policy* function, $\pi : \mathcal{S} \mapsto A$, to determine the action to be executed at a given state of the environment. The *optimal* policy $\pi^*(s)$ can be obtained by computing the Q-function for each state-action pair, and then selecting the action a that maximizes $Q^*(s, a)$ (by solving the Bellman equations [37]). However, when the MDP is unknown, instead of *computing* the optimal policy, the agent needs to adaptively *learn* the policy from its history of interaction with the environment. RL techniques help in systematic exploration of the unknown MDP by learning which sequences of actions are more likely to earn better rewards (or incur less penalties).

One such popular RL algorithm is *Q-Learning* [38, 31], which estimates Q^* values using point samples. Starting from an initial state s_0, Q-Learning iteratively selects an action a_{i+1} at state s_i and observes a new state s_{i+1}. Now, let $\langle s_t \rangle$ be a sequence of states obtained by some policy and $\langle a_t \rangle$ be the sequence of corresponding actions then, given an initial estimate of Q_0, the Q-learning update rule is as follows:

$$Q_{i+1}(s_i, a_{i+1}) \leftarrow (1 - \alpha) \cdot Q_i(s_i, a_{i+1}) + \alpha \cdot \left(\mathcal{R}(s_i, a_{i+1}) + \gamma \max_{a'} Q_i(s_{i+1}, a') \right) \tag{2}$$

Here, $\alpha \in [0, 1]$ is called the *learning parameter*. When $\alpha = 0$, the agent does not learn anything new and retains the value obtained at i^{th} step, while $\alpha = 1$ stores only the most recent information and overwrites all previously obtained rewards. Setting $0 < \alpha < 1$, helps to strike a balance between the new values and the old ones.

The Q-Learning algorithm does not explicitly use the transition probability distribution of the underlying MDP during the update step (unlike Equation 1), and is hence called a *model-free* algorithm. Such algorithms are advantageous when the state-space is huge and it is computationally expensive to try learning all the transition probability distributions.

Exploration methods like *random* completely ignore the historical agent-environment interactions [39, 40], while *counter-based methods* [41, 42] take decisions based solely on the frequency of visited states. In contrast, Q-Learning takes informed stochastic decisions that eventually make worse actions less likely. In our work, we use Softmax [15, 43] as our policy in order to bias the exploration against unfavorable actions based on the current Q-values. According to the Softmax policy, the probability of choosing an action a (from a set of possible choices A) at state s, is given by $\frac{e^{Q(s,a)}}{\sum_{a' \in A} e^{Q(s,a')}}$. Thus, lower the $Q(s, a)$ value, lesser the likelihood of taking action a again at state s.

4 `QL`: Q-Learning-based Controlled Concurrency Testing

This section describes our `QL` scheduler. Let $\mathcal{H}$ be a user-defined function that maps $\Sigma \mapsto \mathcal{S}$, where Σ is the set of program configurations and $\mathcal{S}$ is a set of *abstract states*. When the environment (program) is in the configuration σ, then $\mathcal{H}(\sigma)$ represents the observation that the agent will make about the environment. One can think of $\mathcal{H}$ as defining an abstraction over the program's configuration space. In reality, $\mathcal{H}$ is implemented as a hashing function that is applied to only a fraction of the program's configuration. The reward function $\mathcal{R} : \mathcal{S} \times A \mapsto \mathbb{R}$ is fixed to be a constant -1, that is, $\mathcal{R}(s, a) = -1$ for all states s and actions a.

Given a schedule $\ell = \langle \sigma_{init} \xrightarrow{a_1} \sigma_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} \sigma_n \rangle$, let $\mathcal{H}(\ell)$ denote the *abstracted* schedule with each σ_i replaced by $s_i = \mathcal{H}(\sigma_i)$. The `QL` scheduler is parameterized by the abstraction function $\mathcal{H}$. Its *GetNext* and *PrepareNext* procedures are described in Algorithm 2 and Algorithm 3, respectively. The `QL` scheduler maintains a partial map $\mathcal{Q} : \mathcal{S} \times A \mapsto \mathbb{R}$. For an abstract state s and action a, $\mathcal{Q}(s, a)$, when defined, represents the q-value associated with the state-action pair (s, a).

Algorithm 2: GetNext-`QL`

Input: Set of actions $\{a_1, \ldots, a_n\}$, Configuration σ

1 $s \leftarrow \mathcal{H}(\sigma)$
2 **foreach** $a \in \{a_1, \ldots, a_n\}$ **do**
3 **if** $\mathcal{Q}(s, a)$ ***is undefined*** **then**
 `/* Initialize q-value of new` (s, a) `pair to` 0 `*/`
4 $\mathcal{Q}(s, a) \leftarrow 0$
5 **end**
6 **end**
7 $\mathcal{D} \leftarrow \langle\rangle$ `/* probability distribution over actions */`
8 **foreach** $i \in \{1, \ldots, n\}$ **do**
9 $\mathcal{D}(i) \leftarrow \frac{e^{\mathcal{Q}(s,a_i)}}{\sum_{j=1}^{n} e^{\mathcal{Q}(s,a_j)}}$
10 **end**
11 $i \leftarrow$ `Sample`$(\mathcal{D})$
12 **return** a_i

