A Closer Look at the Use of Reinforcement Learning for Speeding Up Runtime Verification of Software Tests (Experience Paper)

SHINHAE KIM, Cornell University, USA
SAIKAT DUTTA, Cornell University, USA
OWOLABI LEGUNSEN, Cornell University, USA

Runtime verification (RV) found many bugs by monitoring passing tests against formal specifications (specs), but it is slow. A recent work, VALG, used reinforcement learning (RL) to speed up RV by up to 551.5x or 27 hours. VALG aims to probabilistically monitor most unique traces—sequences of spec-related events like method calls—and monitor fewer redundant ones. But, there is no in-depth study of VALG's current limits and how to address them. We study VALG on 93 Java open-source projects to answer five unaddressed questions. (i) *How much slower is VALG than optimal baselines?* Up to 323.8x, or 3.2 hours vs. running tests without RV, and up to 9.6x, or 25.3 minutes vs. a theoretically optimal baseline that monitors only unique traces. (ii) *Where is VALG's time spent?* 30.5% on monitoring and 18.4% on signaling events to monitors, on average. (iii) *What characterizes code where VALG monitors too many redundant traces or misses unique ones?* In 100 cases, 67.8% of redundant traces are due to limitations of VALG's RL convergence heuristic, and 41.3% of missed unique traces occur when a VALG assumption does not hold. (iv) *How much can test non-determinism and RL stochasticity cause monitored unique traces to vary?* By 42.5 percentage points (pp) and 12.3pp on average, respectively, but they vary by up to 98pp. (v) *How do other off-the-shelf RL algorithms compare with VALG's?* Only two of 11 RL algorithms that we survey are feasible for RV during continuous integration. Both are slower and miss more unique traces than VALG, so custom RL algorithms for RV may be needed. So, despite VALG's promising results, it has plenty of room to improve. We highlight several exciting future directions on using RL to speed up RV.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Reinforcement learning**;

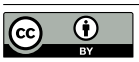
Additional Key Words and Phrases: Runtime Verification, Software Testing, Reinforcement Learning

ACM Reference Format:
Shinhae Kim, Saikat Dutta, and Owolabi Legunsen. 2026. A Closer Look at the Use of Reinforcement Learning for Speeding Up Runtime Verification of Software Tests (Experience Paper). *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA189 (October 2026), 22 pages. https://doi.org/10.1145/3832280

1 Introduction

Runtime verification (RV) monitors program executions against formal specifications (specs). RV first instruments a program to signal spec-related *events*, e.g., method calls or field accesses at runtime. Then, RV creates *monitors*—usually automata—to check *traces*, i.e., event sequences, against specs and report violations. Specs provide extra test oracles, so RV of tests in many open-source projects against specs of correct API usage helped find hundreds of bugs [52, 53, 61, 73, 74].

Authors' Contact Information: Shinhae Kim, sk3364@cornell.edu, Cornell University, Ithaca, New York, USA; Saikat Dutta, saikatd@cornell.edu, Cornell University, Ithaca, New York, USA; Owolabi Legunsen, legunsen@cornell.edu, Cornell University, Ithaca, New York, USA.



ACM 2994-970X/2026/10-ARTISSTA189
https://doi.org/10.1145/3832280

A longstanding hindrance to widespread RV adoption, especially during continuous integration (CI) [35, 36], is its runtime overhead. This overhead remained high, despite algorithms [18, 19, 22, 43, 67] and data structures [21, 58, 68] to reduce them. But, recent work [28] found that 99.87% of monitored traces are *redundant*; they are duplicates of the other *unique* 0.13%.

VALG [47, 48] was proposed to leverage this finding; it uses reinforcement learning (RL) to speed up RV during testing. VALG uses on-the-fly feedback about uniqueness of previously monitored traces to *selectively* monitor future ones. To do so, VALG formulates selective monitor creation as a two-armed bandit RL problem [79] that rewards creation of monitors that check unique traces and penalizes creation of those that check redundant traces. The specs being used are about method calls, so monitoring redundant traces has no bug-finding benefit during testing.

VALG showed promising results. Compared to JAVAMOP [44, 58] and TRACEMOP [30, 81]—two state-of-the-art (SoTA) RV techniques—VALG is up to 20.2x and 551.5x faster respectively, and finds 99.6% of spec violations that these techniques find. For instance, VALG needs only about 12 minutes to monitor three projects where TRACEMOP takes over three days [47]. VALG misses almost 25% of unique traces with default RL hyperparameters. Monitoring more unique traces gives VALG a higher chance to find all spec violations. Online hyperparameter tuning reduces VALG's missed unique traces to about 4.9%, but online tuning takes an average of 1.6 days per project. So, Kim et al. [48] developed faster offline tuning, which takes an average of 4.9 minutes per project.

To build on these results, a study is needed to understand VALG's current limitations, and find opportunities and insights for improving it. Yet, prior work did not address five important points.

(1) Kim et al. [47, 48] do not compare VALG's time vs. running tests without RV or theoretical RV that monitors only unique traces, so the room to speed up VALG w.r.t. these baselines is unknown.
(2) There was no investigation of where VALG spends its time, towards knowing how to speed it up.
(3) There was no analysis of code locations where VALG misses unique traces or monitors redundant ones, to find insights for improving VALG on these metrics where it does well but is not perfect.
(4) The impact of test non-determinism and RL stochasticity on VALG was not studied in depth.
(5) There was no empirical comparison of multiple RL algorithms with the one used in VALG.

Our work. We conduct an in-depth study of VALG to address these gaps, using 93 open-source Java projects: 58 are from VALG's prior evaluation [47, 48] (to ease comparison) and 35 are new. We take five steps. (i) We compare VALG's end-to-end time vs. running tests without RV. (ii) We approximate a theoretically optimal baseline that monitors only unique traces. To do so, we create a prototype that records *time series* [85] per location in one run to find unique traces monitored per time step. Then, in a second run that we use as the baseline, our prototype creates monitors only at time steps with unique traces. We compare our prototype's time and monitored redundant traces with VALG's. (iii) We run a profiler on VALG to record how long it spends on instrumentation, monitoring, RL, etc. (iv) We analyze 100 code locations where VALG misses unique traces or monitors redundant ones. (v) We survey the literature for RL algorithms and compare two with VALG's.

Results. We find plenty of room to improve the use of RL for speeding up RV, and insights for doing so. VALG is up to 323.8x (or, 2.7 hours) and 380.7x (or, 3.2 hours) slower than running tests without RV when applied to JAVAMOP and TRACEMOP, respectively. Also, VALG is up to 4.8 minutes (for JAVAMOP) and 42.6 minutes (for TRACEMOP) slower than our optimal prototype. So, future work to speed up VALG would be worthwhile. Three findings could be exploited. First, our optimal baseline still monitors 34% of VALG's 144,632,331 redundant traces because the underlying algorithm [18, 19] was not optimized using the findings that inspired VALG. Future work should do that optimization to speed up VALG. Second, future work should reduce the other 66% of redundant traces, e.g., via better RL reward mechanisms. Third, our profiling shows that 21.3% (or, 1.3 hours) of VALG overhead is

```
1 List_UnsafeListIterator(List l, ListIterator i) {
2 creation event create after(List l)
3     returning(ListIterator i) :
4     call(ListIterator List+.listIterator()) &&
5     target(l) {}
6 event modify before(List l) : (
7     call(* Collection+.add*(..))
8     call(* Collection+.clear(..))
9     call(* Collection+.remove*(..))
10    call(* Collection+.retain*(..))
11 ) && target(l) {}
12 event useiter before(ListIterator i) : (
13    call(* Iterator+.hasNext(..))
14    call(* ListIterator+.hasPrevious(..))
15    call(* Iterator+.next(..))
16    call(* ListIterator+.previous(..))
17    call(* ListIterator+.nextIndex(..))
18    call(* ListIterator+.previousIndex(..))
19 ) && target(i) {}
20 ere : create useiter* modify+ useiter }
```

(a) `List_UnsafeListIterator` spec

```
262 private TrieMap<V> addChild(String key, V value, ..) {
263   TrieMap<V> node = new TrieMap<V>();
264   node.sym = key.substring(this.keyLength());
265   node.value = value;
266   if (this.ch.isEmpty()) {
267     this.ch.add(node);
268   } else {
269     ListIterator<..> sibs = this.ch.listIterator();
270     while (sibs.hasNext()) {
271       TrieMap<V> sib = sibs.next();
272       if (sib.sym().charAt(0) > node.sym.charAt(0)) {
273         sibs.previous();
274         sibs.add(node);
275         break;
276       } else if (!sibs.hasNext()) {
277         sibs.add(node);
278         break;
279       } } }
280   return node; }
```

(b) Code snippet from `TrieMap.java`

Fig. 1. Example spec and monitored code.

spent deciding whether to dispatch events to monitors. Future work on designing more efficient data structures can greatly reduce this portion of Valg's overhead.

88.6% of monitor-creation locations yield at most five unique traces and 86.7% of unique traces come from only 0.1% of these locations. So, future work could *selectively* apply selective monitoring to only locations with many unique traces.

Test non-determinism and the stochasticity of RL algorithms can cause unique traces monitored to vary across runs by an average of 42.5 percentage points (pp) (max: 98pp) and 12.3pp (max: 87.4pp) when applied to JavaMOP and TraceMOP, respectively. Valg sometimes monitors more unique traces in subsequent runs than during a first run, but it sometimes monitors fewer. Work is needed to tame the negative impact of test non-determinism when RL is applied to RV.

We inspect the (i) top 50 locations with most monitored redundant traces, and (ii) top 50 locations with most missed unique traces. All 45,070,520 redundant traces in (i) occur when Valg's heuristic convergence logic fails. 42 locations in (ii) are affected by Valg's wrong assumption that unique traces are consecutive: only 58.7% of 2,018,320 adjacent pairs of unique traces are consecutive.

Addressing these challenges requires more than using other off-the-shelf RL algorithms. Of 11 RL algorithms (§3.4) that we survey, only two—D-UCB [23] and DS-TS [69]—apply to RV in CI. We implement both and compare them with the RL algorithm used in Valg. D-UCB and DS-TS are up to 8.3x and 3x slower, and monitor up to 81 percentage points (pp) and 94.3pp fewer unique traces than Valg, respectively. So, improving Valg from the RL side may require custom algorithms.

Contributions. This paper makes the following contributions:

- ⋆ We conduct the first in-depth study of Valg, to understand the current limits of using RL to speed up RV during testing, and how to improve it.
- ⋆ We identify reasons why Valg is suboptimal in terms of speedups that it provides, redundant traces that it monitors, and unique traces that it misses.
- ⋆ We discuss exciting future research directions for improving the use of RL during RV of tests.

Our artifacts and data are publicly available at: https://github.com/SoftEngResearch/rl-rv-study.

2 Running Example and Background

We first explain runtime verification (RV) using example code and spec. Then, we present a running example on how Valg speeds up RV that we use in §4 to illustrate findings per research question.

