Valg: A Fast Reinforcement Learning Based Runtime Verification Tool for Java

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

Abstract

Runtime Verification (RV) dynamically monitors whether traces—sequences of program events like method calls—violate formal specifications. RV helped find many bugs, but it incurs high runtime overheads. Prior work showed that 99.87% of monitored traces are redundant: they are identical to the other 0.13%. So, we recently proposed a new technique based on reinforcement learning (RL) to speed up RV by reducing redundant monitoring. This paper presents Valg, a tool that implements the technique for Java. Compared to our previous prototype, Valg adds (i) per-spec hyperparameters, (ii) RL trajectory saving, (iii) offline hyperparameter tuning, and (iv) performance optimizations. We also integrate Valg with the main development branch of JavaMOP and TraceMOP, two state-of-the-art RV tools, and fix a long-standing specification bug. On 56 open-source projects, Valg with tuned hyperparameters is up to 4.2x and 1.9x faster than our prototype when applied to JavaMOP and TraceMOP, respectively, and improves unique trace preservation by up to 94.5pp. Valg's offline hyperparameter tuning is orders of magnitude faster than our prototype's online tuning, and our optimizations make JavaMOP, TraceMOP, and Valg faster. Valg is open-sourced at: https://github.com/SoftEngResearch/tracemop, and a video demo can be found at: https://youtu.be/_QCyHaa_ICc.

CCS Concepts

• **Software and its engineering → Software testing and debugging**;

Keywords

Runtime Verification, Software Testing, Reinforcement Learning

ACM Reference Format:
Shinhae Kim, Saikat Dutta, and Owolabi Legunsen. 2026. Valg: A Fast Reinforcement Learning Based Runtime Verification Tool for Java. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-Companion '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3774748.3787620

1 Introduction

Runtime Verification (RV) [11, 15, 17] monitors program executions against formal specifications (*specs*). An RV tool takes a program and specs as inputs, and instruments the program so that spec-relevant *events*, e.g., method calls, are signaled at runtime. Then, RV creates *monitors* at runtime to check event sequences, i.e., *traces*, against the specs, and report any violations. RV found hundreds of bugs by monitoring *passing* tests in many projects against specs of JDK API usage protocols [19–21]. But, despite extensive research on speeding up RV, e.g., [6–10, 12, 16], a recent study showed that RV overhead can be still as high as 5,000x, or 27 hours, and that 99.87% of monitored traces during testing are redundant [13].

• **Integration** with main JavaMOP/TraceMOP development branch
• **Per-spec hyperparameter** selection and/or disabling
• **RL-trajectory saving** for the sequence of decision-making steps
• **Offline hyperparameter tuning** using RL-trajectories
• **Performance optimizations** on JavaMOP and TraceMOP
• **Bug fix** in the `ListIterator_Set` spec

Table 1: Summary of Valg's new features.

Motivated by these findings, we recently proposed a new technique to speed up RV during testing by reducing redundant monitoring [18]. That technique formulates *selective* monitor creation as a two-armed bandit reinforcement learning (RL) problem [23]. Each monitor creation point l is associated with an RL agent with two possible actions (i.e., *arms*) $\mathcal{A} = \{\texttt{create}, \texttt{ncreate}\}$. At each time step t, the agent selects action A_t and receives reward R_t. If the selected action is `create`, RV creates a monitor, and the agent receives a positive reward if that monitor observes a unique trace; otherwise, it receives zero. For `ncreate`, no monitor is created, and the reward is estimated using the proportion of duplicates among previously observed traces (see [18] for details). Intuitively, rewards aim to reduce redundant monitored traces and preserve unique ones.

At runtime, the RL agent at each location l learns an optimal *policy* that determines which action to take at each time step t. We use an action-value method, which has two phases. First, the agent estimates the *value* $Q_t(A)$, the expected reward for taking action A at time step t. We use the *exponential recency-weighted average* (EWRA) [23] method for this phase, motivated by our observation that a duplicate (unique) trace tends to follow a duplicate (unique) trace. So, recent rewards have higher weights, determined by learning rate α. In the second phase, the agent selects an action based on the estimated values. The occurrence of duplicate and unique traces is inherently unpredictable. So, we use *epsilon-greedy action selection* strategy [23], which chooses the action with the highest value or explores with a random action (based on a hyperparameter ϵ). We also use a convergence threshold δ (the criterion to stop learning) and initial values for each action ($Q_0(\texttt{create}), Q_0(\texttt{ncreate})$).

