Faster Runtime Verification during Testing via Feedback-Guided Selective Monitoring

Shinhae Kim
Department of Computer Science
Cornell University
Ithaca, New York, USA
sk3364@cornell.edu

Saikat Dutta
Department of Computer Science
Cornell University
Ithaca, New York, USA
saikatd@cornell.edu

Owolabi Legunsen
Department of Computer Science
Cornell University
Ithaca, New York, USA
legunsen@cornell.edu

***Abstract*—Runtime verification (RV) uses monitors, which are dynamically synthesized from formal specifications (specs), to check running programs against specs. RV of passing tests in many open-source projects found hundreds of new bugs. But, high overheads make it hard to use RV for testing in practice.**

We propose Valg, the first on-the-fly selective RV technique for testing, and the first to use reinforcement learning (RL) to speed up RV. Valg leverages a recent finding: 99.87% of monitors are *redundant* for testing; they wastefully re-check unique traces—sequences of events, *e.g.*, method calls—that the other *necessary* 0.13% already checked. Valg uses feedback about redundancy of prior monitors and events to selectively monitor only necessary ones subsequently. A key idea in Valg is our novel formulation of selective monitor creation as a two-armed bandit RL problem that rewards necessary monitors and penalizes redundant ones.

We implement Valg for Java and compare it with state-of-the-art RV tools on one revision each of 64 open-source projects. With default RL hyperparameters, Valg is up to 20.2x and 551.5x faster than JavaMOP and TraceMOP, respectively. For example, Valg takes only 11.6 minutes in total to monitor three projects where TraceMOP takes 3.02 days in total. With default RL hyperparameters, Valg finds 99.6% of spec violations found by JavaMOP and TraceMOP, but it only checks 76.7% of their unique traces on average. After tuning RL hyperparameters, Valg checks 95.1% of unique traces on average with minor loss in speed. Using tuned hyperparameters from one revision "into the future" as code evolves preserves Valg's high speedups and rate of checked unique traces, without needing frequent re-tuning.

***Index Terms*—Runtime Verification, Software Testing, Reinforcement Learning**

I. Introduction

Runtime verification (RV) [27], [43], [44], [57] checks if running programs satisfy formal specifications (specs). RV first instruments a program to signal spec-related *events*, *e.g.*, method calls or field accesses. Then, at runtime, RV synthesizes *monitors*—usually automata like finite-state machines—from specs. Monitors check if event sequences, *i.e.*, *traces*, satisfy the specs. If a trace violates a spec, the corresponding monitor raises a violation or performs error recovery.

RV research traditionally focused on its usage in production settings, an approach that is now being adopted, *e.g.*, in ARTCAT [5] and the Linux kernel [20]. But, recent work showed that RV also amplifies the bug-detection capability of *passing* tests during development. RV found hundreds of bugs that testing alone missed in many open-source projects, using specs of correct JDK API usage protocols [64], [66], [78]. The reason is that these behavioral specs, which are hard to express as test assertions, provide additional test oracles.

Wider RV adoption in continuous integration (CI) [47], [48] is hindered by time overheads that monitors incur, especially when checking many specs simultaneously. A recent study shows that these overheads are still as high as 5,000x, or 27 hours, despite decades of research on reducing them [36].

We seek to speed up RV during testing by reducing two kinds of waste caused by loops or multiple calls to a method:

1. Most traces for parametric specs are redundant. Parametric specs concern instances of related object types [18], [19]. So, RV creates a monitor for each of the often hundreds of millions of related instances [63]. But, the specs we use are about method call sequences. So, if two identical traces occur on a program path, then one of them is redundant for bug finding. Such traces are not redundant in production settings [5], [20], where RV must check every trace and react to every violation. Based on this reasoning, Guan and Legunsen [36] find that 99.87% of monitors are *redundant* for testing because they wastefully re-check unique traces that the other *necessary* 0.13% of monitors already checked.

2. Most events checked by monitors for non-parametric specs are redundant. Non-parametric specs [61], [73] are object agnostic; their event definitions often use very expensive computation to check if a (static) method call, its argument(s), or calling context violates an API. RV creates only one monitor per non-parametric spec throughout a monitored program's execution. But, most events checked by such monitors during testing are redundant: we find from data in [36] that over 99.99% of such events that violated an API are redundant.

These high degrees of redundant monitoring suggest that faster RV via selective monitor creation (for parametric specs) and selective event signaling (for non-parametric specs) is needed and feasible. But, prior selective monitoring work like QVM [4], SMCO [49], and time-triggered RV [16] only target production settings. Purandare et al. [87] use loop transformations to reduce redundant monitoring. But, their static analysis has limited support for exceptions, has no tool that one can use today, and was not evaluated during testing of evolving software. §VI discusses related work in more detail.

We propose Valg (Danish for "choice"), the first on-the-fly selective RV technique for testing, and the first to use

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00189

```
1 Appendable_ThreadSafe(Appendable a) {
2  Thread owner = null;
3  event safe_append before(Appendable a, Thread t) :
4  call(* Appendable+.append(..)) && target(a) &&
5   thread(t) && !target(StringBuffer) &&
6   condition(this.owner == null || this.owner == t) {
7   this.owner = t; }
8  event unsafe_append before(Appendable a, Thread t) :
9  call(* Appendable+.append(..)) && target(a) &&
10  thread(t) && !target(StringBuffer) &&
11  condition(this.owner != null && this.owner != t) {}
12 ere: safe_append*
13 @fail { /* print violation */ }
```

```
1 Math_ContendedRandom() {
2  Thread th = null;
3  event onethread_use before(Thread t) :
4  call(* Math.random(..)) && thread(t) &&
5   condition(this.th == null || this.th == t) {
6   this.th = t;
7  }
8  event otherthread_use before(Thread t) :
9  call(* Math.random(..)) && thread(t) &&
10  condition(this.th != null && this.th != t) {
11  }
12  ere : onethread_use*
13  @fail { /* print violation */ }
```

Fig. 1: Examples of parametric (left) and non-parametric (right) specs.

reinforcement learning (RL) to speed up RV. Valg addresses the challenge of on-the-fly prediction of whether about-to-be created monitors for parametric specs, or signaled events for non-parametric specs will be redundant. So, Valg uses feedback about redundancy of prior monitors and events to guide future monitor creation and event signaling. That feedback can be seen as probability p, based on prior traces, that future events at location ℓ in monitored program P will create redundant monitors (or be redundant). If p is below a threshold, P signals the next event to RV and updates $\ell's$ probability. Otherwise, that next event is not signaled.

A key idea in Valg is our novel formulation of selective monitor creation as a two-armed bandit RL problem. Valg's design is based on an insight from our analysis of data from [36]: *if a redundant monitor is created or a redundant event is signaled for spec s at location ℓ, subsequent monitors or events for s at ℓ are likely to also be redundant*. Valg assigns an RL agent to each ℓ. The RL reward function aims to reduce redundant monitors and preserve unique traces. So, each agent rewards necessary monitor-creation actions, and penalizes redundant ones. To reduce redundant events for non-parametric specs, Valg only signals an event at location ℓ if the spec was not previously violated at ℓ.

We implement two variants of Valg for Java: (i) Valg$_\text{J}$, built on JavaMOP [54], [73]—a state-of-the-art (SoTA) *implicit-trace* RV tool that uses event-by-event monitoring algorithms to avoid the time and space costs of storing traces; and (ii) Valg$_\text{T}$, built on TraceMOP [38]—a SoTA *explicit-trace* RV tool that works like JavaMOP but also stores traces.

We evaluate Valg using 160 JDK API specs and default RL hyperparameters on one revision each of 64 Java open-source projects. Valg$_\text{J}$'s and Valg$_\text{T}$'s end-to-end times are up to 20.2x (geometric mean: 1.4x) and 551.5x (geometric mean: 1.8x) faster than JavaMOP's and TraceMOP's, respectively, while preserving 99.6% of spec violations. Excluding instrumentation time, Valg$_\text{J}$ and Valg$_\text{T}$ are up to 625x and 909.1x faster than JavaMOP and TraceMOP, respectively. For example, Valg reduces 24.1, 24.1, and 24.3 hours that TraceMOP takes for three projects to 4.2, 2.6, and 4.8 minutes, respectively. Valg reduces redundant traces and events by 96.4% and 98.7%, respectively. Valg also exceeds the relative speedups and the number of detected violations of two baselines that randomly sample events and traces by up to 20.1x and 7.0x, respectively.

Users who care only about violation preservation can use Valg with default hyperparameters. But, with default hyperparameters, Valg checks only 76.7% of unique traces on average. Checked unique traces is a stronger criterion for Valg's bug-detection ability than violation preservation: checking all unique traces guarantees finding all violations, but not vice versa. So, as an optimization, we use Optuna [1] to tune hyperparameters, resulting in Valg checking 97.1% of unique traces with little loss in speed. Hyperparameter tuning is costly, but it is an offline process whose results can be reused as code evolves. After tuning, Valg preserves its high rate of unique traces checked, violations preserved, and speedups across many revisions of a subset of 46 evaluation projects.

This paper makes the following contributions:

- ⋆ **Technique.** Valg is the first on-the-fly selective monitor creation and event signaling technique for RV of tests and the first to use reinforcement learning to speed up RV.
- ⋆ **Tools.** We implement Valg$_\text{J}$ and Valg$_\text{T}$ on top of SoTA Java RV tools that already integrate with open-source projects.
- ⋆ **Comparisons.** We compare Valg in single- vs. multi-revision settings, with and without hyperparameter tuning, and with random sampling.
- ⋆ **Results.** Valg speeds up RV by up to 551.5x, finds 99.6% of violations, checks 97.1% of unique traces, reduces redundancy by up to 98.7%, and outperforms random sampling.

Our Valg implementation, evaluation scripts, and artifacts are at https://github.com/SoftEngResearch/Valg.

II. Motivating Examples and Background

First, §II-A and §II-B provide examples of the kinds of specs that we use in this paper and the two kinds of redundant monitoring that Valg aims to reduce. Then, §II-C provides a brief background on reinforcement learning.