2.1 An Example Spec, how RV Monitors it, and Redundant Monitoring

Figure 1a shows spec `List_UnsafeListIterator`; it checks for stale views that can occur if a `ListIterator` *li* is used after the `List` from which *li* was obtained is modified. `ListIterator` should

fail fast and throw a `ConcurrentModificationException` in such cases, but that fail-fast behavior is not guaranteed [20]. The definition of `List_UnsafeListIterator` stipulates that an instrumented program should signal (i) a monitor-*creation* event (`create` on lines 2–5) after *li* is obtained via `listIterator()`; (ii) a `modify` event (line 6) before a method on lines 7–10 is called to modify a `List`; and (iii) a `useiter` event (line 12) before *li* is used via a method on lines 13–18. The formal property is a regular expression on line 20; it is violated if a `useiter` event is signaled after one or more `modify` events, in which case RV prints a violation message to aid debugging. Although our example spec uses a regular expression, all RV tools in this paper support several other formalisms, e.g., linear temporal logic (LTL), finite state machines (FSM), and context-free grammars (CFG).

Figure 1b shows code that we simplify from `JaroWinklerSimilarity` [41], where RV creates `List_UnsafeListIterator` monitors. That code is part of an implementation that uses a `TrieMap` to check string similarity. The `addChild` method adds a new node to a `TrieMap`, after traversing sibling nodes to find the right location to add the new node. RV creates a new monitor when a `ListIterator` object, `sibs`, is obtained on line 269. A `useiter` event is signaled every time `hasNext()` is called on `sibs` to check if a sibling node exists (line 270) or when a reference to another sibling node is obtained (lines 271 and 273). Lastly, a `modify` event is signaled every time a node is added to a sibling (lines 274 and 277). The `addChild` method is called from multiple locations and tests, causing RV to create *3,558,968* monitors on line 269. The problem is that, 99.9% (3,555,000) of these monitors are *redundant*; they check the same trace as the other 0.1% (3,968) *unique* ones. Redundant monitors only incur runtime overhead; they do not help find bugs. Redundant monitors are common: a recent study [28] found that over 99.8% of monitors in 1,544 projects are redundant.

2.2 Background on how Valg Works

Motivated by findings about redundant monitors, Valg was recently proposed [47, 48] to use reinforcement learning (RL) for speeding up RV. Valg attaches an RL agent to each monitor-creation location, e.g., line 269 in Figure 1b. Then, at each *time step* t, i.e., when RV is about to create a monitor, that agent takes one of two *actions* $\mathcal{A} = \{\texttt{create}, \texttt{ncreate}\}$. For `create`, the agent creates a new monitor and receives a binary *reward* R_t: a positive reward 1 if the created monitor checks a unique trace, or a zero reward if it checks a redundant one. For `ncreate`, no monitor is created, and the agent receives a continuous reward based on the ratio of all previously created monitors that checked redundant traces. This ratio estimates how likely a monitor would have observed a redundant trace if it was created.

Action-value method. Based on its formulation of selective monitor creation, Valg decides which action ($A_t \in \mathcal{A}$) to take at each time step t in two phases. First, when an RL agent receives reward R_{t-1}, it updates *values*—estimated rewards—for each action $\{Q_t(\texttt{create}), Q_t(\texttt{ncreate})\}$ using the exponential recency-weighted average (ERWA) method [79]. Then, the agent chooses the action with the greater value (exploitation) or it randomly selects an action based on the epsilon-greedy strategy [79] (exploration). The learning rate α and exploration probability ϵ are *hyperparameters* that determine how much to weigh recent rewards and how frequently to explore, respectively. Valg uses two other hyperparameters: a convergence threshold δ to decide when to stop learning and initial action values $\{Q_0(\texttt{create}), Q_0(\texttt{ncreate})\}$. Since Valg performs selective monitor creation in a non-stationary environment (§3.4) where convergence is theoretically infeasible [79], Valg uses a heuristic to decide convergence at time step t:

$$|1 - |Q_t(\texttt{create}) - Q_t(\texttt{ncreate})|| < \delta,$$

This heuristic is motivated by how rewards are defined in Valg. As Valg observes more redundant traces, the rewards for `ncreate` and `create` approach 1 and 0, respectively; conversely, they approach 0 and 1 for unique traces. [47] explains more on how Valg works.

Hyperparameter tuning. With default hyperparameters, VALG monitored only 76.7% of unique traces on average, so it could have missed spec violations. Hyperparameter tuning iteratively samples and tries different hyperparameters to achieve a single-objective of monitoring more unique traces. We do not use multiple objectives when tuning VALG because prior work [47] showed that single-objective tuning yields more unique traces with marginal overhead increase. *Online* tuning improved these unique-trace monitoring rates, but took 1.6 days per project, on average [47]. Recently, faster *offline* tuning was implemented in VALG; it takes 4.9 minutes per project on average [48]. Online and offline tuning use the Tree-structured Parzen Estimator (TPE) [2], and VALG with tuned hyperparameters monitored 95.1% unique traces on average [47, 48].

Trajectories. VALG's offline tuning uses the *trajectory* of RL agents from one RV run. Formally, a trajectory for spec s at monitor-creation location l, $\mathcal{T}_{s,l} = [\sigma_0,\ \sigma_1,\ \ldots,\ \sigma_n]$, is a sequence of steps. Each step σ_t corresponds to a monitor-creation decision made at time step t, defined as:

$$\sigma_t \doteq \langle t : A_t \in \mathcal{A},\ R_t,\ Q_t(\texttt{create}),\ Q_t(\texttt{ncreate})\rangle \tag{1}$$

where t is a time step, $A_t \in \mathcal{A} = \{\texttt{create}, \texttt{ncreate}\}$ is the action taken, R_t is the observed reward, and $Q_t(\texttt{create})$, $Q_t(\texttt{ncreate})$ are the action values. A trajectory only contains partial trace information for `ncreate` actions, so offline tuning first approximates a complete sequence of traces using cumulative decayed weights. Formally, let $\{(t_i, x_i)\}$ be the observed set of partial traces and their uniqueness so far at a monitor-creation location, where $x_i = 1$ if the trace checked by a monitor created at t_i is unique and $x_i = 0$ if that trace is redundant. Given decay rate $\lambda > 0$, the cumulative decayed weight for a trace with value $x \in \{0, 1\}$ at time step t is computed as:

$$w_x(t) \doteq \sum_{(t_i, x_i = x),\ t_i < t} e^{-\lambda(t - t_i)}. \tag{2}$$

Using cumulative decayed weights, offline tuning approximates values $\hat{x}_t$ for each time step t with a `ncreate` action, i.e., $\hat{x}_t = \arg\max_{x \in \{0,1\}} w_x(t)$. Then, offline tuning simulates multiple runs of VALG on the approximated traces to find better hyperparameter values.

3 Methodology

Our study addresses five research questions:

- **RQ1:** How does VALG's end-to-end time compare with running tests without RV, and how do its time savings and traces monitored compare with a theoretically optimal baseline?
- **RQ2:** Where does VALG spend its time among its components?
- **RQ3:** What characterizes monitor-creation locations where VALG monitors many redundant traces or misses unique ones?
- **RQ4:** How much impact can test non-determinism and the stochasticity of RL algorithms have on VALG's monitored unique traces?
- **RQ5:** How do other RL algorithms perform compared with the one previously used in VALG?

3.1 Setup

Baselines. As RV baselines that do not use RL, we use JAVAMOP and TRACEMOP [81]. TRACEMOP stores monitored traces online while JAVAMOP does not store traces. We say "VALGJ" and "VALGT" to mean baselines obtained by applying VALG to JAVAMOP and TRACEMOP, respectively. Unless otherwise stated, we use the same default RL hyperparameters as in [47, 48]: $\langle \alpha = 0.9,\ \epsilon = 0.1,\ \delta = 1\text{e}{-4},\ Q_0 = (5.0,\ 0.0)\rangle$. To compare with other RL algorithms (§3.4), we first run a small experiment on five projects where VALG with default hyperparameters misses 66.9% of unique traces on average from prior work [47]. Then, we run other RL algorithms on those five projects using 5 to 7 different hyperparameter sets, obtained by evenly partitioning the value range of each hyperparameter. Lastly,

we use the hyperparameter set that works best on these five projects (in terms of time savings and unique traces monitored) in the rest of our experiments: $\langle \gamma = 0.1,\ c = 2.0,\ \delta = 1e{-}4,\ Q_0 = (5.0,\ 0.0)\rangle$ (D-UCB) and $\langle \gamma = 0.5, \delta = 1e{-}4,\ Q_0 = (5.0,\ 0.0)\rangle$ (DS-TS). No tuning is performed for other RL algorithms, since we compare them with VALG with default hyperparameters. For experiments on tuning, we reuse the online-tuned hyperparameters from prior work [47] for 55 projects. The hyperparameters are not provided for three projects [47], and the remaining projects are new ones in the evaluation of this paper; online tuning for them is infeasible for us due to compute and monetary costs [47]. We also run offline tuning with 100 trials for the same set of 55 projects.

Table 1. Statistics about 93 projects in our evaluation: no. of tests (#tests), test time in seconds w/o RV (t), lines of code (LoC), statement coverage % (cov^s), branch coverage % (cov^b), no. of commits (#SHAs), years since first commit (age), and no. of stars (#★).

	#tests	t	LoC	cov^s	cov^b	#SHAs	age	#★
Mean	124	39.9	10,674	61.6	55.3	358	10.7	330
Med	41	10.7	4,004	67.7	56.9	116	10.8	51
Min	1	0.2	190	0.8	4.7	3	3.1	6
Max	1,423	1,565.5	92,687	95.4	92.2	4,643	22.0	5,058
Sum	11,564	3,715.2	992,714	n/a	n/a	33,385	n/a	30,752

Evaluation subjects and specs. We use 93 projects that we select from 99: 64 from prior VALG evaluation [47, 48] and 35 from other work on RV [28, 52]. We start with these 99 projects because monitoring incurs greater overhead than all other components in them. We exclude 6 of 99 projects: one is no longer publicly available [38], and time-series collection (§3.3) and VALGT runs out of memory or times out after three days for five projects. (TRACEMOP fails on these projects for the same reasons.) In all experiments, we use the 160 specs [51, 58, 70] of correct JDK API usage protocols used in prior work on RV of tests [27–29, 42, 47, 48, 61]. Table 1 shows summary statistics about the 93 projects: the caption explains the columns and "n/a" are meaningless sums.

Running Experiments. We use a machine with an AMD EPYC Genoa processor, 192 physical (384 CPU) cores, and 1.55 TB of RAM. Our experiments are parallelized via Slurm; we allocate 32 CPU cores and 125 GB of RAM for JAVAMOP and VALGJ, and 32 CPU cores and 503 GB of RAM for TRACEMOP, VALGT, and time-series collection. All experiments are run in Apptainer with Ubuntu 20.04.6 LTS and Java 8. We write scripts to run experiments and analyze results. For test non-determinism evaluation in RQ4, we fix test-execution order by setting the same seed in Maven.

3.2 Study Design

In **RQ1**, compare VALG's end-to-end times with those of running tests without RV on all 93 projects. We also compare VALG's time and unique traces monitored (using default, online-tuned, and offline-tuned hyperparameters) with those of our baseline that aims to monitor only unique traces. Trace comparison is done only on 55 projects where online-tuned hyperparameters are provided from prior work [47]. In **RQ2**, we profile the execution of JAVAMOP, TRACEMOP, and their VALG variants (VALGJ and VALGT) for all 93 projects. In **RQ3**, we first collect time series for all monitor-creation locations in all 93 projects and quantitatively analyze them to identify connections between monitor-creation locations and unique traces. Then, we manually analyze 100 locations to identify reasons why VALG monitors many redundant traces or misses unique ones. In **RQ4**, to evaluate the potential impact of test non-determinism, we collect time series from three RV runs in all 93 projects and quantify inconsistencies across runs. Then, to evaluate the impact of RL stochasticity, we simulate VALG with default hyperparameters three times on a fixed time series and quantify variance in unique traces monitored across simulations. Our design decouples the effect of test non-determinism when evaluating the impact of RL stochasticity, and vice versa. In **RQ5**, we implement two off-the-shelf RL algorithms in VALG and compare their end-to-end execution times and unique traces monitored with those of the RL algorithm implemented in VALG.