Our prototype is up to 20.2x and 555.6x faster than JavaMOP and TraceMOP [14], two state-of-the-art (SoTA) RV tools, respectively [18]. JavaMOP does not store monitored traces, but TraceMOP does. Our prototype also finds 99.6% of violations with the default hyperparameters, and 95.1% of unique traces after tuning.

```
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*    @fail { // print violation } }
```

```
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     ...}
12   return value; }
```

Figure 1: `Appendable_ThreadSafe` spec (left) and example code under test (right).

We present VALG, which improves our prototype with the features summarized in Table 1. These features (i) speed up our prototype or improve its unique-trace preservation (rows 2, 4, and 5); (ii) enhance usability and maintainability (row 1); (iii) enable postmortem analysis of RL trajectories (row 3); and (iv) fix a long-lasting spec bug that caused false-alarm violations [4] (row 6).

We evaluate VALG using 56 projects from [18]. First, we compare our prototype with default hyperparameters against VALG with fixed, non-default per-spec hyperparameters. VALG is up to 1.5x and 6.8x faster than our prototype when applied to JAVAMOP and TRACEMOP, respectively. VALG also checks up to 59pp more unique traces than our prototype. Next, we tune per-spec hyperparameters offline using saved RL trajectories. VALG with tuned hyperparameters is up to 4.2x and 1.9x faster than our prototype when applied to JAVAMOP and TRACEMOP, respectively, and finds up to 94.5pp more unique traces. Lastly, our optimizations of JAVAMOP and TRACEMOP make them up to 4.1x and 316.5x faster, respectively, and find all deterministic traces. We have integrated VALG with the main development branch of JAVAMOP and TRACEMOP at: https://github.com/SoftEngResearch/tracemop. Our video demo is at: https://youtu.be/_QCyHaa_ICc.

2 Example

Figure 1 shows example spec and code. The `Appendable_ThreadSafe` spec checks if an `Appendable` object `appendable` is modified in the different threads, which can cause race conditions. When `append()` is first called on `appendable`, RV creates a monitor and stores it `owner` thread. Then, the monitor checks if subsequent `append()` calls on `appendable` are from `owner`. If so, the `safe_append` event is triggered; otherwise, the `unsafe_append` event is triggered, violating the spec.

The `eval()` method from `expression.parser` [22] evaluates the expression `f_x` w.r.t. the variable `xi`. Line 7 is internally translated into two `append()` method calls on a `StringBuilder` object. Each time line 7 is executed, an `Appendable_ThreadSafe` monitor is created, and two `safe_append` events are signaled, producing trace τ: `[safe_append,safe_append]`. Since `eval()` is called multiple times and contains a loop, RV creates *68,000,157* monitors on line 7. All such monitors observe τ, but only one of them is sufficient for bug finding; the rest are redundant and merely incur overhead.

Our prototype reduces redundant monitors and is faster than the SoTA, but its internals are not observable. VALG's trajectory-saving feature sheds light on the selective monitor creation process. Using this feature, users can observe that VALG still sub-optimally creates 10,000 `Appendable_ThreadSafe` monitors on Line 7 in `eval`.

Our prototype only supports using the same hyperparameter values across all specs, but those values may not be optimal for each spec. VALG addresses this limitation by supporting per-spec hyperparameters. By examining VALG's trajectories, users can better select per-spec hyperparameter values. For example, VALG creates only two monitors on line 7 using hyperparameters $\langle \alpha = 1.0,\ \epsilon = 0.0,\ \delta = 1\text{e}{-4},\ Q_0 = (1.0,\ 0.5) \rangle$ for the `Appendable_ThreadSafe` spec.

3 VALG

We implement VALG atop SoTA RV tools, JAVAMOP and TRACEMOP. Our prototype [18] is in a separate repository that can go out of sync if JAVAMOP and TRACEMOP are updated. So, we integrate VALG with the main branch of JAVAMOP and TRACEMOP. That way, VALG can be enabled and maintained in the same code base (see §4). VALG supports all 160 specs that the JAVAMOP and TRACEMOP support, and it is implemented in only 1.3k lines of Java code. We next describe features that VALG adds to our prototype.