Algorithm 2 takes as input a program configuration σ, and the set of (say, n) actions $enabled(\sigma)$. We fix an arbitrary ordering among these actions and use a_i to refer to the i^{th} action in this order. The *GetNext* procedure first computes $\mathcal{H}(\sigma)$ and stores it in the variable s. For each input action a, if the q-value for (s, a) is not present in $\mathcal{Q}$, then it is initialized to 0 (lines 2-6). Next, a variable $\mathcal{D}$ is initialized. Lines 8-10 creates a probability distribution $\mathcal{D}$ over the set of n enabled actions using the Softmax policy. Finally, Algorithm 2 samples from the distribution $\mathcal{D}$ (i.e., it picks i with probability $\mathcal{D}(i)$) and returns the corresponding action.

Algorithm 3: PrepareNext-`QL`

Input: Schedule $\ell = \langle \sigma_0 \xrightarrow{a_1} \sigma_1, \ldots, \sigma_{n-1} \xrightarrow{a_n} \sigma_n \rangle$

1 $\hat{\ell} \leftarrow \mathcal{H}(\ell)$ `/* Sets` $\hat{\ell}$ `to` $\langle s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n \rangle$ `*/`
2 **foreach** $i \in \{n, \ldots, 1\}$ **do**
3 `maxQ` $\leftarrow \max_a \ \mathcal{Q}(s_i, a)$
4 $\mathcal{Q}(s_{i-1}, a_i) \leftarrow (1 - \alpha) \cdot \mathcal{Q}(s_{i-1}, a_i) + \alpha \cdot \Big(\mathcal{R}(s_{i-1}, a_i) + \gamma \cdot$ `maxQ`$\Big)$
5 **end**

We use the *PrepareNext* procedure to update the q-values according to the previously executed schedule. We assume that *PrepareNext* is not called in the first iteration of Algorithm 1, and in each subsequent iteration, it is passed the completed schedule from the previous iteration. Algorithm 3 updates the q-values for each state-action pair (s_{i-1}, a_i) in the abstracted schedule $\hat{\ell}$, according to Equation 2. Note that the negative reward causes the update rule to decrease the q-value for each observed (s, a) state-action pair. As a result, the Softmax policy lowers the likelihood of selecting a in subsequent iterations when the state s is encountered, thereby increasing the chances of discovering newer states.

5 Implementation

The `P#` Framework We have implemented the `QL` scheduling strategy in `P#` [33, 44], an open-source industrial-strength framework for building and testing concurrent as well as distributed applications. A `P#` program comprises a set of concurrently executing actors (referred to as *machines* in `P#`), communicating with each other via message passing. Each machine in a `P#` program is equipped with an inbox of incoming messages. It dequeues these messages in a single-threaded fashion and processes them one after the one. The processing of a message can update the local state of the machine, or create new machines, or send messages to existing ones. `P#` also provides an API (called NONDET) that returns an unconstrained Boolean value; a program can use NONDET to model data non-determinism. Optionally, a machine can internally have a state-machine (SM) structure that is offered in `P#` for programming convenience.

P# allows the programmer to specify safety and liveness specifications of the application alongside its implementation. The framework comes bundled with a CCT tool, called P#-Tester, which implements Algorithm 1 and systematically tests the code against provided specifications. P#-Tester takes over the scheduling of a P# program. It serializes the execution, allowing only one machine to proceed at a time. A machine is allowed to execute until it hits a *scheduling point*, at which point the control goes back to the P#-Tester and it can choose to schedule some other machine. A scheduling point is inserted right before each *send* and *create* operation. Note that in an actor programming model like P#, a create or send are the only operations that may potentially not commute with all other operations in the program, so there is no need for a scheduling point before each instruction [45]. The P#-Tester also inserts a scheduling point at each call to NONDET and it is up to the scheduler to pick the return value. The resolution of NONDET in all schedulers that come with P# is purely random. QL is the first scheduler that does otherwise.

The semantics of a P# program is easily instantiated from the one described in Section 3.1.1: a worker is simply a machine and an action is all steps that a machine can take until it hits the next scheduling point. A machine, at any point, can have at most one enabled action, except in cases when it is at a call to NONDET, in which case it can have two enabled actions (one for each possible return value of NONDET).

Our decision to use P# was motivated by the following reasons:

1. We were able to get access to production distributed services written in P#. This allowed us to evaluate QL on these highly concurrent services comprising tens of thousands of LOC, in addition to complex protocols.
2. P#-Tester contains implementations of several state-of-the-art scheduling strategies, including Random and PCT. This allowed us to compare QL against independently-implemented schedulers that have been shown to be effective in practice.
3. P# provides the necessary scaffolding to quickly implement the QL scheduler. The framework exposes APIs to perform suitable abstractions of the program configuration, retrieve the set of actions, etc., and has been used in prior studies as well [13, 46, 47].