A. Specs

The left side of Figure 1 shows an example parametric spec, `Appendable_ThreadSafe` [3], which is parameterized over `Appendables` (other than `StringBuffer`, lines 5 and 10); it checks if calls to `append` are thread safe. A `safe_append` event (lines 3–7) is signaled *before* the first call to `append` on `Appendable a`, at which time RV creates a monitor whose `owner` field references the current thread. That event is also signaled on subsequent calls to `a.append()` from `owner`. But, before `a.append()` is called in threads other than `owner`, the `unsafe_append` event (lines 8–11) is signaled, violating the extended regular expression (ERE) property on line 12 and causing RV to print a violation message (line 13).

```
1 private double eval(String f_x, double xi)
2  // exception and local variable declarations
3  for (int i = 0; i < f_x.length(); i++) {
4   char character = f_x.charAt(i);
5   if (character >= '0' && character <= '9') {
6    hasNumber = true;
7    number += character; // calls .append() twice
8    if (i == (f_x.length() - 1)) {
9     value = new Double(number).doubleValue();
10     ...}}...}
11  return value; }
```

```
1 public static String generateData(int byteSize) {
2  ...
3  StringBuilder b = new StringBuilder(byteSize * 2);
4  for (int i = 0; i < byteSize; i++) {
5   if (Math.random() * 100 > 98) {
6    // appends a terminating character to b
7   } else {
8    // appends a random character to b
9   }
10  }
11  return b.toString(); } // called by many threads
```

Fig. 2: Code generating redundant monitors for a parametric spec (left) and redundant events for a non-parametric spec (right).

We show `Appendable_ThreadSafe` here for simplicity. Most parametric specs in this paper are more complex, parameterized by multiple object types, and use other logics such as Linear Temporal Logic (LTL) [85] and Finite State Machines (FSM) [79]. Also, parametric specs often designate *creation event(s)* that trigger monitor creation. If no creation event is designated, the first spec-related event that is signaled acts as the creation event.

The right side of Figure 1 shows a non-parametric spec, `Math_ContendedRandom` [75]; it checks if the static method `Math.random()` is called from multiple threads, which increases thread contention [82]. RV creates only one monitor to check all occurrences of both `Math_ContendedRandom` events in a monitored program. The `onethread_use` event (lines 3–7) is signaled *before* the first `Math.random()` call, stores the current thread object in the monitor's `th` field, and is signaled for all subsequent `Math.random()` calls from `th`. The `otherthread_use` event (lines 8–11) is signaled before `Math.random()` calls from threads other than `th`, and a violation message of the ERE on line 12—`Math.random()` should be called within the same thread—is printed on line 13. `Appendable_ThreadSafe` and `Math_ContendedRandom` helped find confirmed bugs [66].

B. Valg reduces redundant monitoring

Figure 2 shows simplified code snippets where numerous monitors (`eval`, left) and events (`generateData`, right) are redundant. These snippets are from `expression.parser` [92] and `asterisk.java` [6], respectively. To evaluate expression `f_x` w.r.t. variable `xi`, `eval` iterates over and appends each character in `f_x` to `number` (line 7). RV creates a new `Appendable_ThreadSafe` monitor on each call to `eval` to check two `safe_append` events on line 7 (the append operator is internally translated into two `append` method calls). Because `eval` is called often during testing, RV creates 68,000,157 monitors, all of which check the same unique trace [`safe_append`, `safe_append`] of events from the same location. Yet, all such monitors except the first are redundant and cannot find any new violation. Worse, similar snippets in the same class trigger the creation of 164,000,349 redundant `Appendable_ThreadSafe` monitors; they add to RV's overhead but not to its bug-finding ability. Valg speeds up RV by reducing redundant monitor creation.

The `generateData` method on the right of Figure 2 creates character sequences, calling `Math.random()` per iteration (line 5) to add terminating characters. But, `generateData` is called from multiple threads, leading to 260,000,000 redundant events that violate `Math_ContendedRandom` at line 5. Valg also speeds up RV by reducing such redundant violating non-parametric events.

C. A brief background on reinforcement learning (RL)

An RL *agent* interacts with an *environment* to learn an optimal *policy* (a mapping from environment states to actions) for achieving its goal [28]. To do so, an agent takes *actions* in its environment states. Then, the agent receives a *reward*, a numeric value on how good its action selection was with respect to its goal. Next, the agent updates its policy based on the reward. This process is repeated per *time step* and the agent balances *exploration* (trying new actions) and *exploitation* (selecting the best known action) to maximize rewards.

K-armed bandit [102] is a simpler RL problem where environments have no states. At each time step t, the agent selects action $\mathcal{A}_t$ from k possible actions (or *arms*), each of which has an unknown *reward probability distribution*. The agent gets reward $\mathcal{R}_t$ based on $\mathcal{A}_t$'s reward distribution and aims to maximize its rewards. *Action-value methods* are commonly used to solve k-armed bandit problems; they have two phases. The first phase estimates *value* $Q_t(a)$—expected reward at t—of each action a, based on past rewards. The *sample-average* action-value method's estimate is the average sum of past rewards, where:

$$Q_t(a) \doteq \begin{cases} \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}, & \text{if } \sum_{i=1}^{t-1} \mathbb{1}_{A_i=a} > 0 \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where $\mathbb{1}_{A_i=a}$ is 1 if $A_i = a$ and 0 otherwise. Estimated values can be obtained efficiently incrementally [28]: the $n+1$th expected value after taking an action n times, Q_{n+1}, computed from the nth expected value Q_n and its observed reward R_n:

$$Q_{n+1} \doteq Q_n + \frac{1}{n}(R_n - Q_n) \quad (2)$$

Sample-average is simple, but it requires *stationary* environments where reward distributions do not change over time. For *non-stationary* environments, the *exponential recency-weighted average (ERWA)* is more effective [102]. ERWA prioritizes recent rewards using a weighted sum of past rewards that adapts better to non-stationary environment changes:

$$Q_{n+1} \doteq Q_n + \alpha(R_n - Q_n), \text{ where } \alpha \text{ is a learning rate} \quad (3)$$

In the second phase, action-value methods choose an action based on estimated rewards. In the simplest *greedy action selection* strategy, an agent always selects the highest-valued

action. But, in non-stationary environments, strategies enabling probabilistic exploration can be effective, *e.g.*, *epsilon-greedy action selection* strategy [102] chooses a random action a with probability ϵ, or the action with the highest expected reward:

$$\mathcal{A}_t \leftarrow \begin{cases} \arg\max_a Q_t(a) & \text{with probability } 1-\epsilon \\ \text{random action } a & \text{with probability } \epsilon \end{cases} \quad (4)$$

III. Technique

A. Selective Monitoring

Valg uses RL to selectively create parametric monitors, and violation feedback to selectively signal non-parametric events.

Rationale for a k-armed bandit formulation. We formulate selective monitor creation as a k-armed bandit problem for two reasons: (i) Monitor creation is a stateless problem; each decision is not affected by outcomes of previous decisions. (ii) General RL algorithms can incur unacceptable overhead during RV. For example, Q-Learning [51] or Monte Carlo Policy Gradient [106] must update a tabular structure or neural network per reward observation. Such updates could incur overhead that is proportional to the number of states. Moreover, defining states is not trivial in the RV context. A naïve approach would be to define states based on previous traces, but the space of such states may be intractably large.

RL-guided selective parametric monitor creation. Valg's monitor-creation agents learn policies that minimize the number of redundant traces, maximize the number of unique ones, and preserve unique violations. Each agent's environment in the monitored program is a *monitor creation point* ℓ, a unique program location where monitors can be created. *Time step* t^l is when a monitor creation decision is made at ℓ. An RL agent associated with each ℓ takes one of two *actions* (or *arms*) at each t^l, *i.e.*, $\mathcal{A}_t \in \{\mathsf{create},\ \mathsf{ncreate}\}$, where create synthesizes a monitor and ncreate does not. There are only two possible actions, so selective monitor creation is a two-armed bandit problem involving Valg and the monitored program.

The reward for an action is drawn from two time-dependent reward distributions $\mathcal{R}_{\mathsf{create}}(t^l)$ and $\mathcal{R}_{\mathsf{ncreate}}(t^l)$, where:

- $\mathcal{R}_{\mathsf{create}}(t^l)$ is a Bernoulli distribution for create at l, with success probability $p \in \{0, 1\}$; the reward is computed as:

$$R_{\mathsf{create},\ t^l} \doteq 0 \text{ if } (\mathsf{trace}^l_{t^l} \text{ is redundant}) \text{ else } 1 \quad (5)$$

- $\mathcal{R}_{\mathsf{ncreate}}(t^l)$ is a degenerate distribution [89] over [0,1] for ncreate at ℓ; the reward is the ratio of duplicate traces over total traces checked by all monitors created at ℓ prior to t^l:

$$R_{\mathsf{ncreate},\ t^l} \doteq \frac{\sum_{k=0}^{t^l-1} \mathbb{1}(\mathsf{trace}^l_k \text{ is redundant})}{\sum_{k=0}^{t^l-1} \mathbb{1}(\mathsf{trace}^l_k \text{ is observed})} \quad (6)$$

If $\mathcal{A}_t$ is create, Valg assigns the agent a reward of 0 if the monitored trace was previously checked, or 1 otherwise. These reward values aim to preserve unique traces. If $\mathcal{A}_t$ is ncreate, Valg cannot track the trace that would have been monitored. So, Valg assigns a continuous-valued reward based on traces checked by prior monitors created at ℓ. That reward is an estimate of the likelihood that the trace that would have been checked is redundant. The number of observed (redundant) traces increases with time, so the reward value is updated every time a monitor is created.