These RQs are designed to understand VALG and the use of RL for RV. **RQ1** and **RQ2** aim to identify the degree and root causes of runtime overhead. **RQ3** and **RQ4** aim to find insights for

improving Valg in terms of monitoring more unique traces. **RQ5** addresses how other off-the-shelf RL algorithms may work for RV in CI.

3.3 Prototype for Approximating a Theoretically Optimal RV

It is valid to ask how far Valg is from a theoretically optimal baseline that only monitors unique traces (the second part of RQ1). To approximate this baseline, we first extend TraceMOP to collect *time series* for all monitor-creation locations. A time series for monitor-creation location (s, l) where s is spec and l is program location, denoted as $\mathcal{S}_{s,l} = [\tau_0, \tau_1, \ldots, \tau_n]$, is a sequence of binary data points τ_t, each of which corresponds to whether the trace checked by a monitor created at time step t is unique or redundant. τ_t is defined as:

$$\tau_t \doteq \begin{cases} 1, & \text{if a unique trace is checked by a monitor created at time step } t, \\ 0, & \text{if a redundant trace is checked by a monitor created at time step } t. \end{cases} \tag{3}$$

For efficiency, our extension (i) re-uses the TraceMOP's data structure for trace storage [81]; (ii) encodes a unique and redundant trace as 1 and 0, respectively; (iii) compresses consecutive data points that are the same, e.g., `0x100` for 100 redundant traces checked consecutively; and (iv) supports collecting time series *without* collecting traces. These optimizations make TraceMOP with time-series collection (but without trace collection) 1.63x faster on average than TraceMOP. Our extension saves time series per JVM, so it works even if each test spawns a new JVM.

Our prototype to approximate this baseline uses time-series collection and works in two steps. A first run collects time series. Then, in a second run, at every monitor-creation location, RV creates monitors only for time steps where τ_t is 1. For example, in Figure 1b, the time series $\mathcal{S}_{s,l}$ is $[1, \texttt{0x6}, 1, \texttt{0x15}, 1\texttt{x5}, \ldots]$, where $s =$ `List_UnsafeListIterator` and $l =$ `TrieMap.java : L276`. So, our prototype creates monitors only for the time steps $t \in [0, 7, 23, 24, 25, 26, 27, \ldots]$. Assuming that the monitored program is deterministic, the second run in our prototype should only monitor unique traces and monitor no redundant ones.

3.4 Selecting other RL algorithms

We review the literature to identify widely-used RL algorithms that could work for RV in CI. We identify 11 [1, 16, 24, 25, 31, 45, 49, 62, 79, 80, 83, 84] besides ERWA and the epsilon-greedy based algorithm [49, 79] currently used in Valg. Then, we select two of them based on three criteria:

- **Non-stationary reward distributions.** In Valg's formulation of selective monitor creation [47], reward distributions for the two actions can change at each time step t. For the `create` action, the success probability of the Bernoulli distribution [65, 72] alternates between 0 and 1. For the `ncreate` action, the degenerate distribution [65] changes at every time step t when a monitor is created that checks a unique trace (§2.2 gives details). RL algorithms should be able to adjust to these non-stationary changes by prioritizing recent rewards [9, 46, 57].
- **Non-deterministic trace occurrence.** Occurrences of unique and redundant traces are non-deterministic: a unique or redundant trace can be checked by a monitor created at time t regardless of the uniqueness or redundancy of traces checked at previous time steps. So, an algorithm should enable exploration. But, Kim et al. [47] observed empirically that unique traces tend to follow a unique trace in the time series, and the same observation held for redundant traces. As a result, recent traces should have more influence on monitor-creation decisions.
- **Runtime overhead.** Using an RL algorithm to decide whether to create monitors incurs runtime overhead. To be useful for speeding up RV, an RL algorithm's runtime overhead must be less than time saved by selective monitor creation [47].

Among the 11 RL algorithms that we review, Q-learning [79, 80, 84], SARSA [79, 80], Markov Chain Monte Carlo (MCMC) [24, 25], and Bootstrapping [24, 25] all work for *stationary* reward

distributions. So, they cannot work for the non-stationary RV environment. Sample Average [79, 80] and Softmax [79, 80] either fail to capture the tendency of unique traces to follow unique traces or to enable exploration at different time steps. So, they cannot take advantage of the empirical observation underlying VALG. Using Sample Average could help monitor more unique traces when unique traces follow redundant ones. But, our preliminary experiments on 25 projects shows that Sample Average is much slower than VALG, so we omit it from our study. (Detailed results are in our artifacts.) Deep Q-Network [31, 62, 83], Proximal Policy Optimization (PPO) [79, 83], and Soft Actor–Critic (SAC) [31, 83] incur high computational overhead because they perform expensive updates of model parameters at each time step. For all these reasons, we do not implement any of these algorithms in VALG.

The two algorithms that we find could work for RV are Upper Confidence Bound (UCB) [16, 49, 80] and Thompson Sampling (TS) [1, 16, 49]. But, both algorithms assume stationary reward distributions. So, we use their *discounted* variants for *non-stationary* reward distributions: Discounted Upper Confidence Bound (D-UCB) [23] and Discounted Thompson Sampling (DS-TS) [69].

3.4.1 Discounted Upper Confidence Bound (D-UCB). Upper Confidence Bound (UCB) enables exploration by rewarding under-explored actions. We select UCB as it is computationally efficient, and UCB's exploration may capture non-deterministic occurrences of unique traces. The standard UCB algorithm [6] for stationary environments, where the reward distributions do not change over time, uses 2 (for c in $c \ln t$) as its exploration constant and is defined as:

$$\text{UCB1}_a(t) = \hat{\mu}_a(t) + \sqrt{\frac{2 \ln t}{N_a(t)}}, \qquad \hat{\mu}_a(t) = \frac{\sum_{s=1}^{t} r_s \mathbf{1}[a_s = a]}{N_a(t)}, \qquad N_a(t) = \sum_{s=1}^{t} \mathbf{1}[a_s = a], \quad (4)$$

where r_s is the reward at time step s and t is the current time step. UCB keeps an *index* $\text{UCB1}_a(t)$ for each action a, computed as the sum of an estimated reward $\hat{\mu}_a(t)$ and a *confidence term* (the second right-hand side term in $\text{UCB1}_a(t)$). The estimated reward is the average of rewards from past time steps, and the confidence term is greater for less frequently explored actions. The intuition is to prioritize the action with a greater estimated reward, while providing a *bonus* to under-explored actions to encourage exploration. But, our selective monitor creation problem is non-stationary where reward distributions can change at every time step, so we use D-UCB, defined as:

$$\text{D-UCB}_a(t) = \hat{\mu}_a(t) + c\sqrt{\frac{\ln t}{N_a^\gamma(t)}}, \ \hat{\mu}_a(t) = \frac{\sum_{s=1}^{t} \gamma^{t-s} r_s \mathbf{1}[a_s = a]}{N_a^\gamma(t)}, \ N_a^\gamma(t) = \sum_{s=1}^{t} \gamma^{t-s} \mathbf{1}[a_s = a], \quad (5)$$

where the exploration constant $c > 0$ controls the confidence term and the discount factor $\gamma \in (0, 1)$ determines how much to weigh recent rewards; the reward r_s and the pull count $N_a^\gamma(t)$ decay for older time steps. For efficiency, we use an incremental method to update estimated rewards, and learning converges when $|\hat{\mu}_{a_1}(t) - \hat{\mu}_{a_2}(t)| < \delta$, i.e., when the estimated reward for an action reaches 1 and the reward of the other action reaches 0:

$$Q_a^\gamma(t) = \gamma Q_a^\gamma(t-1) + r_t \mathbf{1}[a_t = a], \quad N_a^\gamma(t) = \gamma N_a^\gamma(t-1) + \mathbf{1}[a_t = a], \quad \hat{\mu}_a(t) = \frac{Q_a^\gamma(t)}{N_a^\gamma(t)} \quad (6)$$

3.4.2 Discounted Thompson Sampling (DS-TS). Thompson Sampling keeps a posterior distribution over expected rewards for each action and updates that distribution each time a reward is received. We select DS-TS because it performs *uncertainty-aware* exploration, where the RL agent explores less frequently as observations accumulate. Such exploration may work for RV, where most traces are redundant and reducing exploration for locations with many redundant traces can further speed up RV. We use the Beta distribution [40] because rewards in VALG are in $[0, 1]$ and naturally fit $\text{Beta}(\alpha_a, \beta_a)$; α_a and β_a are two hyperparameters of action a's Beta distribution. At each time step,

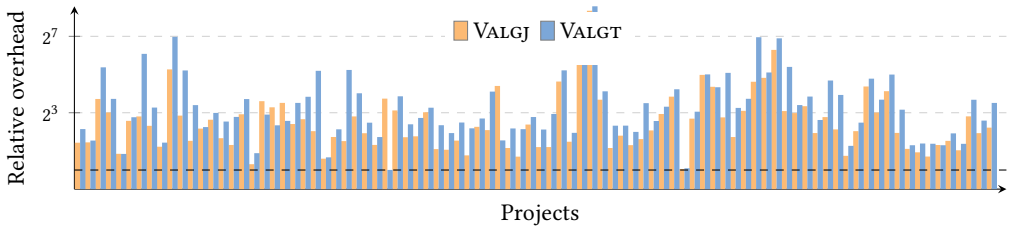


Fig. 2. Overhead of RV (bars) relative to running tests only (dashed line). The y-axis uses a $\log_2$ scale.

Thompson Sampling samples $\theta_a \sim \text{Beta}(\alpha_a, \beta_a)$ for all actions and selects $a_t = \arg\max_a \theta_a$. Then, when a reward r_t is received, Thompson Sampling updates the posterior of the chosen action using

$$\alpha_{a_t} \leftarrow \alpha_{a_t} + r_t, \qquad \beta_{a_t} \leftarrow \beta_{a_t} + (1 - r_t), \qquad \hat{\mu}_a(t) = \frac{\alpha_a}{\alpha_a + \beta_a} \tag{7}$$

Similar to UCB, Thompson Sampling assumes stationary environments. So, we use a discounted variant, DS-TS, where α and β decay based on a discount factor $\gamma \in (0, 1)$:

$$\alpha'_a = (1 - \gamma)\alpha_a, \qquad \beta'_a = (1 - \gamma)\beta_a, \quad \forall a \tag{8}$$

Then, DS-TS follows the same procedure as Thompson Sampling: sampling $\theta_a \sim \text{Beta}(\alpha'_a, \beta'_a)$, selecting $a_t = \arg\max_a \theta_a$, and updating the posterior of the chosen action when a reward r_t is received (Equation. 9). Similar to D-UCB, learning converges when $|\hat{\mu}_{a_1}(t) - \hat{\mu}_{a_2}(t)| < \delta$.

$$\alpha_{a_t} \leftarrow \alpha'_{a_t} + r_t, \quad \beta_{a_t} \leftarrow \beta'_{a_t} + (1 - r_t), \quad \hat{\mu}_a(t) \text{ is the same as in Equation 7} \tag{9}$$

4 Results

4.1 RQ1: How optimal is Valg in terms of time overhead and unique traces monitored?

End-to-end time. Figure 2 shows the relative overhead of Valg vs. running tests without RV, computed as Valg's end-to-end time divided by time to run tests without RV. The dashed line at 1 on the y-axis represents time to run tests without RV, and bars represent relative overheads of ValgJ (orange) and ValgT (blue). Given the wide variance in relative overheads, for ease of presentation, the y-axis uses a $\log_2$ scale. We can see that, *despite the speedups that Valg provides, it is often still much slower than running tests without RV*. On average, ValgJ and ValgT are respectively 13.9x and 22.2x slower than running tests without RV. In the worst case, ValgJ and ValgT have overheads of 323.8x and 380.7x, taking 2.7 and 3.2 more hours than running tests without RV, respectively. In the project where Valg takes 3.2 hours, running tests without RV takes only 30.1 seconds.