Extension to multithreaded applications To demonstrate that our technique is not just limited to actor-based message-passing programs, we implemented an extension to P# that allowed experimenting with multithreaded programs. In this case, a worker is a thread, which communicates with other workers via shared memory. We assumed that synchronization operations (marking scheduling points) were given to us by the programmer. For the multi-threaded benchmarks that we used in our evaluation, the only synchronization operations used were lock acquire and release.

QL scheduler We experimented with several different $\mathcal{H}$ functions for obtaining state observations. Each of them were implemented as a hashing routine that mapped the current program configuration to an integer; they only differed in how much of the program state was hashed. In each case, we first constructed a per-worker (per-machine or per-thread) hash and then applied a commutative hash (in our case, a simple multiplication) of these individual hashes. Thus, in order to describe an abstraction function, we only describe how we construct the hash of a single machine.

We consider the following variants of QL that use a different per-worker hash:

1. $\texttt{QL}^{\texttt{i}}$: hashes only the contents of the inbox of a machine. (This only applies to actor-based programs.)
2. $\texttt{QL}^{\texttt{d}}$: For the case of actors, this hashes the inbox contents, current machine operation (send or create or `NonDet`), as well as the current state of the machine's SM (if any). For the case of multithreaded applications, this only hashes the lock that a thread is currently trying to acquire or release.
3. $\texttt{QL}^{\texttt{c}}$: hashes a machine/thread using a user-defined hashing routine that is free to look at any runtime component used by $\texttt{QL}^{\texttt{d}}$, as well as the local state of the machine/thread. We added convenient APIs to the P# framework to let a user provide this hashing scheme.

It is important to note that $\texttt{QL}^{\texttt{i}}$ and $\texttt{QL}^{\texttt{d}}$ are generic schedulers: they only make use of information that is readily available for all P# programs. They are broadly applicable to any message-passing system. The variant $\texttt{QL}^{\texttt{c}}$ is program-specific because it requires a user-defined routine. We use $\texttt{QL}^{\texttt{i}}$ and $\texttt{QL}^{\texttt{d}}$ to show the general power of our scheduler, whereas the other variants are intended to show that user intuition can further enhance the testing quality.

Our implementation of QL in P#-Tester closely mirrors Algorithm 2 and Algorithm 3, with the exception of two optimizations that we describe next. The implementation maintains a map $\texttt{TF} : \mathcal{S} \rightarrow \texttt{int}$. $\texttt{TF}[s]$ records the number of times the scheduler has encountered the hash s during exploration. Our first optimization is that for a given hash s, we multiply the reward in line 4 of Algorithm 3 with $\texttt{TF}[s]$. This optimization allows QL to rapidly learn to avoid exploring a state repeatedly.

The second optimization applies to the reward function. It is, by default, set to -1 for all state-action pairs. However, for an action a that correspond to sending a message of a *special* type, we assign a high negative reward (-1000) to (s, a) for all states s. This change has the following effect. For some state s and a special action a, once the scheduler has explored a schedule that takes action a on state s, it will be highly discouraged to fire action a in the future when the program is in state s. This optimization helps the scheduler improve its diversity of schedules when it comes to the insertion of this special action. In our experiments, this special action is the injection of a *failure* message that is used by programmers to test the failover logic of their distributed system. A failure message is intended to bring down a set of machines, so one can test the effects of failures in their system.

These optimizations, along with the choice of the abstraction function showcase the degree of control available to a user for enhancing the testing experience for the program they care about most. In contrast, other schedulers offer little control to the user.

For all our experiments, we set the values of α and γ, used in Algorithm 3, to 0.3 and 0.7, respectively. We justify this choice in Section 6.

The ability to observe the program configuration is an added source of information for `QL` compared to stateless schedulers. To evaluate the effectiveness of the learning aspects of the algorithm, we came up with two other strategies to serve as a baseline for `QL`, described below.

Greedy scheduler The `Greedy` scheduler maintains a `TF` map, similar to `QL`. When the program reaches a state s, the scheduler computes the set $\mathcal{N}_s$ of possible target states the program can transition to from s:

$$\mathcal{N}_s = \{s' \mid \exists a \in A : s \xrightarrow{a} s'\}$$

The `Greedy` scheduler chooses a state $s' \in \mathcal{N}_s$ having the lowest $\texttt{TF}[s']$ value, executes the corresponding action and increments $\texttt{TF}[s']$. In case of a tie among several s' states, `Greedy` draws the target state uniformly at random.

Iterative delay-bounding scheduler We also compare against a variant of the *delay-bounding* (DB) [12] scheduler. In our experiments, we found the original DB algorithm to be ineffective at exposing bugs, so we started with an optimized version of it [34] and modified it to leverage state observations. In our variant, which we call *iterative delay-bounding* (IDB), at each scheduling decision, it either selects an action of the worker w that executed the last action, or randomly context-switches to a worker different from w. The number of such random context-switches per run is bounded (called the *delay bound*). IDB keeps tracks of the visited states as it performs exploration. It starts with a delay bound of 0 and then iteratively increments it when no new states are discovered in the last 100 runs.