Algorithm 1 Valg's monitor creation action selection

Inputs: l: monitor creation point in program P, t^l: time step at l
Output: $\mathcal{A}_{t^l}$: selected action at time step t^l
Globals: α: learning rate, ϵ: exploration probability, δ: threshold
Q_n: map of actions to values after n selections
$\mathcal{A}_{t^l-1}$: previous action selected at time step $(t^l - 1)$
trace^l: set of observed traces at monitor creation point l
$n_\mathsf{c}, n_\mathsf{n}, \mathcal{A}_\mathsf{conv}$: action counters and action of convergence

```
1:  procedure decideAction(l, t^l)
2:    n_c ← 0, n_n ← 0, A_conv ← null
3:    if t^l = 0 then return arg max_A (Q_0(A))                ▷ initial values
4:    if A_conv ≠ null then return A_conv                      ▷ convergence
5:    if A_{t^l-1} is create then                              ▷ previous action: create
6:      R_{create,t^l-1} ← isDuplicate(trace^l_{t^l-1}) ? 0 : 1
7:      Q_{n_c+1}(create)
            ← Q_{n_c}(create) + α(R_{create,t^l-1} − Q_{n_c}(create))
8:      n_c ← n_c + 1
9:    else if A_{t^l-1} is ncreate then                        ▷ previous action: ncreate
10:     R_{ncreate,t^l-1} ← duplicatesRatio(trace^l_{0..t^l-1})
11:     Q_{n_n+1}(ncreate)
            ← Q_{n_n}(ncreate) + α(R_{ncreate,t^l-1} − Q_{n_n}(ncreate))
12:     n_n ← n_n + 1
13:   if |Q_{n_c}(create) − Q_{n_n}(ncreate)| < δ then
14:     A_conv ← (Q_{n_c}(create) > Q_{n_n}(ncreate)) ? create : ncreate
15:   if rand(0, 1) < ε then return choose([create, ncreate])
16:   else return (Q_{n_c}(create) > Q_{n_n}(ncreate)) ? create : ncreate
```

Selective non-parametric event signaling. Valg uses location and violation information to selectively signal non-parametric events. Valg records the location ℓ of a non-parametric event that violates an API. Then, that event is subsequently signaled only if its past occurrences at ℓ were non-violating.

B. Action-value Method for Selective Monitor Creation

Value estimation and action selection. To estimate action values, Valg uses ERWA [102], which weighs recent rewards more than past rewards (Equation 3). The rationale is that selective monitor creation is non-stationary: reward distributions change with time—the probability to observe a unique trace is 100% or 0% and continuously alternates over time steps (§III-A). Also, Valg enables probabilistic exploration by adopting the epsilon-greedy action selection strategy (Equation 4). Other strategies, *e.g.*, epsilon-decaying greedy selection [74] are less suitable for selective monitoring because unique traces can be observed at any time regardless of history.

Initial value selection. Valg selects initial values of actions, which control the agent's exploration using a balanced strategy: an initial *optimistic* value for create and a *realistic* value for ncreate. A realistic initial value enables exploration only based on the action-selection strategy (*e.g.*, with probability ϵ in epsilon-greedy action selection); it initially assigns zero to the action and enables the agent to learn real values during exploration. In contrast, an optimistic initial value encourages active exploration; it assigns an initial optimistic value to the action, and when an action value gets smaller than that of another action, the agent switches its exploration. So, Valg's

Algorithm 2 Selective parametric monitor creation

Inputs: l: monitor creation point in P, s: a parametric spec
$e(\theta)$: an event for s at l, binding parameter instances θ
Globals: notTracked: $(s, l) \rightarrow$ untracked parameters for s at l
t^l: current time step at l

1: **procedure** signalEvent($e(\theta), s, l$) ▷ called in P
2: **if** $\theta \notin$ notTracked$[(s, l)]$ **then**
3: status $\leftarrow$ check($e(\theta), s, l, t^l$) ▷ call to Valg
4: **if** status is notTracked **then**
5: notTracked$[(s, l)] \leftarrow$ notTracked$[(s, l)] \cup \{\theta\}$
6: **procedure** check($e(\theta), s, l$) ▷ called in Valg
7: **if** e is a creation event **then**
8: action $\leftarrow$ decideAction(l, t^l); $t^l \leftarrow t^l + 1$ ▷ run Algorithm 1
9: **if** action is create **then**
10: m $\leftarrow$ createMonitor(s); rv(m, $e(\theta)$); **return** tracked
11: **else if** action is ncreate **then return** notTracked
12: **else**
13: m $\leftarrow$ findMonitor(s, θ); rv(m, $e(\theta)$); **return** tracked

balanced strategy encourages the selection of create at earlier time steps. This choice is grounded in our empirical analysis: unique traces often occur more frequently at earlier time steps.

Convergence logic. To determine when an agent should stop learning, Valg uses a heuristic convergence logic: it checks if the absolute difference in action values is sufficiently close to 1. Theoretically, convergence is infeasible in non-stationary environments as it requires the learning rate to decrease over time [104], but the rate is fixed in such environments. However, at monitor creation points, action values often approach to 0 for one action and 1 for another. For example, if a loop repetitively generates duplicate (unique) traces, create and ncreate values will approach 0 (1) and 1 (0), respectively.

Action-selection procedure. Algorithm 1 shows how a Valg agent for monitor creation point ℓ selects an action $\mathcal{A}_{t^l}$ at time step t^l. For the first time step, the agent takes the action with a higher initial value (line 3). If convergence has occurred, the agent takes action $\mathcal{A}_{\mathsf{conv}}$ (line 4). Otherwise, the agent uses its last action $\mathcal{A}_{t^l-1}$ to update its value estimation. If that action was create, Valg assigns a binary reward based on the Bernoulli distribution and updates the creation action value $Q_n(\mathsf{create})$ (lines 5–8). If the last action was ncreate, Valg assigns a continuous reward based on the ratio of duplicate traces seen so far at ℓ and updates the no-creation action value $Q_n(\mathsf{ncreate})$ (lines 9–12). Lastly, the agent updates its convergence status based on δ (lines 13–14) and either explores a random action with the probability ϵ (line 15) or takes an action based on the learned values (line 16).

C. Implementation

Selective parametric monitor creation. Algorithm 2 shows how Valg selectively creates monitors. If an event's parameter instance θ—a partial function from parameter types to concrete objects [19]—is excluded from the "not-being-tracked" set (explained shortly), the instrumented program P signals that event to Valg (lines 2–3). If the signaled event is a creation event (§II), Valg invokes decideAction (Algorithm 1) on the RL agent for that code location to select an action (lines 7–8). If the selected action is create, Valg creates a new monitor

Algorithm 3 Selective non-parametric event signaling

Inputs: l: event location in program P
s: a non-parametric spec, e: an event for s at l
Globals: vLocs: $s \rightarrow$ set of l where s has been violated

1: **procedure** signalEvent(e, l, s) ▷ called in P
2: **if** $l \notin$ vLocs$[s]$ **then** status $\leftarrow$ check(e) ▷ call to Valg
3: **if** status is violated **then** vLocs$[s] \leftarrow$ vLocs$[s] \cup \{l\}$
4: **procedure** check(e) ▷ called in Valg
5: r $\leftarrow$ runMonitor(e); **return** r ▷ r $\in$ {violated, notViolated}

to check the event, and returns tracked to P, indicating that θ is being tracked (lines 9–10). If the action is ncreate, Valg does not create a monitor, and returns notTracked, so P adds θ to the set not being tracked (lines 11, then 4–5). If the signaled event is not a creation event, Valg uses that event's instance parameter to find existing monitors that should check it, sends the event to those monitors, and returns tracked to P (lines 12–13). Valg's parameter-based monitor search uses JavaMOP or TraceMOP's parametric trace slicing algorithms to handle parameter instances [18], [19].

In addition to Algorithm 2, we discuss how Valg addresses two important problems in selective monitor creation.

1. *Monitors for partially bound parameter sets*. Events seen so far may not have bound all parameters of the spec. In such cases, parametric trace slicing requires the ability to create monitors for future events with more completely bound parameter sets by cloning prior monitors for partially bounded parameter sets. For this reason, Valg does not apply selective monitor creation to monitors for partially-bound parameter sets.
2. *Overheads*. Selective monitor creation incurs two main costs: (i) the parameter sets being checked on line 2 can grow very large, causing frequent and costly resizing operations in Valg's internal data structures; and (ii) checking if the (often very long) monitored traces are duplicates per create action is time consuming. Valg uses two approaches to reduce these costs. First, an integer max window size, w, can be used to track the most recent w object instances per pair of spec and location. The intuition is that monitors created long ago are unlikely to be updated. Second, Valg performs efficient checking using integer encoding. To uniquely identify an event, Valg computes a hash of the joinpoint object, i.e., a unique object that distinguishes an instrumented program location. Also, to encode the order between events, Valg adds a number from a pseudo-random sequence with the same seed for each event. Valg keeps a set of the encoded integers for unique traces, so the trace comparison becomes a simple integer set inclusion check.

Selective non-parametric event signaling. Algorithm 3 shows that instrumented program P selectively signals a non-parametric event to Valg only after checking whether the corresponding spec s was not previously violated at location ℓ (line 2). If so, Valg signals that event to the singleton monitor for that spec, which in turn informs P whether that event violates the spec (line 5). If so, P adds ℓ to set of locations where s was previously violated (line 3), and will not signal

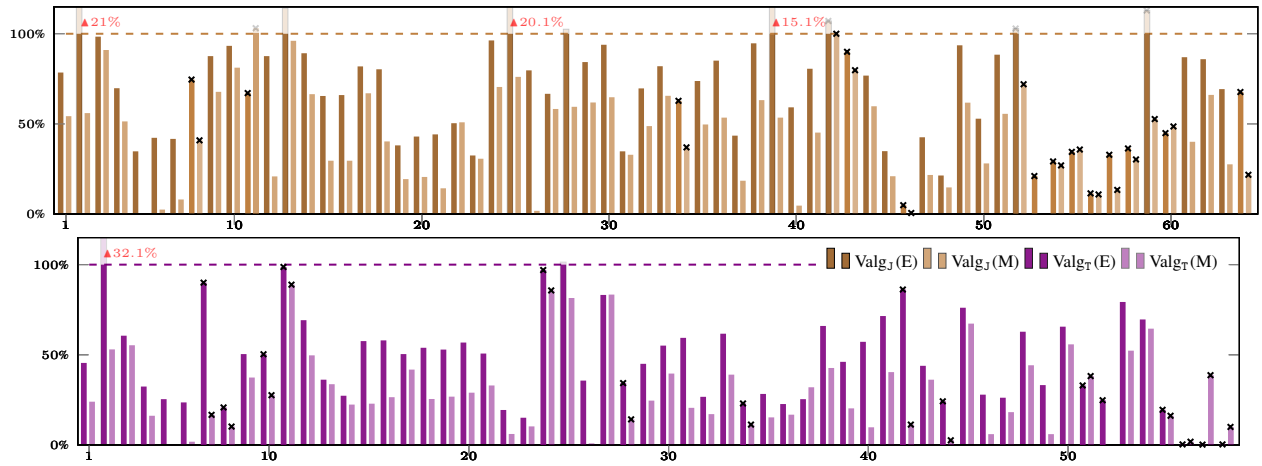

Fig. 3: Reduction ratio of end-to-end (E) and monitoring-only (M) times in different projects

future s events from ℓ.