Compared with our theoretically optimal baseline (not shown), ValgJ and ValgT are on average (max) 1.2x (6.6x) and 1.2x (9.6x) slower, respectively. In absolute terms, ValgJ and ValgT are up to 4.8 and 42.6 minutes slower than our baseline, respectively. In this worst case, ValgT takes 53.4 minutes for a project where our prototype takes 10.9 minutes. Overall, Valg has considerable room for improvement vs. running tests without RV and our theoretically optimal baseline.

Monitored unique traces. Valg finds 99.6% of all spec violations in prior work [47] and 99.5% (801/805) in our evaluation. So, we subsequently focus only on unique traces monitored, which is a stronger criterion for evaluating RV's bug detection capability than spec violations: monitoring all unique traces guarantees finding all spec violations, but not vice versa. Gray bars in Figure 3 show ratio of unique traces monitored by ValgT to those monitored by TraceMOP per project with default hyperparameters. The average ratio of unique traces monitored with default hyperparameters is 74%, which is similar to what Kim et al. find [47]. But, the leftmost bars in Figure 3 show that Valg misses most unique traces in 17 projects (where the ratio is less than the average).

The orange and green lines in Figure 3 show the ratios after online and offline hyperparameter

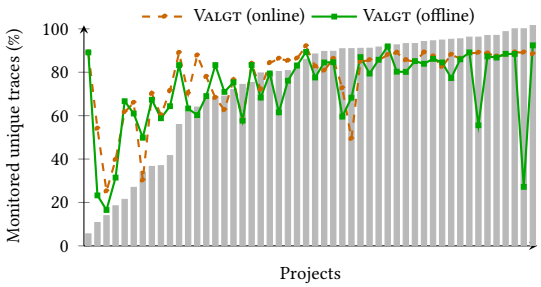


Fig. 3. Ratio of unique traces monitored per project.

tuning, respectively. No prior work performed this comparison. Online tuning improves the unique trace ratio to 86.7%, while offline tuning improves it to 81.4%. Considering the sum of unique traces monitored by TRACEMOP in all these 55 projects, VALG monitors 35.8% (default), 89.1% (online), and 86.2% (offline). The small difference in ratios of monitored unique traces between online and offline tuning is due to inherent imprecision in offline tuning (see §2.2). Online tuning helps monitor a marginally higher ratio of unique traces, but Figure 3 also shows that offline tuning produces equal or better ratio than online tuning in 22 projects. Also, the lower aggregate ratio from offline tuning is mostly due to 5 projects (with difference greater than 25pp) where it performs much worse than online tuning. More worrisome, VALG checks *fewer* unique traces after tuning than without tuning in 16 of the 55 projects due to RL stochasticity (see §4.4) or inherent imprecision of offline tuning (see §2.2). For example, in a project where VALG monitors all 21,495 unique traces with default hyperparameters, VALG monitors only 30.5% after offline tuning. Overall, offline tuning yields a slightly lower monitored unique trace ratio than online tuning. But, offline tuning is much faster than online tuning (470.2x on average per project). So, offline tuning is a valuable alternative to use in practice.

Redundant trace elimination. VALG monitors 93.5% fewer redundant traces [47], and our theoretically optimal baseline is designed to only monitor unique traces. But, that baseline still monitors 34% of VALG's redundant traces. One reason is that, for specs where the first event does not bind all its parameters, the monitoring algorithm used by all RV techniques in this paper [18] creates *partially-bound monitors* from which future monitors can be cloned. For such specs, VALG always creates partially-bound monitors. That is, VALG does not perform selective decision making for such monitors to allow future monitors to be cloned from them. Future work could explore selectively creating these partially-bound monitors to further speed up VALG. Another reason for the theoretically optimal baseline's monitoring redundant traces is test non-determinism between different RV runs (see §4.4). Improving our baseline's precision can be addressed as part of future work on reducing the impact of test nondeterminism when using RL to speed up RV.

Running example. For the `JaroWinklerSimilarity` project in §2, VALG is still 7x (VALGJ) and 16.2x (VALGT) slower than running tests without RV. In absolute terms, running tests without RV takes 28.1 seconds, but VALGJ and VALGT take 196.5 and 455.1 seconds, respectively. Using default hyperparameters, VALGT monitors only 64% of unique traces. After online tuning, VALGT monitors 98.4% of unique traces, but that tuning is very costly. Surprisingly, VALG with online-tuned hyperparameters monitors 7.6x more redundant traces than VALG with default hyperparameters, showing that online tuning benefits may come at the cost of monitoring more redundant traces. With offline tuning, VALG monitors only 67.6% of unique traces and 7.2x more redundant traces than VALG. These observations motivate deeper investigations that we discuss in §4.2 and §4.3.

4.2 RQ2: Where does VALG spend its time?

Procedure. We attach a profiler [5] to RV runs and use a script to post-process how much time is spent on (i) *AspectJ*: instrumentation time; (ii) *Monitoring*: monitoring time; (iii) *Signaling*: time for deciding whether to dispatch signaled events to monitors; (iv) *Project*: time spent in project code and tests; (v) *RL*: time for RL agents' learning; and (vi) others.

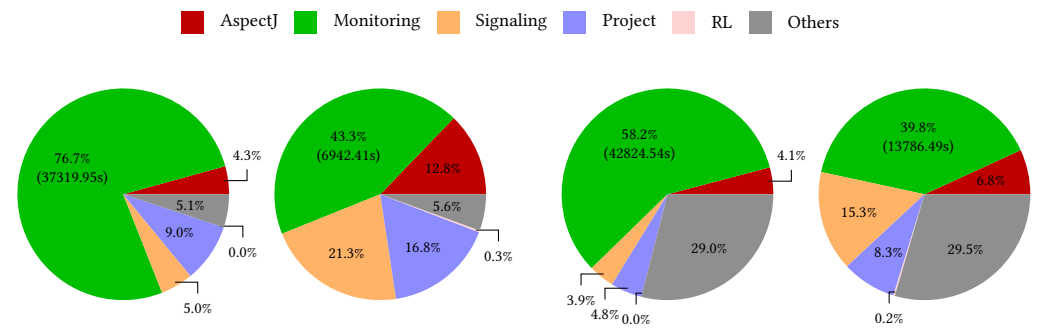


Fig. 4. Profiling results of JavaMOP and ValgJ (left) and TraceMOP and ValgT (right).

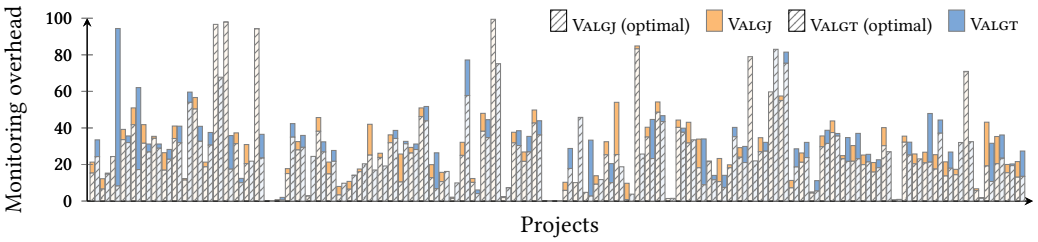


Fig. 5. Difference in monitoring overhead of Valg (solid bars) and the optimal baseline (dashed bars).

Valg vs. JavaMOP and TraceMOP. Figure 4 shows cumulative profiling results across all projects for JavaMOP, ValgJ, TraceMOP, and ValgT, in order from left to right. The results are obtained as $((x/y) * 100)$ where x is the total time spent in each component across all projects and y is the total execution time across all projects. We show the absolute times for monitoring to make it easier to compute times for other portions.

In Figure 4, we see why Valg speeds up RV: it spends 5.4x and 3.1x less time on monitoring than JavaMOP and TraceMOP, respectively. But, we also observe two interesting findings: (i) monitoring is still a significant portion of ValgJ and ValgT overhead at 43.3% and 39.8%, respectively; and (ii) ValgJ and ValgT respectively spend 21.3% and 15.3% of their overhead on making decisions about event signaling, i.e., dispatching events to monitors, in ValgJ and ValgT, respectively. On average among projects, Valg spends 30.5% and 29% of its time on monitoring, and 18.4% and 17.2% of its time on event signaling for ValgJ and ValgT, respectively. These portions of signaling overheads are 4.3x and 3.9x more than those in JavaMOP and TraceMOP, respectively. The reason is that JavaMOP and TraceMOP always signal events to monitors, so no checks are needed. But, Valg selectively signals events only when the monitors (to which the events would be dispatched) are created, and that selectivity incurs overhead. In the worst cases, Valg's monitoring and event signaling incur 95.4% and 97.8% of overheads, or 8.9 and 17.8 minutes, respectively. Overall, Valg's monitoring and event signaling overheads should be reduced to bring its speed closer to running tests without RV.

Valg vs. theoretically optimal baseline. In Figure 5, the solid and dashed bars represent the monitoring overhead (not end-to-end time) of Valg and our theoretically optimal baseline, respectively, for all 93 projects. Each bar's value is computed as $((x/y) * 100)$, where x is the project's monitoring time and y is the end-to-end RV time for the project. Stacked solid bars show the additional monitoring overhead incurred by Valg compared with the optimal baseline (no stacked bars are shown if there is none). ValgJ and ValgT spend 2.2x (69x) and 3.3x (142.4x) more time

on average (max) on monitoring than our baseline, respectively. In absolute terms, Valg spends up to 59.7 (ValgJ) and 23.9 (ValgT) more minutes on monitoring than our prototype. Notably, our baseline has observable monitoring overhead, caused by redundant traces it still monitors. (Test non-determinism may also cause this baseline to miss unique traces and check redundant ones; see §3.3). These comparisons imply that Valg has room for improvement by monitoring less redundant traces. More importantly, the optimal baseline's monitoring overhead is still high in many projects, e.g., more than 40% in 18 projects. So, an approximately correct RV approach, such as selective creation of partially-bound monitors, could be considered.