6 Evaluation

This section describes our empirical evaluation that compares `QL` against other schedulers in terms of bug-finding ability, coverage, robustness to data non-determinism, choice of the state observations and overhead in terms of testing time. All our experiments use the `P#`-Tester. We used existing implementations of schedulers when available (Random and PCT) and implemented others ourselves (`Greedy` and IDB). There were other schedulers (such as DFS) that performed very poorly compared to all others, so we leave them out of the evaluation.

We measure the effectiveness of finding bugs using the metrics $\texttt{Bugs}^{100}$ and $\texttt{Iter}^{10K}$, which we define next. For each program, and for each scheduler, we invoke the `P#`-Tester 100 times. Each invocation of the tester has a budget of exploring up to 10000 schedules (for a total of up to 1 million schedules across all invocations of the tester). The metric $\texttt{Bugs}^{100}$ is the number of times a bug was exposed out of the 100 invocations, while $\texttt{Iter}^{10K}$ is the average number of schedules a scheduler explored before exposing a bug (in runs when it did find a bug). The exact upper bound on the length of each schedule (in the number of scheduling decisions) was varied per program, and was typically in the order of 10^3.

We ran all experiments on a Windows 10 virtual machine configured with 8 Intel Xeon cores and 112GB of RAM.

Benchmarks We consider three different categories of benchmarks: complex *protocols* that are publicly available as part of the `P#` framework, *multithreaded* benchmarks from the SVCOMP [48] and SCTBENCH [14] benchmark suites, and *production* distributed services from Microsoft Azure. We ported over all the benchmarks from the `pthread` library in SVCOMP and all the benchmarks from SCTBENCH using our `P#` extension for multithreaded applications. Most of the SVCOMP and SCTBENCH benchmarks contained trivial bugs, with all the schedulers reporting $\texttt{Bugs}^{100}$ values of over 50. We report here the results for the rest of the benchmarks.

				$\texttt{Bugs}^{100}(\texttt{Iter}^{10K})$						
					Baseline strategies		**Budget-based strategies**			
	Benchmarks	**LoC**	**#T**	$\texttt{QL}^{\texttt{d}}$	**Random**	**Greedy**	**PCT-3**	**PCT-10**	**PCT-30**	**IDB**
Protocols	Raft-v1	1194	17	99 (76)	100 (2120)	83 (2526)	✗	12 (4131)	45 (4425)	28 (6810)
	Raft-v2	1194	17	95 (103)	4 (5813)	3 (5700)	✗	✗	✗	1 (6228)
	Paxos	849	10	66 (6315)	8 (4616)	20 (4678)	19 (5280)	91 (2747)	92 (3278)	33 (6852)
	Chord	859	7	34 (761)	✗	✗	✗	✗	✗	✗
	FailureDetector	692	5	99 (5420)	✗	✗	11 (5915)	100 (2043)	99 (2321)	31 (2601)
Multithreaded	Fib-Bench-2	55	3	100 (10)	100 (19)	100 (25)	✗	82 (1828)	100 (2)	100 (1475)
	Fib-Bench-Longest-2	55	3	100 (21)	100 (1904)	100 (310)	✗	✗	✗	100 (4466)
	Triangular-2	73	3	100 (58)	86 (3979)	100 (138)	✗	✗	2 (2013)	70 (4719)
	Triangular-Longest-2	73	3	100 (66)	✗	79 (3688)	✗	✗	✗	✗
	SafeStack	253	6	1 (148)	✗	23 (5044)	✗	✗	✗	46 (6955)
Production	ProdService1	56649	27	79 (3793)	14 (4202)	24 (4985)	37 (5137)	29 (4612)	25 (4235)	23 (5196)
	ProdService2-v1	33827	15	100 (1060)	✗	✗	100 (381)	100 (569)	37 (4158)	✗
	ProdService2-v2	33827	28	97 (3415)	✗	✗	100 (1104)	36 (6543)	✗	✗
	ProdService3-v1	18663	17	92 (1276)	100 (835)	16 (1268)	76 (2543)	96 (1296)	80 (3881)	✗
	ProdService3-v2	19771	17	100 (348)	100 (932)	10 (1018)	64 (4526)	100 (1947)	90 (1233)	✗

Table 2: Results from applying various schedulers on 5 protocols, 5 multithreaded programs and 5 production services. The reported program statistics are: lines of code (LoC), maximum number of concurrent workers (#T), $\texttt{Bugs}^{100}$and $\texttt{Iter}^{10K}$.

We briefly describe the protocol benchmarks and the custom hashing scheme that we implemented for them (for use with $\texttt{QL}^{\texttt{c}}$). Our choice of the hash was *not* guided by knowledge of the bug in the program, but rather by selecting the components of the program that were central to its logic.