We implement Algorithms 2 and 3 for Java by extending two state-of-the-art RV tools, JavaMOP and TraceMOP, into Valg_J and Valg_T, respectively. These RV tools instrument the program using AspectJ [56], and signal events to handler methods. Valg modifies AspectJ code to selectively signal events, and integrates RL agents into the handler methods. Our tools easily integrate with Maven, a popular build system.

TABLE I: Summary statistics on projects in our evaluation: no. of test methods (#tests), time in seconds w/o RV (time), lines of code (LoC), statement coverage. % (cov^s), branch coverage. % (cov^b), no. of commits (#SHAs), years since 1st commit (age), and no. of stars (#★).

	#tests	time	LoC	cov^s	cov^b	#SHAs	age	#★
Mean	146	44.5	9,643	61.2	53.4	633	9.9	493
Med	41	8.8	3,565	66.6	52.9	142	10.1	57
Min	1	0.2	49	3.1	4.7	3	3.1	6
Max	1,456	1,565.5	92,687	95.4	92.2	17,223	22.0	9,893
Sum	9,546	2,889.8	626,802	n/a	n/a	41,173	n/a	32,085

IV. Evaluation

We organize our evaluation around four research questions:

RQ1. How does Valg compare with JavaMOP and TraceMOP, in terms of overhead, violations, and unique traces?

RQ2. How does Valg compare with random sampling?

RQ3. What is the impact of hyperparameter tuning on Valg's efficiency and effectiveness?

RQ4. How effective and efficient is Valg as code evolves?

RQ1 evaluates whether Valg is faster than state-of-the-art RV tools, and the extent to which it preserves their bug-detection ability. RQ2 compares Valg with two random sampling-based baselines, using the same metrics as in RQ1. RQ3 evaluates Valg's RL-based selective monitor creation under different hyperparameter values. Lastly, RQ4 evaluates how well Valg performs on evolving software.

Evaluation subjects. Table I shows summary characteristics on the 64 open-source projects that we evaluate. They are all of those for which a prior study [36] found that monitoring costs (to signal events, create monitors, find monitors, process events, etc.) dominate—*i.e.*, are greater than 50% of—RV overhead. We exclude the other 1,480 projects in that study, as instrumentation costs (not monitoring) dominate RV overhead in them. In TraceMOP experiments, we exclude 6 of these 64 projects where TraceMOP crashes or exceeds our three-day timeout. We use 160 specs of correct JDK API usage protocols that were formalized by Lee et al. [61], and used in all recent work on RV during testing of open-source projects. To simulate software evolution in RQ4, we collect up to 50 revisions per project that compile, where all tests pass with and without JavaMOP, and where hyperparameter tuning (§IV-D) finishes within our time and financial budgets. We find 1,472 revisions in 46 projects that meet these criteria.

Default hyperparameters. We set Valg's hyperparameters to commonly-used values in the literature [102]: α (learning rate) as 0.9, ϵ (exploration probability) as 0.1, δ (threshold) as 1e-5, and initial optimistic and realistic action values of 5 and 0, respectively. Also, we empirically set the max window size w to 32; this value achieved good violation preservation in our early experiments.

Baselines. §I briefly described state-of-the-art tools JavaMOP and TraceMOP. We use their implementations in the TraceMOP GitHub repository [103], which also includes modernized code for JavaMOP and its RV-Monitor [73] backend. RQ2 and RQ3 compare Valg with several baselines: (i) RS^{10} and RS^{50} randomly choose to create monitors with probability 10% and 50%, they are not feedback guided, and do not use RL; (ii) Valg^α and Valg^ϵ are configurations of Valg's RL component that use learning rate α of 0.5, and exploration probability ϵ of 50%, respectively, and use default values of other hyperparameters.

Experimental setup. For RQ1 and RQ2, we use machines with Intel® Xeon® 72-core 2.2 GHz processors and 125GB RAM (JavaMOP) or 2.6 GHz processors and 503GB RAM (TraceMOP). For hyperparameter tuning in RQ3, we use a cluster of 194 compute nodes. For RQ4, we use Google Cloud Platform (GCP) VMs running Intel® Emerald Rapids® 2-core

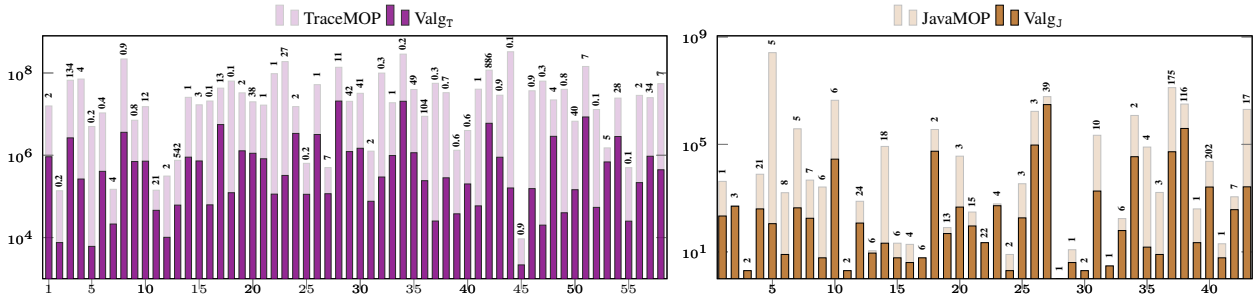

Fig. 4: Redundant parametric traces (left) and non-parametric events (right) checked by TraceMOP vs. Valg$_T$ and JavaMOP vs. Valg$_J$, respectively. Y-axes are log scale (base 10). Y-axis in the left plot does not start from zero.

2.9 GHz processors and 640GB RAM. All experiments are run in Docker containers with Ubuntu 20.04.6 LTS and Java 8. We write scripts to automate hyperparameter tuning and evolution experiments on GCP. For more accurate time measurement, each project is run on a dedicated (virtual) machine.

A. RQ1: Comparing Valg with SoTA RV tools

1) Time Comparison: Figure 3 shows relative end-to-end (darker bars) and monitoring-only (lighter bars) times for Valg$_J$ vs. JavaMOP (top) and Valg$_T$ vs. TraceMOP (bottom); lower is better. The 100% lines mark JavaMOP or TraceMOP time. Monitoring-only time is RV time minus instrumentation time (which Valg does not reduce); it shows the degree to which Valg reduces the monitoring portion of RV overhead.

Valg$_J$ vs. JavaMOP. The top plot in Figure 3 shows that Valg$_J$'s end-to-end time is faster than JavaMOP's in 56 (of 64) projects by an average of 37.5% (max: 95.1%, or 20.2x). In absolute terms, that average end-to-end time reduction is 20.2 minutes (max: 4.3 hours). Considering monitoring-only times, Valg$_J$ is faster than JavaMOP in 62 (of 64) projects by an average of 57.5% (max: 99.8%, or 625x). In absolute terms, that average monitoring-only time reduction is 22.7 minutes (max: 7.4 hours). Bars marked with "x" are projects in the top-quartile by decreasing end-to-end time, where Valg saves significant time. For example, Valg$_J$ reduces end-to-end time in project 56 from 4.9 hours to 34 minutes (tests without RV for that project take 13.9 seconds). In 8 projects, Valg$_J$ takes longer than JavaMOP (above 100% line) because the two overheads of Valg (§III-C) outweigh the savings that it provides. But, Valg$_J$ can achieve further speedup via hyperparameter tuning (§V). Also, end-to-end RV time in these 8 projects is often less than a minute (median: 4.1 minutes).

Valg$_T$ vs. TraceMOP. The bottom plot in Figure 3 shows that Valg$_T$'s end-to-end time is faster than TraceMOP's in 57 (of 58) projects by an average of 54.4% (max: 99.8%, or 551.5x). In absolute terms, that average end-to-end time reduction is 1.6 hours (max: 24.3 hours). Considering monitoring-only times, Valg$_T$ is faster than TraceMOP in all 58 projects by an average of 69.8% (max: 99.9%, or 909.1x). In absolute terms, that average monitoring-only time reduction is 24.6 minutes (max: 7.7 hours). So, TraceMOP benefits more from Valg's time savings than JavaMOP. Projects with higher end-to-end times see more savings with Valg$_T$. For example, TraceMOP's end-to-end time is 24.1, 24.1, and 24.3 hours for projects 61, 62, and 63, where Valg$_T$ reduces to only 4.2, 2.6, and 4.8 mins, respectively. Also, Valg$_T$ finishes for two projects where TraceMOP runs out of memory, and Valg$_T$ is slower in one project where TraceMOP has low end-to-end time.

2) Violation Comparison: Valg$_J$ and Valg$_T$ efficiency gains do not come at the expense of missing violations. Valg$_J$ misses only three or 0.4% of 785 violations across all projects. Those missed violations are in a project with test non-determinism where we could not evaluate Valg$_T$. The violations are in the `else` branch of an `if` statement whose `then` branch is more frequently executed than the `else` branch (§V gives details). Valg$_J$ detects all violations in this project after we adjust the learning rate (α) to a custom value or re-run multiple times.

3) Trace Comparison: Figure 4 compares numbers of redundant parametric traces (left) and redundant violating non-parametric events (right) that Valg$_T$ and Valg$_J$ (dark-colored bars) checks with those of TraceMOP and JavaMOP, respectively (light-colored bars). There, the y-axes are in log (base 10) scale because projects often produce millions of traces and events. The numbers above the bars show thousands of unique parametric traces (left) checked by TraceMOP and number of unique violating non-parametric events (right) checked by JavaMOP. The plot on the right shows only projects with a non-parametric spec violation.