Running example. For the `JaroWinklerSimilarity` project in §2, Valg spends 1.4x and 2.5x less time on monitoring than JavaMOP and TraceMOP, respectively. In absolute terms, ValgJ takes 123 seconds where JavaMOP takes 178, and ValgT takes 452 seconds where TraceMOP takes 1,148. But, monitoring overhead still dominates by 51% for ValgJ and 62.1% for ValgT. Our baseline further reduces this overhead by 1.5x and 18.9x compared to ValgJ and ValgT, respectively. In absolute terms, the baseline takes 79 seconds for ValgJ and 44 seconds for ValgT. Finally, Valg spends 27.5% of its overhead on event signaling in ValgJ and 16.5% in ValgT.

4.3 RQ3: What characterizes code locations where Valg is ineffective?

Valg is *ineffective* at locations where it misses unique traces or monitors many redundant ones.

Characterizing monitor-creation locations. To identify locations to manually analyze, we first quantitatively characterize relationships among monitor-creation locations and the number of unique traces checked by monitors created at those locations. To do so, we collect time series for all 43,600 monitor-creation locations in all 93 projects and write a script to find all time steps where a created monitor checked a unique trace. The distribution of unique traces over monitor-creation locations is skewed: (i) only one unique trace is monitored at 62.3% (27,184) of all monitor-creation locations; (ii) five or fewer unique traces are monitored in 88.6% (38,641) of locations; and (iii) 0.1% (39) of monitor-creation locations check 86.7% (2,474,545) of all 2,854,776 unique traces in our study. So, it seems that monitoring a small subset of locations could be sufficient for checking most unique traces—a direction that future research could investigate.

Selecting locations to analyze. We select 100 monitor-creation locations for manual analysis as follows. First, we collect time series and RL trajectories for all monitor-creation locations in all 93 projects. Let $\mathcal{T}_{s,l} = [\sigma_0, \ldots, \sigma_n]$ be the trajectory and $\mathcal{S}_{s,l} = [\tau_0, \ldots, \tau_n]$ be the time series for spec s at program location l, i.e., (s, l). We define two metrics that characterize Valg's effectiveness for (s, l) in n time steps. *RT*, the number of redundant traces monitored for (s, l) and *MT*, the number of missed unique traces for (s, l) are respectively defined as:

$$RT(s,l) \doteq \sum_{t=0}^{n} \mathbb{I}\left[\sigma_t.A_t = \texttt{create} \ \wedge \ \tau_t = 0\right], \quad MT(s,l) \doteq \sum_{t=0}^{n} \mathbb{I}\left[\sigma_t.A_t = \texttt{ncreate} \ \wedge \ \tau_t = 1\right] \quad (10)$$

Intuitively, $RT(s, l)$ counts the number of time steps where the RL agent for (s, l) creates monitors that check redundant traces, and $MT(s, l)$ counts the number of time steps where the agent does not create a monitor that would have checked a unique trace. Next, we select (i) the top 50 locations with the highest $RT(s, l)$, and (ii) the top 50 locations with the highest $MT(s, l)$. These 100 locations are from 22 projects, and contribute 67.8% (45,070,520) of all 66,445,016 redundant traces monitored by Valg, and 33% (664,513) of all 2,016,164 unique traces monitored by TraceMOP.

Main reasons for Valg's ineffectiveness. We find two main root causes of Valg's ineffectiveness.

(1) *Heuristic convergence logic.* Valg uses a heuristic to determine convergence, which is theoretically infeasible in non-stationary environments like RV [79] (§2.2). Interestingly, *all 45,070,520 redundant traces in the 50 locations we analyze are due to failures of this convergence heuristic.*

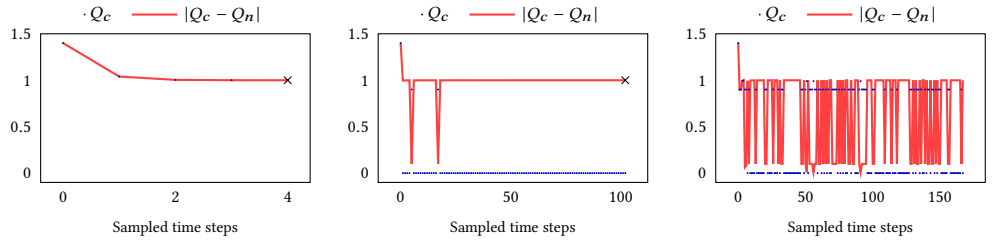


Fig. 6. Learning curves of example locations with many redundant traces.

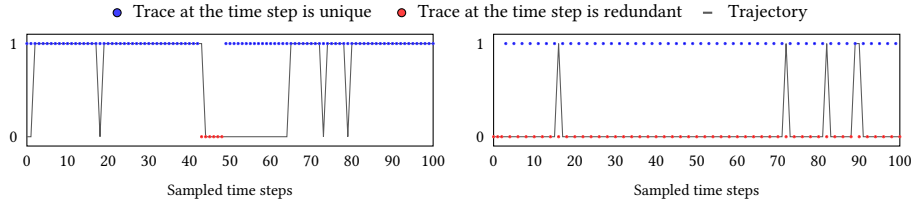


Fig. 7. RL agent's trajectory for example locations with many unique traces missed.

Specifically, VALG's action selection converges too early to `create` (35 locations), converges too late to `ncreate` (13 locations) or does not converge at all (2 locations) with VALG's heuristic.

Figure 6 shows learning curves—visualizations of how estimated action rewards and their absolute differences change over time steps—for three example locations, one per category. Q_c is 1 when VALG creates a monitor that checks a unique trace, and 0 otherwise. Red lines show the *absolute gap*—the absolute difference—between the actions' estimated rewards. Learning converges (marked with x in Figure 6) when $|Q_c - Q_n|$ becomes smaller than threshold δ. Each monitor-creation location often has a large number of time steps, so we only show up to 100 time steps with an interval of total number of steps divided by 100, and up to 100 time steps where $Q_c = 1$.

The leftmost pattern in Figure 6 occurs when unique traces are consecutively observed during initial time steps but redundant traces come later. In such cases, VALG's RL agent prematurely converges to `create` and creates monitors at all subsequent time steps, causing many redundant traces to be monitored. The middle pattern is the opposite: VALG's RL agent takes too long to converge to `ncreate`. So, many redundant traces are again monitored. Because of how VALG's reward for `ncreate` is designed (§2.2), a very long sequence of redundant trace observations is needed for VALG to converge to `ncreate`. The rightmost pattern occurs when VALG fails to converge due to intermittent occurrences of unique traces.

(2) *Wrong assumption about consecutive unique traces.* In 42 of 50 monitor-creation locations that we manually analyze, *VALG misses unique traces because it wrongly assumes that unique traces occur in consecutive time steps.* That is, VALG can miss unique traces that are separated by redundant ones. Figure 7 shows parts (100 time steps) of VALG's RL agent's trajectory for two locations. Red and blue circles respectively show time steps with value of 0 (a redundant trace was monitored) and 1 (a unique trace was monitored) in the time series. The line plots show the RL agents' trajectories, where 0 and 1 respectively represent `ncreate` and `create` actions. In the trajectory on the left, most unique traces appear consecutively, and VALG monitors 74 of 100 unique traces. But, VALG misses *all but one* of 100 unique traces in the trajectory on the right where adjacent unique traces are separated by redundant ones. In fact, VALG monitors only 14,255 out of 270,344 unique traces at this location. We next quantify how many unique traces can be potentially missed due to the

interleaving of unique and redundant traces. Figure 8 shows the proportion of "D1 pairs", which we use to refer to adjacent *and* consecutive pairs of unique traces, among all adjacent unique trace pairs. For example, let the observed trace sequence be $\omega_{u_1}\ \omega_{u_2}\ \omega_{r_1}\ \omega_{u_3}$, where ω_{u_1}, ω_{u_2}, and ω_{u_3} are unique traces and ω_{r_1} is a redundant trace. $(\omega_{u_1}, \omega_{u_2})$ is a D1 pair: the two unique traces are adjacent and there is no redundant trace between them. By contrast, $(\omega_{u_2}, \omega_{u_3})$ is an adjacent but not a D1 pair because ω_{r_1} prevents these two unique traces from being consecutive. Each point in the graph shows a project's ratio of unique traces monitored on the x-axis and the ratio of D1 pairs in y-axis. As Figure 8 shows, there is only a weak positive correlation between percentage of D1 pairs and unique traces monitored (Pearson's R: 0.1669). The correlation partly shows why VALG works, but future work must revisit this consecutiveness assumption.

Running example. For the monitor-creation location at line 269 in Figure 1b, VALG monitors 177,721 (of 3,555,000) redundant traces. Our manual analysis confirms that all 177,721 are caused by convergence failures: unique traces intermittently occur next to redundant ones at that location, making VALG fail to converge to either action. Also, only 18.6% of unique traces at that location are adjacent with no interleaving of redundant traces. So, VALG monitors only 238 of 3,968 unique traces at that location.

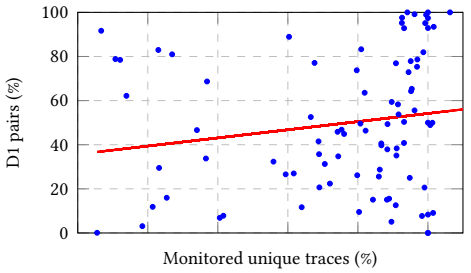


Fig. 8. Scatter plot: D1 pairs vs. unique trace ratio.

4.4 RQ4: How can test non-determinism and RL stochasticity impact VALG's results?

Evaluating the impact of test non-determinism. We evaluate the potential impact of test non-determinism by collecting time series from three runs of VALG and checking for inconsistencies. For each (s, l), i.e., spec s at monitor-creation location l, let $\mathcal{S}_{s,l}^{(i)} = [\tau_0^{(i)}, \ldots, \tau_{n_i}^{(i)}]$ denote the time series collected from the i-th run of VALG, where $i \in \{1, 2, 3\}$. Let $N_{s,l} \doteq \max\{n_1, n_2, n_3\}$, which is the maximum number of time steps observed across three runs; obtaining this number is needed because test non-determinism can cause the number of time steps to vary between runs. We first retain "active" time steps where at least one unique trace was checked by a monitor created at l ($\mathcal{T}_{s,l}^{\star}$) and define an inconsistency indicator $Inc_{s,l}(t)$ for each $t \in \mathcal{T}_{s,l}^{\star}$:

$$\mathcal{T}_{s,l}^{\star} \doteq \left\{ t \in [0, N_{s,l}] \;\middle|\; \exists i \in \{1..3\},\ \tau_t^{(i)} = 1 \right\},\ Inc_{s,l}(t) \doteq \mathbb{I}\left[\exists i, j \in \{1..3\},\ i \neq j : \tau_t^{(i)} \neq \tau_t^{(j)}\right] \quad (11)$$

Intuitively, inconsistency here means that RV monitors a unique trace at time step t in some runs, but monitors a redundant trace at t in other runs, so the three data points for time step t ($\tau_t^{(i)}$, $i \in \{1..3\}$), obtained for each time series, are inconsistent. Let $\mathcal{L}_p$ denote all (s, l) in project p. We define the impact of test non-determinism on p ($ND(p)$) as the ratio of the total number of inconsistent time steps across all locations to the total number of active time steps. Intuitively, a higher value of $ND(p)$ indicates that VALG has more inconsistencies and can therefore be more impacted by test non-determinism across monitor-creation locations in p:

$$ND(p) \doteq \frac{\sum_{(s,l)\in\mathcal{L}_p} \sum_{t\in\mathcal{T}_{s,l}^{\star}} Inc_{s,l}(t)}{\sum_{(s,l)\in\mathcal{L}_p} |\mathcal{T}_{s,l}^{\star}|}. \quad (12)$$

The heatmap in Figure 9 shows the impact of test non-determinism per project on the x-axis. The y-axis has 100 buckets per project. Each time step $t \in \mathcal{T}_{s,l}^{\star}$ such that $(s, l) \in \mathcal{L}_p$ in project p is mapped to one bucket, increasing the bucket's opacity if $Inc_{s,l}(t) = 1$. Each project has multiple monitor-creation locations and each location can have a large number of time steps in $\mathcal{T}_{s,l}^{\star}$. So, we

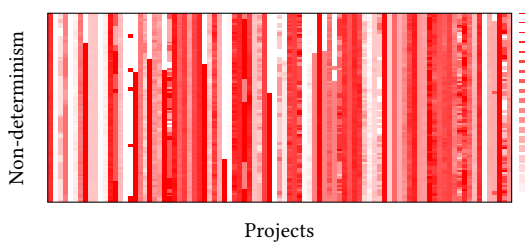


Fig. 9. Heatmap for test non-determinism.