Raft is a consensus protocol that uses a voting process to elect a leader among participating nodes. A timer fires periodically to initiate the voting process. The two versions of Raft differ in the number of timer events required to initiate the leader election. A bug in the protocol occurs when two leaders get elected simultaneously. We hash the state of the voting process: what each node voted for, the hash of the current leader, and the contents of the log for each node.

Chord is a protocol for a peer-to-peer distributed hash table, which maintains a collection of keys; their associated values are distributed among multiple nodes. A bug occurs if a client attempts to retrieve a key, which should be present, but gets a response that is does not exist. We hash the set of keys stored in each node.

FailureDetector is a protocol for detecting node failures. It contains a heartbeat mechanism, where a manager initiates rounds where each node is expected to send heartbeats. If a node fails to send a heartbeat within a period of time, it is deemed to have failed in that round. The custom hash includes the set of alive nodes, and the set of nodes that are still deemed alive in the current round. A bug occurs if three rounds elapse, and a node stays failed.

Paxos is a consensus protocol. A bug occurs if two different values are agreed upon simultaneously. We hash the set of proposed, accepted and learned values.

The selected set of benchmarks span a wide range of programs, covering a wide spectrum of bug patterns. All the benchmarks have a single known bug.

Effectiveness of QL in bug-finding Table 2 compares the performance of the $\texttt{QL}^{\texttt{d}}$ scheduler, on the $\texttt{Bugs}^{100}$ and $\texttt{Iter}^{10K}$ metrics, against the other schedulers. For PCT, we used a priority-switch bound of 3 because lower bounds performed worse. Interestingly, we found that higher bounds sometimes performed better (theoretically, lower bounds have a better chance in the worst-case), so we report bounds of 10 and 30 as well. However, in all production benchmarks, PCT regressed on $\texttt{Bugs}^{100}$ with bounds higher than 10.

Our main conclusions from Table 2 are as follows. Comparing to the other schedulers, $\texttt{QL}^{\texttt{d}}$ was the single scheduler which *never* missed a bug and its $\texttt{Bugs}^{100}$ was consistently among the highest. In Raft-v2, $\texttt{QL}^{\texttt{d}}$ had $\texttt{Bugs}^{100}$ of 95, whereas all other schedulers had $\texttt{Bugs}^{100}$ of less than 5. For Chord, $\texttt{QL}^{\texttt{d}}$ is the *only* scheduler that exposed the bug. For scenarios where other strategies matched the $\texttt{Bugs}^{100}$ metric for $\texttt{QL}^{\texttt{d}}$, the latter frequently had lower $\texttt{Iter}^{10K}$. Further, note that $\texttt{QL}^{\texttt{d}}$ only relies on the "black-box" state observation readily offered by P# with *no* additional inputs from the user.

The performances of PCT and IDB are very sensitive to the choice of parameters such as bounds on priority-change points and maximum schedule length. As an example, in the FailureDetector benchmark, PCT-30 performs better than

PCT-3, but it is the other way around for PRODSERVICE2-V1. We also found that the performance of PCT deteriorated as we increased the maximum schedule length parameter of P#-Tester. It is very hard for the user to make a good educated guess of these parameters a priori. QL is robust to such parameters, adding to its appeal for practical use.

Lastly, we find that exploration strategies based on state abstractions seem to perform well in general, even without learning. For example, the performance of the Greedy scheduler is competitive compared to PCT-3. The extra component of adaptive learning in QL provides it with an additional edge, allowing it to outperform Greedy.

Effect of handling data non-determinism in QL As we highlighted in earlier sections, QL is the first CCT scheduler that accounts for data non-determinism in a way other than resolving it uniformly-at-random. We evaluate its benefit in Table 3. We use QL^d-NDN to denote a version of QL^d where we turn off the handling of data non-determinism (i.e., resolve it uniformly-at-random). Table 3 highlights that QL^d-NDN clearly regresses in its ability to expose bugs. In fact, these results together show that for benchmarks such as Raft-v2 and Chord, QL outperforms all other schedulers because of its handling of data non-determinism.

	$Bugs^{100}(Iter^{10K})$	
Benchmarks	QL^d**-NDN**	QL^d
Raft-v1	86 (2551)	99 (76)
Raft-v2	1 (283)	95 (103)
Paxos	19 (6357)	66 (6315)
Chord	✗	34 (761)
PRODSERVICE1	22 (4482)	79 (3793)

Table 3: Comparing QL^d without and with the handling of data non-determinism.

Effect of state abstraction on QL We investigate the effect of using the different state abstractions with QL. Table 4 summarizes our findings. For the protocols, QL^i, which tracks only the contents of the inbox of the P# machines, sufficed to expose all bugs besides Chord. A possible reason for this is that the communication between concurrent workers in a P# program involves message passing, and the bugs are triggered by specific message reorderings. Thus, tracking just the inbox drives QL^i to explore different possible inbox contents for each P# machine, which results in the bug being exposed.