1. Per-spec Hyperparameters. VALG allows users to set hyperparameters per spec, an improvement over our prototype, which used only one set of hyperparameters that may not be equally optimal for all specs. Our evaluation (§5.1) shows that per-spec hyperparameters can make VALG faster and preserve more unique traces than our prototype. VALG has a command-line argument for setting per-spec hyperparameters, e.g., `-spec Iterator_HasNext "{0.5,0.5,0.0001,5.0,5.0}"`; default values are used for specs for which users provide none. Users can also disable RL agents for a spec in future runs if RL overheads outweighed time savings in a previous run. This per-spec feature and the flag expand VALG's hyperparameter space from four to five variables times the spec count.

2. RL trajectory Saving. Unlike our prototype, VALG allows persisting RL agent trajectories for postmortem analysis. A *trajectory* for a spec s at monitor creation point 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:

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

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 estimated action values. Figure 2 shows a partial trajectory for the `eval()` method in Figure 1, using default hyperparameters $\langle \alpha = 0.9,\ \epsilon = 0.1,\ \delta = 1\text{e}{-4},\ Q_0 = (5.0,\ 0.0) \rangle$. At the monitor creation point on line 7, only the first trace is unique. So, the RL agent chooses `create` at the first three steps, but receives a positive reward (1.00) only at $t = 0$. At $t = 3$, the agent explores `ncreate`, receives a continuous reward (0.67), and updates $Q_4(\texttt{n}^*)$ accordingly. The agent then exploits the action values and selects `ncreate` again at $t = 4$, suppressing the creation of a redundant monitor. Trajectories make agents' decision making observable and they can help

$$\mathcal{T}_{s,l} = \begin{bmatrix} \langle 0 : \texttt{create},\ R_0 = 1.00,\ Q_0(\texttt{c}^*) = 5.00,\ Q_0(\texttt{n}^*) = 0.00\rangle, \\ \langle 1 : \texttt{create},\ R_1 = 0.00,\ Q_1(\texttt{c}^*) = 1.40,\ Q_1(\texttt{n}^*) = 0.00\rangle, \\ \langle 2 : \texttt{create},\ R_2 = 0.00,\ Q_2(\texttt{c}^*) = 0.14,\ Q_2(\texttt{n}^*) = 0.00\rangle, \\ \langle 3 : \texttt{ncreate},\ R_3 = 0.67,\ Q_3(\texttt{c}^*) = 0.01,\ Q_3(\texttt{n}^*) = 0.00\rangle, \\ \langle 4 : \texttt{ncreate},\ R_4 = 0.67,\ Q_4(\texttt{c}^*) = 0.01,\ Q_4(\texttt{n}^*) = 0.60\rangle, \\ \dots \end{bmatrix}$$

where s = `Appendable_ThreadSafe`, l = `eval@7`, and $\texttt{c}^*$ and $\texttt{n}^*$ are shorthand for the `create` and `ncreate` actions.

Figure 2: Partial trajectory for the code and spec in Fig 1.

users reason about and select hyperparameter values (§2). Trajectories can have other uses, such as offline hyperparameter tuning (described next) or evaluating custom metrics (future work).

3. Offline Hyperparameter Tuning. A key question in RL is how to automate the search for better hyperparameter values. Our prior work [18] tuned hyperparameters *online*, which was costly (because it involved running RV multiple times) and took days for some projects. VALG provides an automated script that uses trajectories to perform faster *offline* tuning. Since VALG selectively creates monitors, it can only obtain a partial sequence of monitored traces due to steps with the `ncreate` action. So, VALG first infers a more complete sequence using the cumulative decayed weight:

Definition 1 (Cumulative decayed weight)**.** Let $\{(t_i, x_i)\}$ be an observed partial sequence of traces, where $x_i = 1$ if the trace is unique and $x_i = 0$ if it is a duplicate. Let $\lambda > 0$ be the decay rate. The cumulative decayed weight for x at time step t is defined as:

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

The weight can be efficiently computed by incremental updates:

$$w_x(t+1) = \gamma \cdot w_x(t) + \delta_x(t+1), \text{ where } \gamma = e^{-\lambda} \text{ and}$$

$$\delta_x(t+1) = \begin{cases} 1 & \text{if the observed trace at } t+1 \text{ has value } x, \\ 0 & \text{otherwise.} \end{cases}$$

VALG first builds the complete sequence by filling missing steps at time t with the value x_t that has the higher cumulative weight, i.e., $\hat{x}_t = \arg\max_x w_x(t)$. Then, VALG uses Optuna [5] to tune hyperparameters by simulation on the inferred sequence, without needing to re-run RV multiple times as our prototype did. The objective here is to maximize the number of unique traces. Offline tuning outputs a set of hyperparameter values per spec that helps preserve more unique traces. The inferred trace sequence can be imprecise, but our simulation-based optimization reduces the time for hyperparameter tuning from days to minutes.