Across all projects on the left, Valg$_T$ checks only 98,682,002 of all 2,720,614,555 redundant parametric traces that TraceMOP checks; a 96.4% reduction (avg: 93.5% per project). As highlights, in projects 44 and 47, TraceMOP checks 328,001,801 and 63,329,660 redundant traces, but Valg$_T$ checks only 160,498 and 20,055 redundant traces, respectively. Similarly, Valg$_J$ checks only 3,698,125 out of 292,600,394 redundant violating non-parametric events that JavaMOP checks; a 98.7% reduction (avg: 67.4% per project). These reductions in redundant traces and violating events explain the speedups that Valg provides over JavaMOP and TraceMOP.

In 10 of 58 projects, Valg$_T$ checks all unique traces that TraceMOP finds. But, across all projects, TraceMOP checks only 58.1% of unique traces (avg: 76.9% per project). Checking fewer traces in Valg$_T$ compared with TraceMOP can be a problem because Valg$_T$ could potentially miss future violations in such projects. In RQ3, we turn to the question of how to improve such low unique trace preservation ratios.

TABLE II: Valg vs. two random sampling approaches and a baseline that uses the sample-average method. Each cell: total (relative change of total) / average (average of relative change per project). ▲ and ▼: baseline is better. ▲ or ▼: Valg is better.

	E2E Time$_J$ (s)	Redundant Events	Uniq. Events	E2E Time$_t$ (s)	Redundant Traces	Uniq. Traces
RS^{10}	4,061 (▲56.3%) / 169 (▲96.9%)	5,704,369 (▲710.3%) / 228,175 (▲46436.9%)	124 (▼39.8%) / 5.0 (▼30.5%)	5,368 (▼40.7%) / 233 (▼5.0%)	61,195,416 (▲144.3%) / 2,549,809 (▲507.8%)	86,258 (▼66.1%) / 3,594 (▼36.3%)
RS^{50}	4,153 (▲59.8%) / 166 (▲123.7%)	9,140,459 (▲1198.4%) / 365,618 (▲46741.0%)	180 (▼12.6%) / 7.2 (▼10.0%)	12,359 (▲36.6%) / 494 (▲51.4%)	354,277,760 (▲1314.2%) / 14,171,110 (▲4637.5%)	239,698 (▼5.7%) / 9,588 (▲27.8%)
Valg	**2,599 / 104**	**703,992 / 28,160**	**206 / 8.2**	**9,050 / 362**	**25,051,852 / 1,002,074**	**254,197 / 10,168**

B. RQ2: Comparing Valg with random sampling baselines

We compare Valg with RS^{10} and RS^{50}, which randomly create monitors with probability 10% and 50%, respectively. We use 25 randomly selected projects with violations and whose end-to-end time for TraceMOP is less than two hours.

As Table II shows, Valg outperforms these baselines: it (i) is faster, (ii) checks fewer redundant traces and events, and (iii) finds more violations. Notably, RS^{50} is up to 123.7% slower, and checks 46,741% and 4,637.5% more redundant events and traces than Valg$_J$ and Valg$_T$, respectively. These percentages are computed as $((x-y)/y*100)$, where x and y is the number of redundant traces checked by a baseline and Valg, respectively. RS^{10} is slightly faster than Valg$_T$, but it checks 66.1% fewer unique traces and misses 39.8% of violations. RS^{10} also misses 27.2% of violations that RS^{50} finds. But, RS^{50} checks 488.1 and 1,169.9 percentage points (pp) more redundant events and traces, respectively, than RS^{10}. RS^{10} and RS^{50} have similar end-to-end times for JavaMOP (56.3% vs 59.8% relative overheads). But, RS^{50} is 77.3 pp slower than RS^{10} for TraceMOP.

C. RQ3: Investigating the impact of hyperparameters

We investigate whether hyperparameter tuning can help Valg better preserve unique traces, while maintaining its efficiency. Hyperparameter tuning is costly in general [24], and specifically in our RV setting. So, we first perform a small formative study using 25 projects from RQ2. Specifically, we change Valg's most influential hyperparameters: learning rate α and exploration probability ϵ. To avoid confounding effects, we keep all other hyperparameters constant, and change only α and ϵ to 0.5 one at a time. We call the resulting configurations Valg$^{\alpha}$ and Valg$^{\epsilon}$, respectively. Valg$^{\alpha}$ and Valg$^{\epsilon}$ find all 206 violations and check 8,677 and 31,595 more unique traces in 13 and 19 projects than before tuning.

Tuning setup. To better explore the space of hyperparameter values, we use Optuna [1] to automate the hyperparameter tuning process. Optuna allows one to define custom objective functions, and it finds optimal hyperparameter values based on repetitive trials. To design our experiments, we make two initial attempts on six projects: (i) *single*-objective to maximize the number of unique traces checked, and (ii) *multi*-objective to maximize unique traces checked and minimize time. Single-objective tuning checks more unique traces in five projects, with at most 4% additional time overhead compared to multi-objective tuning. So, we use single-objective in the rest of this RQ3. Also, Optuna often finds more optimal hyperparameters after 50 trials. So, we use 100 trials per iteration and three iterations per project.

TABLE III: Unique trace preservation after tuning.

Uniq. Traces$_d$ → Uniq. Traces$_o$ | Total Uniq. Traces	
5,370 (**14.1%**) → 18,792 (**49.5%**) | 37,977	1,997 (**40.1%**) → 4,092 (**82.3%**) | 4,975
29,142 (**5.4%**) → 542,383 (**100.0%**) | 542,432	3,390 (**69.8%**) → 4,638 (**95.5%**) | 4,859
22,646 (**47.3%**) → 35,411 (**74.0%**) | 47,881	889 (**75.2%**) → 1,087 (**92.0%**) | 1,182
26,497 (**25.3%**) → 89,531 (**85.5%**) | 104,715	989 (**57.2%**) → 1,729 (**100.0%**) | 1,729
8,403 (**72.9%**) → 11,503 (**99.8%**) | 11,527	1,971 (**69.2%**) → 2,090 (**73.4%**) | 2,846
3,554 (**28.6%**) → 10,575 (**85.2%**) | 12,412	451 (**48.8%**) → 923 (**99.8%**) | 925
26,369 (**58.9%**) → 38,280 (**85.5%**) | 44,765	818 (**38.8%**) → 2,027 (**96.2%**) | 2,107
3,060 (**12.4%**) → 20,246 (**82.0%**) | 24,689	59 (**75.6%**) → 81 (**103.8%**) | 78
5,803 (**76.5%**) → 7,457 (**98.3%**) | 7,582	299 (**19.6%**) → 946 (**62.2%**) | 1,522
16,165 (**46.5%**) → 34,904 (**100.4%**) | 34,774	673 (**61.4%**) → 1,075 (**98.1%**) | 1,096
Σ **Other 38 Projects:** 1,055,879 (**87.9%**) → 1,203,751 (**100.2%**) | 1,201,306	
Σ **All Projects: 1,214,424 (58.1%) → 2,031,521 (97.1%) | 2,091,378**	

Tuning helps Valg check more unique traces. Table III shows how many unique trace Valg checks before (Uniq. Traces$_d$) and after (Unique. Traces$_o$) tuning. Due to space limits, here we only present 20 projects whose preservation ratio is lower than the average (76.7%). The other 38 projects preserved 87.9% of their unique traces in total. Across all 58 projects, out of 2,091,378 unique traces, Valg checks 1,214,424 (58.1%) before tuning. That rate improves to 2,031,521 (97.1%) after tuning. As a highlight, in one project Valg$_T$ checks 29,142 (5.4%) unique traces before tuning, and 542,383 (100%) after tuning. Some projects improve to more than 100.0% due to nondeterministic executions that cause Valg to check different traces in different runs on the same project revisions and tests. To partially mitigate the effect of non-determinism, we run TraceMOP three times and take the average unique-trace count as the baseline. When analyzed per project, hyperparameter tuning improves Valg's average ratio of unique traces checked from 76.7% to 95.1%. RQ4 reports on the impact of tuned hyperparameter values on Valg's efficiency as software evolves.

D. RQ4: Investigating the impact of software evolution

We compare Valg with default hyperparameters vs. Valg with tuned hyperparameters on 1,472 versions of 46 projects.
1. Valg's speedups as software evolves. The left plot in Figure 5 shows time-saving ratios across all revisions per project (lower is better). Each ratio is $((\sum_{i=1}^{n} x_i)/(\sum_{i=1}^{n} y_i)) * 100$, where x is end-to-end time for Valg$_J$ (Valg$_T$), y is end-to-end time for JavaMOP (TraceMOP), and n is the revision count; lower is better. The dashed bars show Valg's time with default hyperparameters, and the solid bars above show slowdowns after tuning. When Valg is faster with tuned hyperparameters, no solid bars are shown. *Overall, Valg with*