	$Bugs^{100}(Iter^{10K})$		
Benchmarks	QL^i	QL^d	QL^c
Raft-v1	100 (70)	99 (76)	100 (53)
Raft-v2	100 (96)	95 (103)	100 (81)
Paxos	30 (3713)	66 (6315)	33 (5256)
Chord	✗	34 (761)	2 (886)
FailureDetector	38 (3108)	99 (5420)	78 (6590)
SafeStack	N/A	1 (253)	31 (2438)

Table 4: Effect of using different state abstractions on QL.

We found the QL^d abstraction to be sufficient for exposing all the bugs in our benchmarks. Hashing additional local state, as allowed by QL^c may actually regress the $Bugs^{100}$ as can be seen for the Paxos and FailureDetector protocols in Table 4. A notable exception is SafeStack, which comprises a shared stack being concurrently updated by multiple threads. In this case, the QL^d abstraction was too coarse to expose the bug. Our custom abstraction involved tracking the exact contents of the stack, which forced QL^c to explore different stack contents thereby exposing the bug. Once again, this custom abstraction involved tracking a component which is central to the logic of the program, and is *not* based on our knowledge of the bug.

Coverage achieved by QL during exploration We also measured the number of unique abstract states covered by each of the schedulers across all iterations of the same run of P#-Tester. We used the default abstraction for this experiment. Figure 6 summarizes our findings for a subset of the benchmarks; these results are representative of our findings for the remaining protocols and production services.

We find that QL^d discovers significantly more unique abstract states, compared to the other schedulers. An interesting scenario occurs in PRODSERVICE2-V2. The benchmarks involves testing against injected failures. The Random

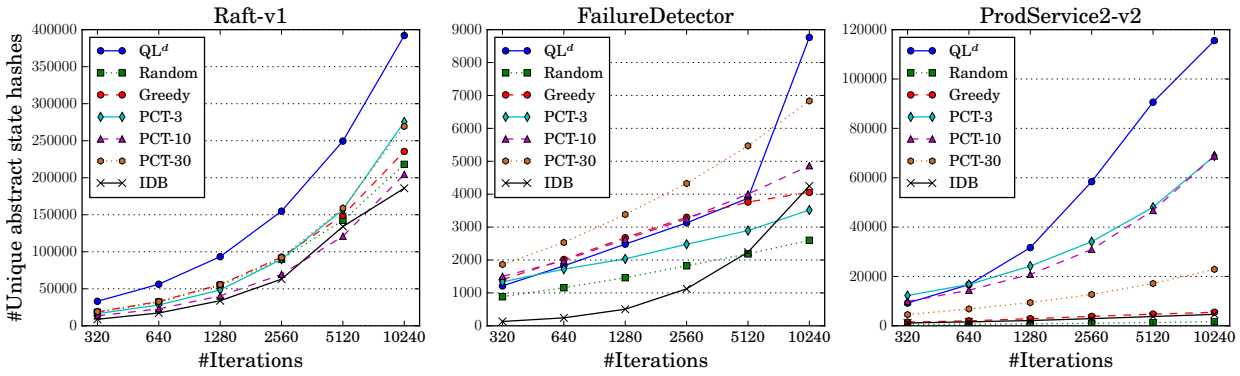

Figure 6: Number of unique abstract states explored by various schedulers.

scheduler repeatedly injected the failure very early in the run, which prevents the program from progressing to interesting corner cases. As a result, it fails to discover too many states (or uncover the bug).

Performance overhead of `QL` To evaluate the overhead of `QL` due to tracking states and performing adaptive learning, we measure the execution times of the $\texttt{QL}^{\texttt{d}}$, Random, Greedy and PCT-10 strategies on the three programs used in Figure 6. Table 5 summarizes our findings. In general, $\texttt{QL}^{\texttt{d}}$ has an overall slowdown of $7\times$ and $3\times$ compared to Random and PCT-10, respectively. However, the degree of coverage achieved by a scheduler has a direct bearing on the execution times. For example, as Figure 6 highlights, Random discovers far fewer states compared to $\texttt{QL}^{\texttt{d}}$ on the PRODSERVICE2-V2 program, which results in an overall faster execution time.

	Execution Time (seconds)			
Benchmarks	$\texttt{QL}^{\texttt{d}}$	**Random**	**Greedy**	**PCT-10**
Raft-v1	210	109	119	102
FailureDetector	767	491	594	533
PRODSERVICE2-V2	830	46	50	133

Table 5: Comparing execution times of $\texttt{QL}^{\texttt{d}}$, Random, Greedy and PCT-10 strategies for a single invocation of `P#`-Tester.

Dependence on the α and γ parameters Two key parameters in the Q-Learning algorithm are the α (learning rate) and γ parameters used in Equation 2. We evaluated the protocols for different combinations of α and γ, and we did not find a significant variation in either $\texttt{Bugs}^{100}$ or $\texttt{Iter}^{10K}$ due to the choices. We settled for $\alpha = 0.3$ and $\gamma = 0.7$ since it strikes a reasonable balance in Equation 2 between weighing immediate versus future rewards.