4. New Optimizations. We implement three optimizations in JAVAMOP and TRACEMOP [4]. First, we remove unnecessary location parameters of event-handler methods and disable unnecessary collection of debug information. Second, we look up event locations more efficiently. Lastly, we reduce synchronization overheads in internal data structures by using per-spec locks instead of the previous global locks. Per-spec locks are faster as they enable parallel processing of events for different specs. These three optimizations speed up JAVAMOP, TRACEMOP, and VALG.

5. Fixed a Spec Bug. Lastly, we identify and fix a bug in the `ListIterator_Set` spec that has been present since at least 2022 [3]. The spec checks if `set` comes only after `next` or `previous`, but incorrectly constrained `remove` calls. We fix to allow `remove` calls and validate our repair by checking that it preserves true bugs and avoids false positives produced by the buggy version.

Table 2: Comparison between baselines and VALG configurations. (#projects with improvements) avg. / max. Bl.: Baseline, `-P`: Prototype, `-a`: VALG$^{\alpha}$, `-e`: VALG$^{\epsilon}$, `-o`: VALG$^{\texttt{off}}$.

	Time (JAVAMOP)	Time (TRACEMOP)	Unique Traces
Bl.	1,024k / 17,929k	3,525k / 86,675k	36,212 / 840,543
`-P`	▼(**49**) 2.1x / 14.8x	▼(**53**) 12.1x / 504.7x	▼(**49**) 74.5% / 98.9%
`-a`	▼(**35**) 1.1x / 1.3x	▼(**31**) 1.3x / 6.8x	▲(**3**) 1.5pp / 2.2pp
`-e`	▼(**24**) 1x / 1.2x	▼(**17**) 1.2x / 2.9x	▲(**26**) 10.8pp / 59pp
`-o`	▼(**17**) 1.1x / 1.5x	▼(**10**) 1.1x / 1.4x	▲(**39**) 20.5pp / 84.3pp

4 Installation and Usage

Two implementation-level goals in VALG are: (i) minimizing effort for users to enable/disable features; and (ii) easily switching to SoTA JAVAMOP and TRACEMOP. To enable VALG features, users simply specify `-valg` when building the Java agents that are used to integrate RV with testing frameworks [2]. VALG also supports the `-spec` flag for per-spec hyperparameters and/or disabling:

```
$ bash install.sh [false:JavaMOP | true:TraceMOP] false
    -valg (-spec ⟨spec-name⟩ ["{⟨α⟩,⟨ϵ⟩,⟨δ⟩,⟨Q₀⟩}" | off])*
```

Afterwards, users can use the VALG-enabled Java agents in the same way as those for JAVAMOP and TRACEMOP [1]. For Maven projects, users simply add the agent to the `pom.xml` file and run `mvn test`. Non-Maven projects can run VALG with the `-javaagent` flag, e.g., `java -javaagent:⟨path-to-agent⟩/valg-agent.jar Main`. Building agents without `-valg` yields JAVAMOP or TRACEMOP.

Users can optionally set `-traj` when building the agent. Using this flag without also setting `-valg` raises an error. After monitoring with VALG, trajectories are stored in a `trajectories` directory in the same location as the `pom.xml` file. Lastly, VALG provides a script to automate offline hyperparameter tuning:

```
$ python param_tune.py ⟨trajectories⟩ ⟨trials⟩ ⟨out-file⟩
```

5 Evaluation

We compare VALG with our prototype, JAVAMOP, and TRACEMOP. We henceforth refer to extensions of JAVAMOP and TRACEMOP with VALG as VALG-J and VALG-T, respectively. We use 56 of 58 projects used to compare our prototype with TRACEMOP in [18]; TRACEMOP takes days on the other two. We run experiments on machines with Intel® Xeon® 36-core 2.2 GHz processors and 125GB RAM (for JAVAMOP, VALG-J), or 56-core 2.6 GHz processors and 503GB RAM (for TRACEMOP, VALG-T).