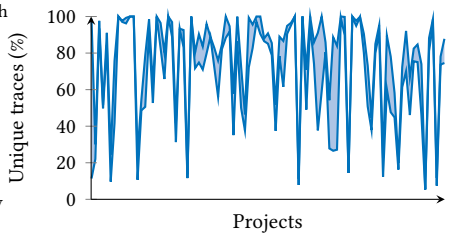


Fig. 10. RL stochasticity's impact per project.

apply modular arithmetic; if a time step reaches a bucket index equal to 100, it is mapped to the first (0-indexed) bucket. Intuitively, a project can be more impacted by test non-determinism if the corresponding column for the project in Figure 9 has more buckets with higher opacity. 28 of 93 projects have very bright plots, showing that the impact of test non-determinism on monitoring them is likely low. But, among the other 65 projects, we find that *the impact of test non-determinism can be high, at 42.5pp on average and a max of 98pp.* Non-determinism affects VALG's monitored unique trace ratios in different runs. For example, in one project, VALG monitors 2.1% of unique traces in one run and 100% of unique traces in another, using the same action selections in VALG's RL agents. Overall, we find the impact of test non-determinism is considerable, and custom RL algorithms for RV could be designed to reduce them in the future.

Evaluating the impact of RL algorithm stochasticity. Even without test non-determinism, stochastic exploration by RL algorithms can cause VALG to monitor different sets of unique traces in different runs of a program. We evaluate the impact of stochasticity like so. Let $\mathcal{L}_p$ be the set of all (s, l) in project p. For each $(s, l) \in \mathcal{L}_p$, let $\mathcal{S}_{s,l} = [\tau_0, \ldots, \tau_{n_{s,l}}]$ be the time series from one TRACEMOP run. We simulate VALG three times on $\mathcal{S}_{s,l}$ to obtain trajectories: $\mathcal{T}_{s,l}^{(i)} = [\sigma_0^{(i)}, \ldots, \sigma_{n_{s,l}}^{(i)}]$, $i \in \{1, 2, 3\}$. A simulation run mimics an RL agent, but rewards are determined based on the time series. For each i-th trajectory, we check how many unique traces VALG monitors ($UT_{s,l}^{(i)}$), and find the min ($UT_{s,l}^{\min}$) and max ($UT_{s,l}^{\max}$) values among the three simulated runs:

$$UT_{s,l}^{(i)} \doteq \sum_{t=0}^{n_{s,l}} \mathbb{I}[\sigma_t^{(i)}.A_t = \texttt{create} \wedge \sigma_t^{(i)}.R_t = 1], \quad UT_{s,l}^{\min} \doteq \min_i UT_{s,l}^{(i)}, \quad UT_{s,l}^{\max} \doteq \max_i UT_{s,l}^{(i)} \tag{13}$$

In other words, $UT_{s,l}^{(i)}$ is the number of time steps where the RL agent creates a monitor and receives a positive reward, i.e., a unique trace is monitored. Then, we check UT_p^{total} the total number of unique traces in p and compute the lower bound $LB(p)$ and upper bound $UB(p)$ of unique traces monitored. Lastly, we measure the impact $SC(p)$ of algorithmic stochasticity on p as the absolute difference between these two bounds:

$$\begin{gathered} UT_p^{\text{total}} \doteq \sum_{(s,l)\in\mathcal{L}_p} \sum_{t=0}^{n_{s,l}} \mathbb{I}[\tau_t = 1], \quad \text{where } \tau_t \in \mathcal{S}_{s,l} = [\tau_0, \ldots, \tau_{n_{s,l}}], \\ LB(p) \doteq \frac{\sum_{(s,l)\in\mathcal{L}_p} UT_{s,l}^{\min}}{UT_p^{\text{total}}}, \quad UB(p) \doteq \frac{\sum_{(s,l)\in\mathcal{L}_p} UT_{s,l}^{\max}}{UT_p^{\text{total}}}, \quad SC(p) \doteq UB(p) - LB(p) \end{gathered} \tag{14}$$

Figure 10 visualizes the results. There, the x-axis corresponds to projects, and the y-axis has two values for the lower and upper bounds. Intuitively, a larger region between the lower and upper bound y-values for a project reflects a greater impact of RL stochasticity. For 49 of 93 projects, the impact of RL stochasticity is low (less than 12.3pp). So, many projects benefit from VALG without suffering from the impact of RL stochasticity. But, among the other 44 projects *algorithmic*

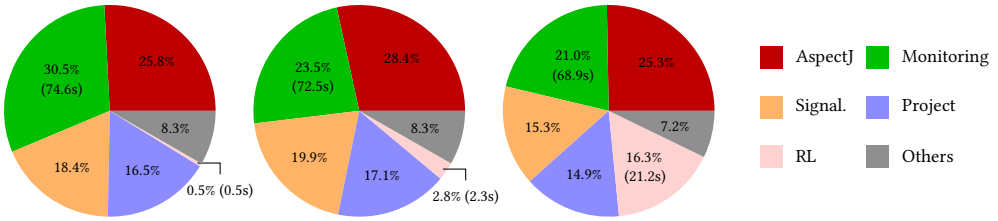


Fig. 11. Profiling results for VALG (left), DUCB (middle), and DS-TS (right).

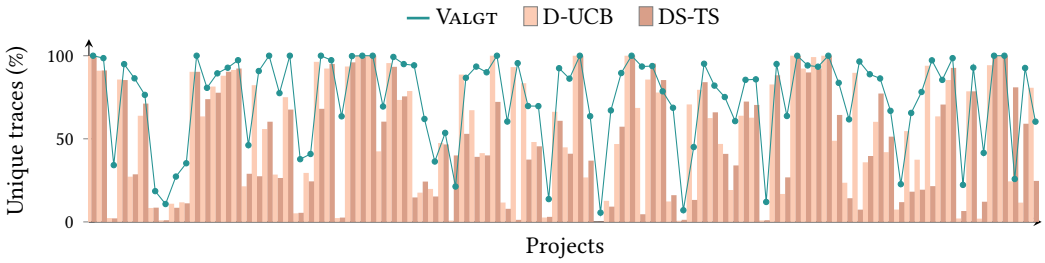


Fig. 12. Ratios of unique traces monitored by VALGT (line), D-UCB, and DS-TS (bars).

stochasticity impacts the ratio of unique traces monitored by 12.3pp on average, with a max of 87.4pp. This swing is not always negative; we find that 42 projects have more unique traces monitored in the second run than in the first one. For example, in `ASM-NonClassloadingExtensions` [26], VALG monitors 11.4% of unique traces in one run, and monitors 98.8% of unique traces in another, even though we simulate its runs on the same time series. We observe that the impact of stochasticity reduces after hyperparameter tuning. We ran `ASM-NonClassloadingExtensions` three times with online-tuned hyperparameters, and the impact of RL stochasticity dropped from 87.4pp to zero.

Running example. For the monitor-creation location on line 269 in Figure 1b, test non-determinism impacts 44.1% of unique traces (2,224 inconsistent time steps out of 5,048 active time steps). Also, stochasticity's impact on monitored unique traces is up to 94.5pp; VALG monitors up to 94.5pp more unique traces in one run than in another, even on the same time series.

4.5 RQ5: How do other RL algorithms (D-UCB and DS-TS) compare with VALG's?

End-to-end time and profiling. VALGJ and VALGT are respectively 1.1x and 1.2x faster than D-UCB on average. In the best case, they are 2.5x and 8.3x faster than D-UCB, respectively. Similarly, VALGJ and VALGT are respectively 1.2x and 1x faster than DS-TS on average. In the best case, they are 3x and 2.2x faster than DS-TS, respectively. In two projects with the largest absolute time differences, VALGT takes 5.9 minutes where D-UCB takes 48.9 minutes, and VALGT takes 95.3 minutes where DS-TS takes 103.6 minutes. Interestingly, our profiling of D-UCB and DS-TS shows that *the choice of RL algorithm influences the RL agent overheads*. Figure 11 shows average profiling results per project, computed as $\left(\sum_{i=1}^{n}((x_i/y_i) * 100)\right)/n$ where x is the time spent in each component, y is the project's end-to-end execution time with RV, and n is the number of projects. We use the same profiling procedure and time categories as in RQ2. The pink portions ("RL") in D-UCB and DS-TS are 5.9x and 34.2x larger than the one for VALG, respectively, showing how the choice of RL algorithms impacts learning overhead. Also, the "Monitoring" portions are slightly smaller in D-UCB and DS-TS than in VALG, but this reduction comes at the cost of monitoring fewer unique traces than VALG, as discussed next.

Unique traces monitored. The line in Figure 12 represents ratios of all unique traces monitored by ValG, and the two bars show ratios for D-UCB and DS-TS per project. ValG monitors equal or more unique traces in 93.5% (86 of 93) and 95.7% (88 of 93) of projects than D-UCB and DS-TS, respectively. On average, ValG monitors 74% of unique traces, while D-UCB and DS-TS monitor 55.3% and 50.4%, respectively. In the worst cases, D-UCB and DS-TS monitor only 11.6% and 1.3% of unique traces in two projects where ValG monitors 92.6% and 95.5%. *ValG misses unique traces, but other RL algorithms miss even more.* Theoretically, test non-determinism or RL stochasticity could have impacted the ratios of monitored unique traces to some degree. But, D-UCB and DS-TS rarely choose to create monitors based on their computed internal values, so ValG is likely to monitor many more unique traces. Overall, using off-the-shelf RL algorithms is insufficient for RV; future research can aim to design RL algorithms that are customized for RV.

Running example. For the `JaroWinklerSimilarity` project in Figure 1b, ValGJ and ValGT are 1.3x faster than DS-TS. Compared with D-UCB, ValGJ is 1.1x faster, but ValGT is 1.2x slower. However, this seemingly faster speed of D-UCB occurs because D-UCB misses many unique traces. For this project, D-UCB and DS-TS only monitor 2.3% and 2.6% of unique traces while ValG monitors 63.5%.

5 Suggestions for Future Research

We discuss six potential directions for improving the use of RL to speed up RV.