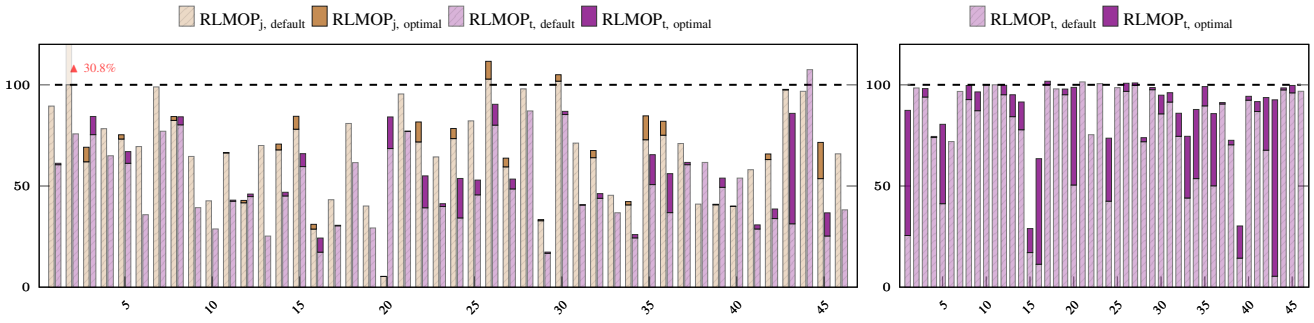

Fig. 5: Total Time Savings and Unique Trace Preservation Across Versions

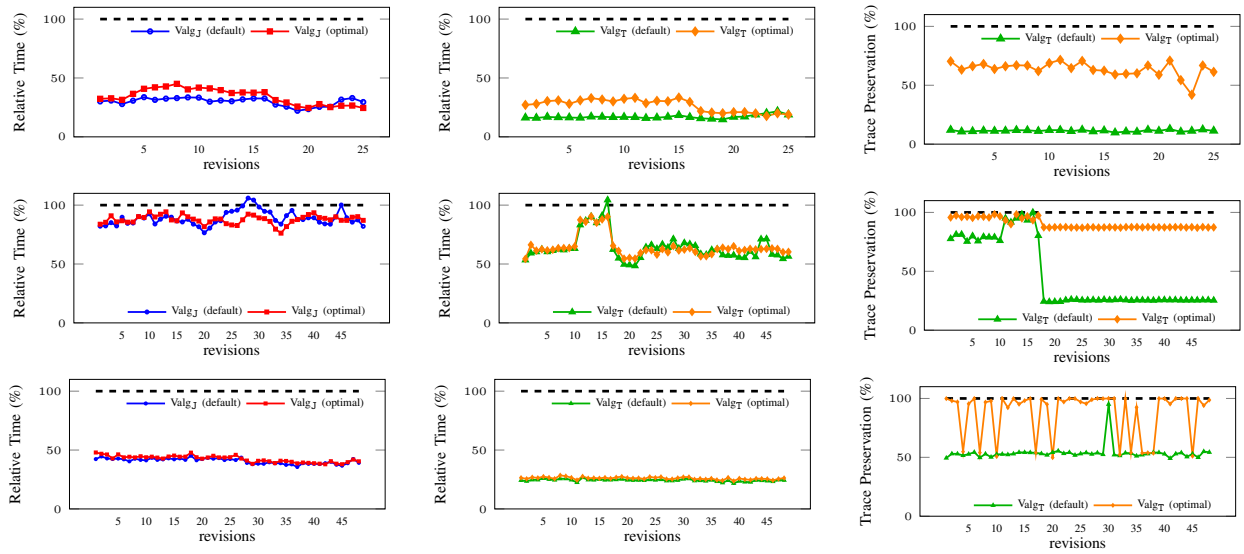

Fig. 6: Effect of hyperparameter tuning on Valg_J time (left), Valg_T time (middle), and unique traces checked (right). The projects are `cassette-nibbler` [52], `ASM-NonClassloadingExtensions` [34], and `jpoker` [22], from top to bottom.

tuned hyperparameters is still more efficient than JavaMOP and TraceMOP (the 100% line).

The average per-project slowdowns among all 46 projects amount to 0.7 (max: 59.5) minutes and 6.1 (201.7) minutes, respectively. Using tuned hyperparameters is slower than using default ones in 723 and 826 of 1,472 revisions across all 46 projects, with average per-revision slowdowns of 4.1% (max: 34.0%) and 2.0% (max: 21.6%) for Valg_T and Valg_J, respectively. These per-sha slowdowns amount to an average of 0.5 (max: 2.8) minutes and 1.4 (max: 7.6) minutes, respectively. These averages are affected by projects and revisions where using default hyperparameters is *slower* than using tuned ones; there are 22 and 15 such projects for Valg_J and Valg_T, respectively. The existence of such projects shows that tuned hyperparameters can preserve (or even provide more) efficiency when using Valg.

2. Unique traces checked by Valg as software evolves. The right plot in Figure 5 shows proportions of TraceMOP's unique traces that Valg checks across all revisions per project (higher is better). Each ratio is computed using the same formula as for time, but higher values are better here. The dashed bars show Valg's unique trace preservation with default hyperparameters, and the solid bars above show how much the ratio is improved after tuning. *Overall, using tuned hyperparameters improves the proportion of TraceMOP's unique traces checked by* Valg_T *(the 100% line).*

In 35 out of 46 projects, Valg_T with tuned hyperparameters checks more unique traces than Valg_T with default ones. Valg_T after tuning checks an average of 11.6% (max: 87.2%) more unique traces than before tuning per project. These translate to an average of 531,547 (max: 22,699,326) more unique traces. After tuning, Valg_T checks more traces in 1,107 of 1,472 revisions, by an average of 12.6% (max: 94.7%) and 16,611 (max: 513,384) per revision. But, in 11 projects, Valg checks 6.8% (max: 24.1%) less unique traces on average after tuning.

3. More Valg trends as software evolves. Figure 6 shows how hyperparameter tuning affects time and unique trace preservation. For the lack of space, we present three projects (122 revisions) as examples. The plots for other projects can be found in our repository [97]. The dashed lines at 100% represent JavaMOP and TraceMOP. Tuned hyperparameters still bring much speedup; compared to default hyperparameters, they have average slowdowns of 4.2% and 15.4% for JavaMOP and TraceMOP, respectively. With such small loss

```
boolean reorderIncorrectShiftTransition(...) {
 ...
 if (lastBinary.side == Transition.Side.RIGHT) {
  for (int i = 0; i<leftoverBinary.size(); ++i)
   cursor.add(new Transition(..));
   cursor.add(new Transition(..)); }
 else {
  cursor.add(new Transition(..)); // violation
  for (int i = 0; i<leftoverBinary.size()-1; ++i)
   cursor.add(new Transition(..)); // violation
   cursor.add(new Transition(..)); } // violation
 return true; }
```

Fig. 7: Unique violations missed by Valg.

in efficiency, Valg_T preserves 52.2%, 61.8%, and 34.2% more unique traces for these three projects. Also, the high ratios of checked traces are preserved across many versions.

V. Discussion

Comparison with evolution-aware RV. Our evolution-related experiments may make the reader curious about how Valg compares with eMOP [62]. Unlike Valg, which is evolution-*unaware*, eMOP is an evolution-*aware* RV that aims to speed up RV by taking code changes into account. In a new code revision, eMOP re-monitors all tests against a subset of specs related to code affected by changes. We integrate Valg with two eMOP variants: (i) ps_1^c uses a more conservative change-impact analysis (CIA), so it selects more specs as being related to affected code, and is therefore slower; and (ii) ps_3^{c1} uses less conservative CIA, selects fewer specs, and is therefore faster. A *safe* evolution-aware RV technique is one that finds all new violations after code changes [63]. ps_1^c is designed to be safe (modulo static analysis limitations), while ps_3^{c1} is designed to be fast at the expense of safety.

We use 63 projects with 2,157 revisions from our evaluation subjects, and we only use Valg_J for these experiments, since the eMOP tool [112] only works with JavaMOP. By themselves, ps_1^c and ps_3^{c1} are 68.7% and 84.8% faster than JavaMOP, respectively, across all these revisions. With Valg integration, ps_1^c and ps_3^{c1} become even faster by 2.8% and 11.5%, respectively. Also, Valg preserves 10,534 and 9,152 among 10,597 and 9,162 new violations that ps_1^c and ps_3^{c1} find, respectively. These results show that Valg is complementary and orthogonal to evolution-aware RV with little loss (0.4% on average) in violation preservation.

Ablation. To evaluate if selective monitor creation or event signaling contributes more to Valg's efficiency, we enable only selective monitor creation in a variant called Valg_sm. On 63 projects in our evaluation, Valg_sm speeds up JavaMOP and TraceMOP by 8.9% and 84.4% on average, respectively, compared to 26.9% and 89.2% for Valg_J. So, RL-guided selective monitor creation contributes more, but selective non-parametric event signaling provides additional speedup.

Speedup rates $\neq$ redundancy reduction rates. Valg's speedups (avg: 54.4%) are lower than its redundant trace and event reduction rates (avg: 96.4%), due in part to (i) Valg's overheads (§III-C); (ii) instrumentation costs, which Valg does not address; and (iii) unique traces being on average much longer than redundant ones—monitoring fewer traces does not always translate to proportionally less monitoring costs. As an example of the latter, in one project, the average length of its 66,770,809 redundant traces is 718. But, the 134,561 unique traces in this project have 15,601 events on average.

Missed violations. Probabilistic aspects of RL cause Valg to miss the three violations in Figure 7, simplified from CoreNLP [33]. RV reports these three violations because the `ListIterator_Set` spec [69], which checks that `add` should not come before `set`, is violated. There, the program execution mostly explores the `if` block, misguiding Valg to predict that traces in this method are redundant and to not create monitors that could check the related violating traces. But, there are occasional executions of the `else` block, and Valg misses those violations. Interestingly, our manual analysis confirms that all three violations are false positives due to imprecision in the spec. So, these misses have no impact on the RV's bug detection ability in this project. Also, custom hyperparameter values or running Valg multiple times find these violations.

Memory overheads. To evaluate how much additional memory Valg uses compared to the baselines, we randomly selected 9 projects and profiled their memory usage. Valg used 16.6% and 25.3% less memory for JavaMOP and TraceMOP, respectively. Valg needs to store encoded trace information whereas JavaMOP uses references to prior memory states, and also uses complex data structures to track parameter instances and violating locations. However, Valg uses an efficient encoding mechanism for traces and significantly reduces monitor creation, which offset its overhead.

Limitations. (i) Theoretically, if events occur on objects that are no longer within the max window size w, Valg could wrongly report violations in suffixes of traces for such objects. Our use of $w = 32$ works well for these projects, and we observe no such wrongly-reported violations in multiple runs of our experiments. Also, we do not set w for TraceMOP, so that we can capture all unique traces. (ii) Some projects see increased time with Valg due to Valg's overheads. But, hyperparameter tuning is not limited to unique trace preservation. One could also tune hyperparameters to minimize time. As a proof-of-concept, we try a different value for α and ϵ each, on 25 projects. Valg_J and Valg_T show additional time savings in 12 and 10 projects, respectively. (iii) Projects can have non-deterministic test executions, so RV can check different unique traces in different runs of the same code. This limitation has also been observed in recent work [35], [38], and it affects our comparisons based on unique-trace counts. So, we run TraceMOP three times and take the union of its traces to partially mitigate effects of non-deterministic executions.

Future work. Valg opens a new direction of learning-based optimizations for RV and shows promising results. But, a lot of future research is needed. We highlight four directions. First, work is needed to speed up hyperparameter tuning, which took 38.5 hours and $9.6 per project on average. Even though tuning is an offline cost, many users are unlikely to be able to afford it. Second, metrics and their automated measurement are needed to detect when re-tuning is needed as code evolves.