5.1 Benefits of Per-spec Hyperparameters

To evaluate the degree to which per-spec hyperparameters provide speedups and improve unique trace preservation, we compare our prototype with default hyperparameters against VALG with per-spec hyperparameters. We first identify *effective specs* with at least one trace. Then, we sample from the effective specs and assign fixed, non-default hyperparameter values for the sampled specs: (1) VALG$^{\alpha}$ with $\alpha = 0.5$ and (2) VALG$^{\epsilon}$ with $\epsilon = 0.5$. We also compare with VALG$^{\texttt{off}}$, which disables RL agents for the sampled specs.

Table 2 shows the results. Notably, in the worst case, our prototype and all VALG configurations find 99.8% of all violations (648/649) found by JAVAMOP and TRACEMOP. So, we exclude violation counts from Table 2. Our prototype is faster than JAVAMOP and TRACEMOP, respectively, in 49 and 53 projects by an average (max) of 2.1x

(14.8x) and 12.1x (504.7x). Compared with our prototype, VALG$^{\alpha}$ and VALG$^{\epsilon}$ are respectively faster on average (max) by 1.3x (6.8x) and 1.2x (2.9x) in 31 and 17 projects vs. JAVAMOP and TRACEMOP, respectively. Our prototype misses an average of 25.5% of TRACEMOP's unique traces in 49 projects. But, VALG improves the trace preservation rate in 27 projects with per-spec hyperparameters. For example, VALG$^{\epsilon}$ checks 74.2% of TRACEMOP's unique traces in a project where our prototype checks only 34.1%.

5.2 Benefits of Hyperparameter Tuning

To evaluate offline hyperparameter tuning, we tune hyperparameters (per spec, using the approach described in §3) with 100 trials for all 56 projects. We refer to the tuned VALG as VALG$^{\text{tune}}$. Compared with our prototype, VALG$^{\text{tune}}$ is respectively faster than JAVAMOP and TRACEMOP in 24 projects by an average (max) of 1.2x (4.2x) and 1.1x (1.9x). VALG$^{\text{tune}}$ is faster than all VALG configurations from §5.1 in 9 projects. Also, VALG$^{\text{tune}}$ preserves more unique traces than our prototype in 34 projects with an average of 17.1pp. For example, VALG$^{\text{tune}}$ finds 99.9% traces in a project where our prototype finds 5.49%. In 27 projects, VALG$^{\text{tune}}$ finds more unique traces than all VALG configurations from §5.1.

5.3 Evaluating Our Tool Optimizations

For all 56 projects, we compare JAVAMOP and TRACEMOP with and without our new optimizations. Our optimizations yield speedups in 28 and 49 projects, by an average (max) speedup of 1.1x (4.1x) and 6.8x (316.5x) compared to JAVAMOP and TRACEMOP, respectively. These speedups save up to 24 hours for a project. TRACEMOP benefits more from our optimizations because: (i) some optimizations apply only to TRACEMOP, e.g., reduced debug-information collection; (ii) to store traces, TRACEMOP performs much more location lookups than JAVAMOP; and (3) event processing is slower in TRACEMOP, so parallel processing is more beneficial. Also, we check that original monitoring behavior is preserved. Tests can be non-deterministic (but they always pass in our evaluation), so RV can produce slightly different sets of traces in different runs [14]. We run each project three times, and for the deterministic projects—i.e., those with the same traces across all three runs—we confirm that TRACEMOP produces the same traces post optimization. VALG's effectiveness is preserved even when compared with the optimized baselines; VALG is still faster by an average (max) of 1.3x (15.3x) for JAVAMOP and 1.6x (6.3x) for TRACEMOP.

6 Limitations

Since VALG predicts whether unseen traces will be unique or redundant, VALG can miss violations due to mispredictions. We found that such misses are rare and happen when a unique trace follows a long sequence of redundant ones, misguiding an RL agent [18]. But, VALG still preserves over 99% of violations found by JAVAMOP and TRACEMOP. Some program patterns can make VALG fail to converge, and program non-determinism can affect the performance of VALG. We leave a qualitative analysis of convergence and program non-determinism as future work. Lastly, trajectory saving incurs time and memory overheads. We profiled 20 projects among our evaluation subjects, and the average time overhead was 14.8% (or, 10.9 seconds). Interestingly, the average memory overhead was −0.6% (4.2MB less memory). VALG creates much fewer monitors, which and compensates for the overhead of saving trajectories.