(1) Reducing monitoring and event-signaling overheads. RQ1 shows that ValG is often still much slower than running tests without RV. The components of ValG's overhead that future work should target first are its monitoring and event-signaling times. Speeding up ValG's monitoring could be achieved by reducing redundant traces that it monitors. To do so, one direction that may be worth exploring is whether an approximately correct RV approach that selectively creates partially-bound monitors (§4.2) would be faster while monitoring most unique traces that ValG monitors. Also, to speed up event signaling, specialized data structures or RL-aware parametric trace slicing algorithms [19] could be designed.

(2) Window-based value updates. RQ3 showed that ValG misses many unique traces in cases where its estimated action rewards at each time step wrongly assumes that unique traces are consecutive. This assumption allows ValG to be fast, but many missed unique traces show its potential to miss violations. One future direction could be a window-based approach where RL agents use observations from multiple time steps to update reward values. That way, an RL agent may capture a unique trace that occurs after some redundant ones, and continue to choose `create`.

(3) Second-order selective monitoring. Most monitor-creation locations result in few unique traces, and most unique traces come from a small subset of such locations. So, future work could consider *selective* application of selective monitoring to only those locations with many unique traces; other locations can create monitors at initial time steps and then disable monitor creation. This approach could work well for both single program versions and during software evolution. For single versions, work would first be needed to define and identify relevant program *contexts* that RV can use to decide locations in which to perform selective monitoring. During software evolution, RV could use traces from previous versions to make those decisions.

(4) Better heuristic convergence criterion. To further reduce redundant monitoring, future work could exploit the three patterns we identify in §4.3 to design better convergence heuristics. For example, disallowing convergence at initial time steps could prevent premature convergence to `create`. Also, loosening the condition for `ncreate` could enable faster convergence to `ncreate`.

(5) Evaluation metrics. Future work could define custom metrics based on applying statistical approaches across multiple runs to evaluate the use of RL to speed up RV. Doing so could help with designing more effective RL algorithms for RV, e.g., those that can be adjusted on the fly based on

test non-determinism. Such metrics could also help improve the evaluation of RV in the presence of RL stochasticity.

(6) Custom RL algorithms. The results in RQ5, and the associated mini-survey in §3.4 suggest that off-the-shelf RL algorithms alone are not sufficient for further improving the use of RL to speed up RV. So, future work could investigate custom RL algorithms that are specialized for RV. Better convergence heuristics and the proposed window-based value update strategy could be incorporated in the design of such RL algorithms.

6 Threats to Validity

External Validity. Our results may not generalize beyond the projects that we evaluate. However, using the same set of projects from prior work [47, 48] enables comparison with previously reported results. We also add 35 projects that were not used in prior work on using RL to speed up RV. All these projects have diverse program characteristics (see Table 1). Different results may be obtained with different specs. But, the set of specs that we use is the largest available, and prior work [52, 53] showed that the specs are more precise than automatically mined ones. We use Maven-based Java open-source projects and developer-written tests. Results may be different for other programming languages, build systems, or automatically-generated tests. We also study JavaMOP, TraceMOP, and Valg. Other RV techniques exist and can have different outcomes. However, no other RV technique was shown to scale for checking multiple specs at once in hundreds of open-source projects. JavaMOP is one of the most widely-cited tools in the RV community, and TraceMOP and Valg are implemented on top of JavaMOP.

Internal Validity. We write scripts and Maven extensions to automate our experiments and implement two RL algorithms for evaluation. We publicly release all our scripts and code for external validation. We extend TraceMOP to obtain time series (§3.3). Our changes were reviewed by TraceMOP developers and we empirically check that they preserve the semantics of TraceMOP's trace collection. In fact, the time-series collection can be disabled so that TraceMOP's running time and collected traces are not affected.

7 Related Work

Machine learning and RV. RVPrio [61] trained machine learning classifiers using past violations, for use in classifying future violations as true bugs or false alarms. But, RVPrio does not aim to speed up RV. Rather, it aims to make debugging RV violations faster as manual debugging is time consuming [52, 53]. Other learning-based techniques for RV include using conformal prediction [56], BDD representation [63], Bayesian networks [39], deep neural networks [17], or Hidden Markov Models [8, 63]. These approaches target production settings in very different domains (e.g., real-time systems) or logics (e.g., signal temporal logic), than ours. In the opposite direction, RV was applied to learning-based systems, including deep neural networks [32, 34, 82, 86] and autonomous systems [3, 66, 88]. RTSA [50] is a safety assurance system for autopilots that activates a recovery controller, a backup controller that keeps the aircraft within safety constraints in unsafe scenarios. But, the recovery controller can reduce operational efficiency, so RTSA uses reinforcement learning to learn an optimal strategy for switching to the recovery controller. RTSA cannot be applied to testing in CI, where there is no notion of recovery controllers. Valg is designed for use during testing, uses reinforcement learning to speed up RV by reducing redundancy, and is not specifically designed for learning-based systems.

Selective monitoring. Over the past decades, researchers proposed approaches to speed up RV via selective monitoring. Time-triggered RV [14, 15, 87] is a technique for production settings in which RV only runs at fixed time intervals to keep aggregated overhead below a limit. QVM [4] and

SMCO [37] are other RV techniques for production settings. These techniques adjust the rates of monitor creation and event signaling to keep RV's overhead under user-specified limits. In contrast, VALG is designed for testing where RV does not have the long time span of production settings. Also, unlike these prior approaches, VALG does not use a time budget, it speeds up RV by using RL to reduce redundancy in monitored traces. The transformation approach proposed by Purandare et al. [67] reduces redundant event signaling in loops, but it is more limited than VALG. First, unlike VALG, their approach cannot reduce redundancy that is unrelated to loops, e.g., when a methods is called from multiple tests. Also, the static analysis used in their approach has limited support for exception handling and takes very long. VALG does not have these limitations and uses on-the-fly feedback to decide what to monitor. Lastly, their approach has no publicly available tool, so we could not compare it with VALG.

Other approaches for reducing RV overhead. Other approaches have been proposed to speed up RV. Several techniques [10, 11, 29, 64, 75–78] tackle instrumentation time, which dominates RV overhead in many projects [28]. VALG reduces monitoring overhead, so it is complementary to, and can be combined with instrumentation-driven techniques. Evolution-aware RV [29, 54, 55, 89] aims to reduce RV overhead during software evolution. It only monitors parts of code that are affected by code changes. VALG is complementary and orthogonal with evolution-aware RV, as VALG directly optimizes the RV, i.e., when running RV for single program versions. Other approaches that reduce RV overhead include designing specialized data structures [21, 58, 68], defining new algorithms [18, 19, 33, 43, 59, 60, 71], and ahead-of-time static analysis [7, 10, 12, 13, 22]. Many of these data structures and algorithms target JAVAMOP, which VALG is based on. Also, VALG is complementary to the static analysis approaches, but, these approaches do not have available tools and it is not known if they scale for use during CI.

Empirical studies of RV. Several studies helped to understand the effectiveness of RV for bug finding [52, 53] and the main causes of RV's runtime overhead [28]. Our study is partly inspired by [28], but we seek to shed light on whether VALG still has high runtime overhead, and why. Also, our work is specific to the application of RL for speeding up RV, and the insights that we find can be the basis of future work in this direction.

8 Conclusion

Recent work proposed VALG to use reinforcement learning (RL) for speeding up RV. VALG showed promising results, but many questions were not answered about its current limits and how to improve it further. This paper presents the first critical analysis of VALG. Our results unveil many interesting findings and challenges, and we propose several potential directions for future work.

Data Availability

We publicly release all our scripts, data, and code at this GitHub repository: https://github.com/SoftEngResearch/rl-rv-study

Acknowledgments

We thank Kevin Guan, Pengyue Jiang, Stephen Shen, Joonhwan Yoo, David Rodriguez, and the reviewers for valuable comments. This work is supported by Google Research Award 2026, Meta LLM Evaluation Research Grant 2025, and US NSF Grants CCF-2045596, CCF-2319473, CCF-2403035, and CCF-2525243.

References

[1] Shipra Agrawal and Navin Goyal. 2012. Analysis of Thompson Sampling for the Multi-Armed Bandit Problem. In *COLT*.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-Generation Hyperparameter Optimization Framework. In *KDD*.
[3] Dongdong An, Jing Liu, Min Zhang, Xiaohong Chen, Mingsong Chen, and Haiying Sun. 2020. Uncertainty Modeling and Runtime Verification for Autonomous Vehicles Driving Control: A Machine Learning-Based Approach. *JSS* 167 (2020).
[4] Matthew Arnold, Martin Vechev, and Eran Yahav. 2008. QVM: An Efficient Runtime for Detecting Defects in Deployed Systems. In *OOPSLA*.
[5] Async-Profiler. 2026. Async-Profiler: Sampling CPU and Heap Profiler for Java. https://github.com/async-profiler/async-profiler.
[6] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-Time Analysis of the Multiarmed Bandit Problem. In *Machine Learning*.
[7] Pavel Avgustinov, Julian Tibble, and Oege de Moor. 2007. Making Trace Monitors Feasible. In *OOPSLA*.
[8] Ezio Bartocci, Radu Grosu, Atul Karmarkar, Scott A. Smolka, Scott D. Stoller, Erez Zadok, and Justin Seyster. 2013. Adaptive Runtime Verification. In *RV*.
[9] Omar Besbes, Yonatan Gur, and Assaf Zeevi. 2014. Stochastic Multi-Armed-Bandit Problem with Non-Stationary Rewards. In *NIPS*.
[10] Eric Bodden. 2010. Efficient Hybrid Typestate Analysis by Determining Continuation-Equivalent States. In *ICSE*.
[11] Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem. 2007. Collaborative Runtime Verification with Tracematches. In *RV*.
[12] Eric Bodden, Laurie Hendren, and Ondrej Lhoták. 2007. A Staged Static Program Analysis to Improve the Performance of Runtime Monitoring. In *ECOOP*.
[13] Eric Bodden, Patrick Lam, and Laurie Hendren. 2008. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time. In *FSE*.
[14] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. 2011. Sampling-Based Runtime Verification. In *FM*.
[15] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. 2013. Time-Triggered Runtime Verification. In *FMSD*.
[16] Sébastien Bubeck, Nicolo Cesa-Bianchi, et al. 2012. Regret Analysis of Stochastic and Nonstochastic Multi-Armed Bandit Problems. In *Foundations and Trends in Machine Learning*.
[17] Francesca Cairoli, Luca Bortolussi, and Nicola Paoletti. 2021. Neural Predictive Monitoring Under Partial Observability. In *RV*.
[18] Feng Chen, Patrick O'Neil Meredith, Dongyun Jin, and Grigore Roşu. 2009. Efficient Formalism-Independent Monitoring of Parametric Properties. In *ASE*.
[19] Feng Chen and Grigore Roşu. 2009. Parametric Trace Slicing and Monitoring. In *TACAS*.
[20] ConcurrentModificationException 2026. Class ConcurrentModificationException. https://docs.oracle.com/en/java/javase/26/docs/api/java.base/java/util/ConcurrentModificationException.html.
[21] Normann Decker, Jannis Harder, Torben Scheffel, Malte Schmitz, and Daniel Thoma. 2016. Runtime Monitoring with Union-Find Structures. In *TACAS*.
[22] Matthew B. Dwyer, Rahul Purandare, and Suzette Person. 2010. Runtime Verification in Context: Can Optimizing Error Detection Improve Fault Diagnosis?. In *RV*.
[23] Aurélien Garivier and Eric Moulines. 2011. On Upper-Confidence Bound Policies for Switching Bandit Problems. In *ALT*.
[24] W. R. Gilks, S. Richardson, and D. J. Spiegelhalter. 1995. *Markov Chain Monte Carlo in Practice*.
[25] Neil J. Gordon, David J. Salmond, and Adrian F. M. Smith. 1993. A Novel Approach to Nonlinear/Non-Gaussian Bayesian State Estimation. In *IEE Proceedings F*.
[26] Grundlefleck. 2025. ASM-NonClassloadingExtensions. https://github.com/Grundlefleck/ASM-NonClassloadingExtensions.
[27] Kevin Guan, Marcelo d'Amorim, and Owolabi Legunsen. 2025. Faster Explicit-Trace Monitoring-Oriented Programming for Runtime Verification of Software Tests. In *OOPSLA*.
[28] Kevin Guan and Owolabi Legunsen. 2024. An In-Depth Study of Runtime Verification Overheads during Software Testing. In *ISSTA*.
[29] Kevin Guan and Owolabi Legunsen. 2024. Instrumentation-Driven Evolution-Aware Runtime Verification. In *ICSE*.
[30] Kevin Guan and Owolabi Legunsen. 2025. TraceMOP: An Explicit-Trace Runtime Verification Tool for Java. In *FSE Demo*.
[31] Tuomas Haarnoja et al. 2018. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning. In *ICML*.
[32] Vahid Hashemi, Jan Křetínský, Sabine Rieder, Torsten Schön, and Jan Vorhoff. 2024. Gaussian-Based and Outside-the-Box Runtime Monitoring Join Forces. In *RV*.