Third, our tuning focuses on two parameters; future work can investigate if tuning all four hyperparameters improves Valg's speedups and unique traces checked. Finally, we have only explored few simple RL algorithms. Future work should investigate other algorithms to potentially adapt for RV. Beyond these RL-related directions, future work can investigate the combination and comparison of Valg and other orthogonal and complementary techniques like regression test selection [25], [26], [29]–[32], [40], [41], [58], [65], [67], [70], [71], [83], [90], [91], [95], [109], [111], [113]–[115], iMOP [37], and LazyMOP [35]. Also, future work can investigate whether techniques for detecting and mitigating flaky tests [9], [39], [59], [72], [84], [94], [96] could be adapted to help address the problem of non-deterministic traces in RV. (All tests pass in our experiments.)

Threats to validity. Our evaluation results may not generalize beyond the projects we use. But, we show that Valg is effective across 1,472 revisions. Also, other outcomes could result from using different specs. But, the key idea in Valg is spec- and test-agnostic, so we expect similar speedups. Many other RV techniques [13], [14], [21], [53], [76], [88] and tools [4], [11], [12], [55] exist. But, JavaMOP is a mature and widely-cited tool in the RV community, on which TraceMOP is based. We write scripts to automate experiments; they are subject to errors but we are releasing them for external validation.

VI. Related Work

Selective monitoring. QVM [4] and SMCO [49] are designed for production settings; they adjust monitor creation and event signaling rates over a long period to keep RV overhead under a user-provided limit. Time-triggered RV [15], [16], [108] also targets production settings, but for time-sensitive domains; it only monitors at fixed time intervals to keep aggregated RV overhead over a long period within a limit. Unlike these prior works, Valg (i) is designed for testing, where one often does not have a long time span over which to adjust monitoring rates; (ii) does not require specifying a time budget; (iii) is evaluated on evolving software; (iv) aims to reduce redundant traces and events monitored; and (v) uses RL. We would love to compare Valg with these techniques, but they have no publicly available tools (*e.g.*, QVM is proprietary). Purandare et al. [87] proposed transformations to reduce redundant events signaled in loops. But, their approach is limited: it cannot reduce redundancies that are not related to loops, *e.g.*, those due to multiple tests calling a loop-free method many times. Even within loops, their static analysis has limited support for handling events in exception-handling code. Also, their approach requires expensive (hours-long) static analysis per program and spec pair, and was not evaluated during testing of evolving software. Finally, they have no publicly available tool. Valg does not have these limitations.

Other approaches for reducing RV overhead. Several techniques [10], [12], [37], [81], [98]–[101] reduce instrumentation costs, which can dominate RV overhead [36]. Valg reduces monitoring costs, which dominate RV overhead in all projects that we evaluate. But, Valg is complementary and could be combined with these instrumentation-driven techniques in the future. Evolution-aware RV [37], [62], [63], [112] does not optimize RV directly; rather, it indirectly speeds up RV during regression testing by re-monitoring only parts of code affected by changes. Valg is complementary and orthogonal (§V), it directly optimizes the RV, regardless of how it is applied. Other techniques speed up RV via ahead-of-time static analysis [7], [10], [13], [14], [23], specialized data structures [21], [73], [88] or algorithms [18], [19], [45], [53], [76], [77], [93]. Many of these data structures and algorithms are implemented in JavaMOP and TraceMOP, which Valg optimizes. Also, Valg is orthogonal and complementary to the static-analysis based approaches, none of which has a tool that we can evaluate or was evaluated at scale during testing of evolving code. Future work can revisit these static-analysis based techniques in CI.

Learning-based techniques and RV. RVPrio [78] used classifiers to classify spec violations as true bugs or false alarms. Some techniques used conformal prediction [68] and BDD representation [80] to predict future events. Others used Hidden Markov Models [8], [80], deep neural networks [17], or Bayesian networks [50] to estimate the likelihood that a monitor will end up in a violating state. In the other direction, RV has been used for quality assurance of learning-based autonomous systems [2], [86], [110] and deep neural networks [42], [46], [105], [107]. RTSA [60] monitors autopilots and switches to the recovery controller in unsafe scenarios. To balance between safety and efficiency, it uses RL to learn an optimal switching policy. None of these works target the reduction of redundant monitoring during testing, and Valg is the first to use RL to speed up RV to the best of our knowledge.

VII. Conclusion

We introduced Valg, the first technique that speeds up RV during testing by using on-the-fly feedback from prior monitors and events to reduce the amount of redundant traces and events that are checked subsequently. Valg is also the first to use RL to speed up RV. Valg achieves speedups that are as high as 20.2x and 551.5x, compared to two SoTA tools, while finding 99.6% of violations that they find. Also, Valg reduces redundant traces and events by up to 98.7%. We find that RL hyperparameter tuning helps improve the number of unique traces checked—a stronger measure of Valg's effectiveness than violations found—from 76.7% to 95.1% on average. Valg opens up a new direction in using learning-based approaches to speed up RV and we highlight some exciting future directions.

Acknowledgment

We thank Kevin Guan for helping us set up JavaMOP and TraceMOP during the early stages of this research, and Pengyue Jiang, Stephen Shen, and the anonymous reviewers for their valuable comments. We also thank Google for their Cloud Platform credits. This work is partially supported by a Google Cyber NYC Institutional Research Award, an Intel Rising Star Faculty Award, and the United States National Science Foundation (NSF) under Grant Nos. CCF-2045596, CCF-2319473, CCF-2403035, and CCF-2525243.

REFERENCES

[1] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *KDD*, 2019.

[2] D. An, J. Liu, M. Zhang, X. Chen, M. Chen, and H. Sun, "Uncertainty modeling and runtime verification for autonomous vehicles driving control: A machine learning-based approach," *JSS*, vol. 167, 2020.

[3] "Appendable_threadsafe.mop," https://github.com/SoftEngResearch/tracemop/blob/master/scripts/props/Appendable_ThreadSafe.mop.

[4] M. Arnold, M. Vechev, and E. Yahav, "QVM: An efficient runtime for detecting defects in deployed systems," in *OOPSLA*, 2008.

[5] "ARTCAT: Autonomic response to cyber-attack," https://grammatech.github.io/prj/artcat.

[6] Asterisk-java, "Asterisk-java," 2025, https://github.com/asterisk-java/asterisk-java.

[7] P. Avgustinov, J. Tibble, and O. de Moor, "Making trace monitors feasible," in *OOPSLA*, 2007.

[8] E. Bartocci, R. Grosu, A. Karmarkar, S. A. Smolka, S. D. Stoller, E. Zadok, and J. Seyster, "Adaptive runtime verification," in *RV*, 2013.

[9] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *ICSE*, 2018.

[10] E. Bodden, "Efficient hybrid typestate analysis by determining continuation-equivalent states," in *ICSE*, 2010.

[11] ——, "MOPBox: A library approach to runtime verification," in *RV*, 2011.

[12] E. Bodden, L. Hendren, P. Lam, O. Lhoták, and N. A. Naeem, "Collaborative runtime verification with Tracematches," in *RV*, 2007.

[13] E. Bodden, L. Hendren, and O. Lhoták, "A staged static program analysis to improve the performance of runtime monitoring," in *ECOOP*, 2007.

[14] E. Bodden, P. Lam, and L. Hendren, "Finding programming errors earlier by evaluating runtime monitors ahead-of-time," in *FSE*, 2008.

[15] B. Bonakdarpour, S. Navabpour, and S. Fischmeister, "Sampling-based runtime verification," in *FM*, vol. 6664.

[16] ——, "Time-triggered runtime verification," in *FMSD*, vol. 43, 2013.

[17] F. Cairoli, L. Bortolussi, and N. Paoletti, "Neural predictive monitoring under partial observability," in *RV*, 2021.

[18] F. Chen, P. O. Meredith, D. Jin, and G. Roşu, "Efficient formalism-independent monitoring of parametric properties," in *ASE*, 2009.

[19] F. Chen and G. Roşu, "Parametric trace slicing and monitoring," in *TACAS*, 2009.

[20] D. B. de Oliveira, "Efficient runtime verification for the Linux kernel," https://research.redhat.com/blog/article/efficient-runtime-verification-for-the-linux-kernel.

[21] N. Decker, J. Harder, T. Scheffel, M. Schmitz, and D. Thoma, "Runtime monitoring with union-find structures," in *TACAS*, 2016.

[22] Dperezcabrera, "Jpoker," 2025, https://github.com/dperezcabrera/jpoker.

[23] M. B. Dwyer, R. Purandare, and S. Person, "Runtime verification in context: Can optimizing error detection improve fault diagnosis?" in *RV*, 2010.

[24] T. Eimer, M. Lindauer, and R. Raileanu, "Hyperparameters in reinforcement learning and how to tune them," *arXiv preprint arXiv:2306.01324*, 2023.

[25] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *IST*, vol. 52, no. 1, 2010.

[26] E. Engström, M. Skoglund, and P. Runeson, "Empirical evaluations of regression test selection techniques: A systematic review," in *ESEM*, 2008.

[27] U. Erlingsson and F. B. Schneider, "IRM enforcement of Java stack inspection," in *IEEE S&P*, 2000.

[28] M. Ghasemi and D. Ebrahimi, "Introduction to reinforcement learning," 2024.

[29] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *ICSE Demo*, 2015.

[30] ——, "Practical regression test selection with dynamic file dependencies," in *ISSTA*, 2015.

[31] M. Gligoric, R. Majumdar, R. Sharma, L. Eloussi, and D. Marinov, "Regression test selection for distributed software histories," in *CAV*, 2014.

[32] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov, "An empirical evaluation and comparison of manual and automated test selection," in *ASE*, 2014.

[33] T. S. N. Group, "Stanford CoreNLP: A Java suite of core NLP tools," https://github.com/stanfordnlp/CoreNLP, 2025.

[34] Grundlefleck, "ASM-NonClassloadingExtensions," 2025, https://github.com/Grundlefleck/ASM-NonClassloadingExtensions.

[35] K. Guan, M. d'Amorim, and O. Legunsen, "Faster explicit-trace monitoring-oriented programming for runtime verification of software tests," in *OOPSLA*, 2025.