7 Related Work

Controlled Concurrency Testing Controlled concurrency testing has been subject to extensive research, and several tools and techniques exist which aim to find complex concurrency bugs [2, 12, 34, 49, 50, 11, 51]. Thomson et al. [14] provides a nice survey and empirical comparison of several recent stateless strategies such as probabilistic concurrency testing (PCT) [11] and delay-bounding [12]. We cover several of these techniques in detail in Section 3.

Learning based Software Testing Learning algorithms have been applied to the problem of *fuzzing* program inputs [52, 53, 54, 55]. In particular, Böttinger et al. [52] formalizes input fuzzing as an RL problem, and applies deep Q-learning to learn mutations that yield new program inputs that are likely to maximize code coverage. Zheng et al. [56, 57] applies supervised learning techniques to automatically identify program features or instrumentation predicates which are likely to trigger a bug, based on a data set comprising user reports of program executions. Mariani et al. [58] leverages Q-learning to generate test cases for testing GUI-based applications. The problem of input fuzzing is orthogonal to CCT: the former is about input values but the latter is primarily about controlling scheduling decisions.

Veanes et al. [59] formalizes the problem of *conformance checking* between a model and an implementation as a Markov Decision Process, and use simple heuristics inspired from reinforcement learning. Unlike our work, the paper assumes limited communication between concurrently executing agents, and also assumes the agents' actions being

deterministic. Moreover, their implementation handles a small toy example, whereas our implementation can scale to production code.

Baskiotis et al. [60] aims to maximize path coverage in sequential programs by identifying distinct feasible paths in the control flow graph with high probability, using an adaptive sampling mechanism. In contrast, our `QL` scheduler can handle concurrent programs and does not require an explicit control flow graph representation.

8 Conclusion

In this paper, we proposed a controlled concurrency testing (CCT) scheduler, called `QL`, which leverages Q-Learning to explore a user-defined abstraction of the program state space. Since our scheduler is geared towards coverage and adapts to the application under test, it is effective at finding concurrency bugs irrespective of their pattern. `QL` is also the first scheduler that accounts for data non-determinism. We implemented `QL` in an open-source industrial-strength CCT framework. In our benchmarks, comprising complex protocols and production cloud services, we showed that `QL` outperforms state-of-the-art CCT strategies.

References

[1] Jim Gray. Why do computers stop and what can be done about it? In *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12. IEEE, 1986.

[2] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 267–280. USENIX Association, 2008.

[3] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, Planning Report 02-3*, 2002.

[4] Amazon. Summary of the AWS service event in the US East Region. `http://aws.amazon.com/message/67457/`, 2012.

[5] Ben Treynor. GoogleBlog – Today's outage for several Google services. `http://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html`, 2014.

[6] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1st edition, 2011.

[7] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, pages 484–487, 2004.

[8] Edmund M. Clarke, Kenneth L. McMillan, Sérgio Vale Aguiar Campos, and Vasiliki Hartonas-Garmhausen. Symbolic model checking. In *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*, pages 419–427, 1996.

[9] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model checker. *STTT*, 2(4):410–425, 2000.

[10] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 446–455, 2007.

[11] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In James C. Hoe and Vikram S. Adve, editors, *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, pages 167–178. ACM, 2010.

[12] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. Delay-bounded scheduling. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 411–422. ACM, 2011.

[13] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering bugs in distributed storage systems during testing (not in production!). In Angela Demke Brown and Florentina I. Popovici, editors, *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016.*, pages 249–262. USENIX Association, 2016.

[14] Paul Thomson, Alastair F. Donaldson, and Adam Betts. Concurrency testing using controlled schedulers: An empirical study. *TOPC*, 2(4):23:1–23:37, 2016.

[15] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

[16] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1):1–103, 2010.

[17] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[18] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[19] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[21] Gerald Tesauro. Practical issues in temporal difference learning. In John E. Moody, Stephen Jose Hanson, and Richard Lippmann, editors, *Advances in Neural Information Processing Systems 4, [NIPS Conference, Denver, Colorado, USA, December 2-5, 1991]*, pages 259–266. Morgan Kaufmann, 1991.

[22] Sascha Lange, Martin Riedmiller, and Arne Voigtländer. Autonomous reinforcement learning on raw visual input data in a real world application. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2012.

[23] Matthew O'Kelly, Aman Sinha, Hongseok Namkoong, Russ Tedrake, and John C Duchi. Scalable end-to-end autonomous vehicle testing via rare-event simulation. In *Advances in Neural Information Processing Systems*, pages 9827–9838, 2018.

[24] Dong Li, Dongbin Zhao, Qichao Zhang, and Yaran Chen. Reinforcement learning and deep learning based lateral control for autonomous driving [application notes]. *IEEE Computational Intelligence Magazine*, 14(2):83–98, 2019.

[25] Qingpeng Cai, Aris Filos-Ratsikas, Pingzhong Tang, and Yiwei Zhang. Reinforcement mechanism design for fraudulent behaviour in e-commerce. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[26] Jing-Cheng Shi, Yang Yu, Qing Da, Shi-Yong Chen, and An-Xiang Zeng. Virtual-taobao: Virtualizing real-world online retail environment for reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4902–4909, 2019.