[33] Klaus Havelund and Grigore Roşu. 2002. Synthesizing Monitors for Safety Properties. In *TACAS*.
[34] Weicheng He, Changshun Wu, and Saddek Bensalem. 2025. Box-Based Monitor Approach for Out-of-Distribution Detection in YOLO: An Exploratory Study. In *RV*.
[35] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in Continuous Integration: Assurance, Security, and Flexibility. In *FSE*.
[36] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *ASE*.
[37] Xiaowan Huang, Justin Seyster, Sean Callanan, Ketan Dixit, Radu Grosu, Scott A. Smolka, Scott Stoller, and Erez Zadok. 2012. Software Monitoring with Controllable Overhead. *IJSTTT* 14, 3 (2012).
[38] hyperminhash 2026. HyperMinHash-java: Union, Intersection, and Set Cardinality in loglog Space. https://github.com/LiveRamp/HyperMinHash-java.
[39] Manfred Jaeger, Kim G. Larsen, and Alessandro Tibo. 2020. From Statistical Model Checking to Run-Time Monitoring Using a Bayesian Network Approach. In *RV*.
[40] M. V. Jambunathan. 1954. Some Properties of Beta and Gamma Distributions. *Annals of Mathematical Statistics* 25, 2 (1954).
[41] JaroWinklerSimilarity 2026. JaroWinklerSimilarity: A Java Implementation of the Jaro–Winkler Similarity. https://github.com/fusion-jena/JaroWinklerSimilarity.
[42] Pengyue Jiang, Kevin Guan, Mahdi Khosravi, Moustafa Ismail, Marcelo d'Amorim, and Owolabi Legunsen. 2026. Fine-Grained Analyses for Evolution-Aware Runtime Verification. In *ICSE*.
[43] Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Roşu. 2011. Garbage Collection for Monitoring Parametric Properties. In *PLDI*.
[44] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. 2012. JavaMOP: Efficient Parametric Runtime Monitoring Framework. In *ICSE Demo*.
[45] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement Learning: A Survey. *JAIR* 4, 1 (1996).
[46] Baekjin Kim and Ambuj Tewari. 2020. Randomized Exploration for Non-Stationary Stochastic Linear Bandits. In *UAI*.
[47] Shinhae Kim, Saikat Dutta, and Owolabi Legunsen. 2025. Faster Runtime Verification during Testing via Feedback-Guided Selective Monitoring. In *ASE*.
[48] Shinhae Kim, Saikat Dutta, and Owolabi Legunsen. 2026. Valg: A Fast Reinforcement Learning Based Runtime Verification Tool for Java. In *ICSE Demo*.
[49] Tor Lattimore and Csaba Szepesvári. 2020. *Bandit Algorithms*.
[50] Christopher Lazarus, James G. Lopez, and Mykel J. Kochenderfer. 2020. Runtime Safety Assurance Using Reinforcement Learning. In *DASC*.
[51] Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, and Grigore Roşu. 2012. *Towards Categorizing and Formalizing the JDK API*. Technical Report. Computer Science Dept., UIUC.
[52] Owolabi Legunsen, Nader Al Awar, Xinyue Xu, Wajih Ul Hassan, Grigore Roşu, and Darko Marinov. 2019. How Effective are Existing Java API Specifications for Finding Bugs During Runtime Verification? *ASE Journal* 26, 4 (2019).
[53] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. How Good are the Specs? A Study of the Bug-Finding Effectiveness of Existing Java API Specifications. In *ASE*.
[54] O. Legunsen, D. Marinov, and G. Roşu. 2015. Evolution-Aware Monitoring-Oriented Programming. In *ICSE NIER*.
[55] O. Legunsen, Y. Zhang, M. Hadzi-Tanovic, G. Roşu, and D. Marinov. 2019. Techniques for Evolution-Aware Runtime Verification. In *ICST*.
[56] Lars Lindemann, Xin Qin, Jyotirmoy V. Deshmukh, and George J. Pappas. 2023. Conformal Prediction for STL Runtime Verification. In *ICCPS*.
[57] Yueyang Liu, Xu Kuang, and Benjamin Van Roy. 2023. A Definition of Non-Stationary Bandits. *arXiv preprint arXiv:2302.12202* (2023).
[58] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu. 2014. RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties. In *RV*.
[59] P.O. Meredith, Dongyun Jin, Feng Chen, and G. Roşu. 2008. Efficient Monitoring of Parametric Context-Free Patterns. In *ASE*.
[60] Patrick Meredith and Grigore Roşu. 2013. Efficient Parametric Runtime Verification with Deterministic String Rewriting. In *ASE*.
[61] Breno Miranda, Igor Lima, Owolabi Legunsen, and Marcelo d'Amorim. 2020. Prioritizing Runtime Verification Violations. In *ICST*.
[62] Volodymyr Mnih et al. 2015. Human-level Control Through Deep Reinforcement Learning. *Nature* 518, 7540 (2015).
[63] Omer Moran and Doron Peled. 2023. Runtime Verification Prediction for Traces with Data. In *RV*.

[64] Samaneh Navabpour, Chun Wah Wallace Wu, Borzoo Bonakdarpour, and Sebastian Fischmeister. 2011. Efficient Techniques for Near-Optimal Instrumentation in Time-Triggered Runtime Verification. In *RV*.
[65] Raydonal Ospina and Silvia L.P. Ferrari. 2012. A General Class of Zero-or-One Inflated Beta Regression Models. *Computational Statistics & Data Analysis* 56, 6 (2012).
[66] Corina S. Păsăreanu, Ravi Mangal, Divya Gopinath, and Huafeng Yu. 2023. Assumption Generation for Learning-Enabled Autonomous Systems. In *RV*.
[67] Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. 2010. Monitor Optimization via Stutter-Equivalent Loop Transformation. In *OOPSLA*.
[68] Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. 2013. Optimizing Monitoring of Finite State Properties Through Monitor Compaction. In *ISSTA*.
[69] Han Qi, Fei Guo, and Li Zhu. 2025. Thompson Sampling for Non-Stationary Bandit Problems. *Entropy* 27, 51 (2025).
[70] RV Specifications 2026. Specs for Correct JDK API Usage Protocols. https://github.com/SoftEngResearch/tracemop/tree/master/scripts/props.
[71] Koushik Sen, Grigore Roşu, and Gul Agha. 2003. Generating Optimal Linear Temporal Logic Monitors by coinduction. In *Advances in Computing Science*.
[72] Samuel S. Shapiro and Hassan Zahedi. 1990. Bernoulli Trials and Discrete Distributions. *Journal of Quality Technology* 22, 3 (1990).
[73] Zhuohang Shen, Mohammed Yaseen, Kevin Guan, Denini Silva, Marcelo d'Amorim, and Owolabi Legunsen. 2026. PyMOP: A Runtime Verification Tool for Python. In *FSE Demo*.
[74] Zhuohang Shen, Mohammed Yaseen, Denini Silva, Kevin Guan, Junho Lee, Marcelo d'Amorim, and Owolabi Legunsen. 2025. A Generic and Efficient Python Runtime Verification System and its Large-scale Evaluation. *arXiv preprint arXiv:2509.06324* (2025).
[75] Chukri Soueidi and Yliès Falcone. 2023. Bridging the Gap: A Focused DSL for RV-Oriented Instrumentation with BISM. In *RV*.
[76] Chukri Soueidi and Yliès Falcone. 2023. Instrumentation for RV: From Basic Monitoring to Advanced Use Cases. In *RV*.
[77] Chukri Soueidi, Yliès Falcone, and Sylvain Hallé. 2023. Dynamic Program Analysis with Flexible Instrumentation and Complex Event Processing. In *ISSRE*.
[78] Chukri Soueidi, Marius Monnier, and Yliès Falcone. 2023. Efficient and Expressive Bytecode-Level Instrumentation for Java Programs. *IJSTTT* 25, 4 (2023).
[79] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*.
[80] Csaba Szepesvári. 2010. *Algorithms for Reinforcement Learning*.
[81] TraceMOP 2024. TraceMOP: A Trace-Aware Runtime Verification Tool for Java. https://github.com/SoftEngResearch/tracemop.
[82] Muhammad Usman, Divya Gopinath, Youcheng Sun, and Corina S. Păsăreanu. 2022. Rule-Based Runtime Mitigation Against Poison Attacks on Neural Networks. In *RV*.
[83] Lingwei Wang, Lei Zhang, Jiaxin Zhang, et al. 2020. Truly Proximal Policy Optimization. In *UAI*.
[84] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-Learning. *Machine Learning* 8, 3 (1992).
[85] Christian H. Weiß. 2021. Time Series Modelling. *Entropy* 23, 9 (2021).
[86] Changshun Wu, Yliès Falcone, and Saddek Bensalem. 2023. Customizable Reference Runtime Monitoring of Neural Networks Using Resolution Boxes. In *RV*.
[87] Chun Wah Wallace Wu, Deepak Kumar, Borzoo Bonakdarpour, and Sebastian Fischmeister. 2013. Reducing Monitoring Overhead by Integrating Event- and Time-Triggered Techniques. In *RV*.
[88] Beyazit Yalcinkaya, Hazem Torfah, Daniel J. Fremont, and Sanjit A. Seshia. 2023. Compositional Simulation-Based Analysis of AI-Based Autonomous Systems for Markovian Specifications. In *RV*.
[89] Ayaka Yorihiro, Pengyue Jiang, Valeria Marques, Benjamin Carleton, and Owolabi Legunsen. 2023. eMOP: A Maven Plugin for Evolution-Aware Runtime Verification. In *RV*.

Received 29 January 2026; accepted 16 April 2026