[36] K. Guan and O. Legunsen, "An in-depth study of runtime verification overheads during software testing," in *ISSTA*, 2024.

[37] ——, "Instrumentation-driven evolution-aware runtime verification," in *ICSE*, 2024.

[38] ——, "TraceMOP: An explicit-trace runtime verification tool for Java," in *FSE Demo*, 2025.

[39] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov, "NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications," in *FSE Demo*, 2016, pp. 993–997.

[40] A. Gyori, O. Legunsen, F. Hariri, and D. Marinov, "Evaluating regression test selection opportunities in a very large open-source ecosystem," in *ISSRE*, 2018.

[41] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression test selection for Java software," in *OOPSLA*, 2001.

[42] V. Hashemi, J. Křetínský, S. Rieder, T. Schön, and J. Vorhoff, "Gaussian-based and outside-the-box runtime monitoring join forces," in *RV*, 2024.

[43] K. Havelund and G. Roşu, "Monitoring Java programs with Java PathExplorer," in *RV*, 2001.

[44] ——, "Monitoring programs using rewriting," in *ASE*, 2001.

[45] ——, "Synthesizing monitors for safety properties," in *TACAS*, 2002.

[46] W. He, C. Wu, and S. Bensalem, "Box-based monitor approach for out-of-distribution detection in yolo: An exploratory study," in *RV*, 2025.

[47] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: Assurance, security, and flexibility," in *FSE*, 2017.

[48] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *ASE*, 2016.

[49] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. Stoller, and E. Zadok, "Software monitoring with controllable overhead," *IJSTTT*, vol. 14, no. 3, 2012.

[50] M. Jaeger, K. G. Larsen, and A. Tibo, "From statistical model checking to run-time monitoring using a Bayesian network approach," in *RV*, 2020.

[51] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, "Q-learning algorithms: A comprehensive classification and applications," *IEEE Access*, vol. 7, 2019.

[52] Jim, "Cassette-nibbler: Data recovery from 8-bit computer cassette tapes," 2017, https://github.com/eightbitjim/cassette-nibbler.

[53] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu, "Garbage collection for monitoring parametric properties," in *PLDI*, 2011.

[54] D. Jin, P. O. Meredith, C. Lee, and G. Roşu, "JavaMOP: Efficient parametric runtime monitoring framework," in *ICSE Demo*, 2012.

[55] M. Karaorman and J. Freeman, "jMonitor: Java runtime event specification and monitoring library," in *RV*, 2004.

[56] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP*, 1997.

[57] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky, "Formally specified monitoring of temporal properties," in *ECRTS*, 1999.

[58] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, "Change impact identification in object oriented software maintenance," in *ICSM*, 1994.

[59] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *ICST*, 2019.

[60] C. Lazarus, J. G. Lopez, and M. J. Kochenderfer, "Runtime safety assurance using reinforcement learning," in *DASC*, 2020.

[61] C. Lee, D. Jin, P. O. Meredith, and G. Roşu, "Towards categorizing and formalizing the JDK API," Computer Science Dept., UIUC, Tech. Rep., 2012.

[62] O. Legunsen, D. Marinov, and G. Roşu, "Evolution-aware monitoring-oriented programming," in *ICSE NIER*, 2015.

[63] O. Legunsen, Y. Zhang, M. Hadzi-Tanovic, G. Roşu, and D. Marinov, "Techniques for evolution-aware runtime verification," in *ICST*, 2019.

[64] O. Legunsen, N. A. Awar, X. Xu, W. U. Hassan, G. Roşu, and D. Marinov, "How effective are existing Java API specifications for finding bugs during runtime verification?" *ASE Journal*, vol. 26, no. 4, 2019.

[65] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *FSE*, 2016.

[66] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov, "How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications," in *ASE*, 2016.

[67] O. Legunsen, A. Shi, and D. Marinov, "STARTS: STAtic Regression Test Selection," in *ASE*, 2017.

[68] L. Lindemann, X. Qin, J. V. Deshmukh, and G. J. Pappas, "Conformal prediction for STL runtime verification," in *ICCPS*, 2023, p. 142–153.

[69] "Listiterator_set.mop," https://github.com/SoftEngResearch/tracemop/blob/master/scripts/props/ListIterator_Set.mop.

[70] Y. Liu, J. Zhang, P. Nie, M. Gligoric, and O. Legunsen, "More precise regression test selection via reasoning about semantics-modifying changes," in *ISSTA*, 2023.

[71] E. Lundsten, "EALRTS: A predictive regression test selection tool," Master's thesis, KTH Royal Institute of Technology, Sweden, 2019.

[72] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *FSE*, 2014.

[73] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Şerbănuţă, and G. Roşu, "RV-Monitor: Efficient parametric runtime verification with simultaneous properties," in *RV*, 2014.

[74] A. Maroti, "RBED: Reward based epsilon decay," *arXiv preprint arXiv:1910.13701*, 2019.

[75] "Math_contendedrandom.mop," https://github.com/SoftEngResearch/tracemop/blob/master/scripts/props/Math_ContendedRandom.mop.

[76] P. Meredith and G. Roşu, "Efficient parametric runtime verification with deterministic string rewriting," in *ASE*, 2013.

[77] P. Meredith, D. Jin, F. Chen, and G. Roşu, "Efficient monitoring of parametric context-free patterns," in *ASE*, 2008.

[78] B. Miranda, I. Lima, O. Legunsen, and M. d'Amorim, "Prioritizing runtime verification violations," in *ICST*, 2020.

[79] E. F. Moore, "Gedanken-experiments on sequential machines," in *Automata Studies*, 1956.

[80] O. Moran and D. Peled, "Runtime verification prediction for traces with data," in *RV*, 2023.

[81] S. Navabpour, C. W. W. Wu, B. Bonakdarpour, and S. Fischmeister, "Efficient techniques for near-optimal instrumentation in time-triggered runtime verification," in *RV*, 2011.

[82] Oracle, "Math.random() method," 2006, https://docs.oracle.com/javase/6/docs/api/java/lang/Math.html#random%28%29.

[83] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *FSE*, 2004.

[84] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *ICSME*, 2017.

[85] A. Pnueli, "The temporal logic of programs," in *FOCS*, 1977.

[86] C. S. Păsăreanu, R. Mangal, D. Gopinath, and H. Yu, "Assumption generation for learning-enabled autonomous systems," in *RV*, Berlin, Heidelberg, 2023.

[87] R. Purandare, M. B. Dwyer, and S. Elbaum, "Monitor optimization via stutter-equivalent loop transformation," in *OOPSLA*, 2010.

[88] ——, "Optimizing monitoring of finite state properties through monitor compaction," in *ISSTA*, 2013.

[89] S. M. Ross, *Introduction to probability models*. Academic Press, 2014.

[90] G. Rothermel and M. J. Harrold, "A safe, efficient algorithm for regression test selection," in *ICSM*, 1993.

[91] ——, "A safe, efficient regression test selection technique," *TOSEM*, vol. 6, no. 2, 1997.

[92] Sbesada, "Java.math.expression.parser," 2023, https://github.com/sbesada/java.math.expression.parser.

[93] K. Sen, G. Roşu, and G. Agha, "Generating optimal linear temporal logic monitors by coinduction," in *Advances in Computing Science*, 2003.

[94] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *ICST*, 2016.

[95] A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen, "Reflection-aware static regression test selection," in *OOPSLA*, 2019.

[96] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: A framework for automatically fixing order-dependent flaky tests," in *FSE*, 2019.

[97] SoftEngResearch, "Valg," 2025, https://github.com/SoftEngResearch/Valg.

[98] C. Soueidi and Y. Falcone, "Bridging the gap: A focused DSL for RV-oriented instrumentation with BISM," in *RV*, 2023.

[99] ——, "Instrumentation for RV: From basic monitoring to advanced use cases," in *RV*, 2023.

[100] C. Soueidi, Y. Falcone, and S. Hallé, "Dynamic program analysis with flexible instrumentation and complex event processing," in *ISSRE*, 2023.

[101] C. Soueidi, M. Monnier, and Y. Falcone, "Efficient and expressive bytecode-level instrumentation for Java programs," *IJSTTT*, vol. 25, no. 4, 2023.

[102] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," 2018.

[103] "TraceMOP: A trace-aware runtime verification tool for Java." 2024, https://github.com/SoftEngResearch/tracemop.

[104] J. N. Tsitsiklis, "Asynchronous stochastic approximation and Q-learning," *Machine Learning*, vol. 16, Sep. 1994.

[105] M. Usman, D. Gopinath, Y. Sun, and C. S. Păsăreanu, "Rule-based runtime mitigation against poison attacks on neural networks," in *RV*, 2022.

[106] L. Weng, "Policy gradient algorithms," 2018, https://lilianweng.github.io/posts/2018-04-08-policy-gradient.

[107] C. Wu, Y. Falcone, and S. Bensalem, "Customizable reference runtime monitoring of neural networks using resolution boxes," in *RV*, 2023.

[108] C. W. W. Wu, D. Kumar, B. Bonakdarpour, and S. Fischmeister, "Reducing monitoring overhead by integrating event- and time-triggered techniques," in *RV*, 2013.

[109] G. Xu and A. Rountev, "Regression test selection for AspectJ software," in *ICSE*, 2007.

[110] B. Yalcinkaya, H. Torfah, D. J. Fremont, and S. A. Seshia, "Compositional simulation-based analysis of AI-based autonomous systems for markovian specifications," in *RV*, 2023.

[111] U. Yilmaz, "A method for selecting regression test cases based on software changes and software faults," Master's thesis, Hacettepe University, Turkey, 2019.

[112] A. Yorihiro, P. Jiang, V. Marques, B. Carleton, and O. Legunsen, "eMOP: A Maven plugin for evolution-aware runtime verification," in *RV*, 2023.

[113] J. Zhang, Y. Liu, M. Gligoric, O. Legunsen, and A. Shi, "Comparing and combining analysis-based and learning-based regression test selection," in *AST*, 2022.

[114] L. Zhang, "Hybrid regression test selection," in *ICSE*, 2018.

[115] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric, "A framework for checking regression test selection tools," in *ICSE*, 2019.