7 Conclusion

VALG is a reinforcement learning (RL) based selective monitoring RV tool for Java. VALG is based on our recent work, which formulated selective monitor creation as a two-armed bandit RL problem. VALG adds several features and optimizations to our original prototype, e.g., per-spec hyperparameters and trajectory saving. We also integrate VALG with the main development branch of JAVAMOP and TRACEMOP and fix a long-lasting spec bug. VALG can be faster and monitor more unique traces than our prototype.

Acknowledgments. We thank Kevin Guan for helping us integrate VALG with JAVAMOP and TRACEMOP, and Pengyue Jiang, Stephen Shen, and the reviewers for valuable comments. We thank Google for Cloud Platform credits. This work is supported by US NSF Grants CCF-2045596, CCF-2319473, CCF-2403035, and CCF-2525243.

References

[1] 2025. Adding TraceMOP's Java Agent to a Maven project. https://github.com/SoftEngResearch/tracemop/blob/master/docs/AddAgent.md
[2] 2025. Building a TraceMOP Java Agent. https://github.com/SoftEngResearch/tracemop/blob/master/docs/BuildAgent.md
[3] 2025. Fixed ListIterator_Set spec. https://github.com/SoftEngResearch/tracemop/commit/c608797a90030d52e49c97e3aefe18813fc7910c
[4] 2025. Integration of Valg into JavaMOP and TraceMOP. https://github.com/SoftEngResearch/tracemop/pull/38
[5] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. In *KDD*.
[6] Pavel Avgustinov, Julian Tibble, and Oege de Moor. 2007. Making trace monitors feasible. In *OOPSLA*.
[7] Eric Bodden, Patrick Lam, and Laurie Hendren. 2008. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-time. In *FSE*.
[8] Feng Chen, Patrick O'Neil Meredith, Dongyun Jin, and Grigore Roşu. 2009. Efficient formalism-independent monitoring of parametric properties. In *ASE*.
[9] Feng Chen and Grigore Roşu. 2009. Parametric trace slicing and monitoring. In *TACAS*.
[10] Matthew B. Dwyer, Rahul Purandare, and Suzette Person. 2010. Runtime Verification in Context: Can Optimizing Error Detection Improve Fault Diagnosis?. In *RV*.
[11] Ulfar Erlingsson and Fred B. Schneider. 2000. IRM enforcement of Java stack inspection. In *IEEE S&P*.
[12] Kevin Guan, Marcelo d'Amorim, and Owolabi Legunsen. 2025. Faster Explicit-Trace Monitoring-Oriented Programming for Runtime Verification of Software Tests. In *OOPSLA*.
[13] Kevin Guan and Owolabi Legunsen. 2024. An In-depth Study of Runtime Verification Overheads during Software Testing. In *ISSTA*.
[14] Kevin Guan and Owolabi Legunsen. 2025. TraceMOP: An Explicit-Trace Runtime Verification Tool for Java. In *FSE Demo*.
[15] Klaus Havelund and Grigore Roşu. 2001. Monitoring programs using rewriting. In *ASE*.
[16] Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Roşu. 2011. Garbage Collection for Monitoring Parametric Properties. In *PLDI*.
[17] Min Kim, Mukund Viswanathan, Houssem Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. 1999. Formally specified monitoring of temporal properties. In *ECRTS*.
[18] Shinhae Kim, Saikat Dutta, and Owolabi Legunsen. 2025. Faster Runtime Verification during Testing via Feedback-Guided Selective Monitoring. In *ASE*.
[19] 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).
[20] 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*.
[21] Breno Miranda, Igor Lima, Owolabi Legunsen, and Marcelo d'Amorim. 2020. Prioritizing Runtime Verification Violations. In *ICST*.
[22] sbesada. 2023. java.math.expression.parser. https://github.com/sbesada/java.math.expression.parser
[23] Richard S. Sutton and Andrew G. Barto. 2018. Reinforcement Learning: An Introduction. (2018).