[27] Hua Wei, Guanjie Zheng, Huaxiu Yao, and Zhenhui Li. Intellilight: A reinforcement learning approach for intelligent traffic light control. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2496–2505. ACM, 2018.

[28] Harshad Khadilkar. A scalable reinforcement learning algorithm for scheduling railway lines. *IEEE Transactions on Intelligent Transportation Systems*, 20(2):727–736, 2018.

[29] Zhenpeng Zhou, Xiaocheng Li, and Richard N Zare. Optimizing chemical reactions with deep reinforcement learning. *ACS central science*, 3(12):1337–1344, 2017.

[30] Emre O Neftci and Bruno B Averbeck. Reinforcement learning in artificial and biological systems. *Nature Machine Intelligence*, 1(3):133–143, 2019.

[31] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[32] Jing Peng and Ronald J Williams. Incremental multi-step q-learning. *Machine Learning*, 22(1-3):283–290, 1996.

[33] Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. Asynchronous programming, analysis and testing with state machines. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 154–164. ACM, 2015.

[34] Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. Systematic testing of asynchronous reactive systems. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 73–83. ACM, 2015.

[35] Verena Heidrich-Meisner, Martin Lauer, Christian Igel, and Martin A Riedmiller. Reinforcement learning in a nutshell. In *ESANN*, pages 277–288. Citeseer, 2007.

[36] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003.

[37] Richard Bellman et al. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

[38] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[39] Steven D Whitehead. Complexity and cooperation in q-learning. In *Machine Learning Proceedings 1991*, pages 363–367. Elsevier, 1991.

[40] S Whitehead. A study of cooperative mechanisms for faster reinforcement learning univ. rochester, rochester. Technical report, NY, Tech. Rep. TR-365.

[41] Mitsuo Sato, Kenichi Abe, and Hiroshi Takeda. Learning control of finite markov chains with an explicit trade-off between estimation and control. *IEEE transactions on systems, man, and cybernetics*, 18(5):677–684, 1988.

[42] Andrew G Barto and Satinder Pal Singh. On the computational economics of reinforcement learning. In *Connectionist Models*, pages 35–44. Elsevier, 1991.

[43] Chris Watkins. Models of delayed reinforcement learning. In *PhD thesis, Psychology Department, Cambridge University*, 1989.

[44] The P# Team. P#: A framework for rapid development of reliable asynchronous software. `https://github.com/p-org/PSharp`, 2019.

[45] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, December 1975.

[46] Rashmi Mudduluru, Pantazis Deligiannis, Ankush Desai, Akash Lal, and Shaz Qadeer. Lasso detection using partial-state caching. In Daryl Stewart and Georg Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 84–91. IEEE, 2017.

[47] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. Randomized testing of distributed systems with probabilistic guarantees. *PACMPL*, 2(OOPSLA):160:1–160:28, 2018.

[48] Dirk Beyer. Automatic verification of C and java programs: SV-COMP 2019. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*, pages 133–155. Springer, 2019.

[49] Madanlal Musuvathi and Shaz Qadeer. Fair stateless model checking. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 362–371. ACM, 2008.

[50] Jirí Simsa, Randy Bryant, and Garth A. Gibson. dbug: Systematic testing of unmodified distributed and multi-threaded systems. In Alex Groce and Madanlal Musuvathi, editors, *Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*, volume 6823 of *Lecture Notes in Computer Science*, pages 188–193. Springer, 2011.

[51] Patrice Godefroid. Software model checking: The verisoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[52] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing. In *2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018*, pages 116–122. IEEE Computer Society, 2018.

[53] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: efficient fuzzing with neural program learning. *CoRR*, abs/1807.05620, 2018.

[54] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: machine learning for input fuzzing. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 50–59. IEEE Computer Society, 2017.

[55] Ketan Patil and Aditya Kanade. Greybox fuzzing as a contextual bandits problem. *CoRR*, abs/1806.03806, 2018.

[56] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: simultaneous identification of multiple bugs. In William W. Cohen and Andrew Moore, editors, *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*, volume 148 of *ACM International Conference Proceeding Series*, pages 1105–1112. ACM, 2006.

[57] Alice X. Zheng, Michael I. Jordan, Ben Liblit, and Alexander Aiken. Statistical debugging of sampled programs. In Sebastian Thrun, Lawrence K. Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003, December 8-13, 2003, Vancouver and Whistler, British Columbia, Canada]*, pages 603–610. MIT Press, 2003.

[58] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 81–90. IEEE Computer Society, 2012.

[59] Margus Veanes, Pritam Roy, and Colin Campbell. Online testing with reinforcement learning. In Klaus Havelund, Manuel Núñez, Grigore Roşu, and Burkhart Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification*, pages 240–253, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[60] Nicolas Baskiotis, Michèle Sebag, Marie-Claude Gaudel, and Sandrine Gouraud. A machine learning approach for statistical software testing. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, pages 2274–2279. Morgan Kaufmann Publishers Inc., 2007.