Faster Runtime Verification during Testing via Feedback-Guided Selective Monitoring

Shinhae Kim, Saikat Dutta, and Owolabi Legunsen (Cornell University)

1. Background and Problem

Runtime Verification (RV)

- Monitors program executions against formal specifications (specs)
- Found **hundreds of bugs** regarding correct JDK API usage [1-3]

Problem: RV has **runtime overhead** which can be as high as 5,000x compared to unit testing only, or 27 hours [4]

- There exist **two types of specs** in our style of RV (MOP)
- Each type of specs incurs runtime overhead for different reasons

1. Parametric Specs

- RV creates a **monitor** for <u>each</u> set of spec-related objects
- **99.87%** monitors are **redundant** for bug finding [4]. E.g.:

Spec: Appendable_TheadSafe

Event: safe_append
- append(..) is called in thread T

Event: unsafe_append
- append(..) is called in thread T' != T

unsafe_append* → violation

```
private double eval(final String f_x, final double xi) ..
  String number;
  for (int i = 0; i < f_x.length(); i++) {
    final char character = f_x.charAt(i);
    if (character >= '0' && character <= '9') {
      hasNumber = true;
      number += character;
number = new StringBuilder( )
.append(number).append(character).toString( )
```

Trace: [safe_append, safe_append]

RV creates **68,000,157 parametric monitors** that check the same trace!

2. Non-Parametric Specs

- RV creates only one **monitor** for <u>all</u> spec-related objects or static calls
- Over **99.99%** of signaled events are **redundant** for bug finding. E.g.:

Spec: Math_ContendedRandom

Event: onethread_use
- Math.random() is called in thread T

Event: otherthread_use
- Math.random() is called in thread T' != T

otherthread_use* → violation

```
public static String generateData(int byteSize) ..
  ..
  StringBuilder b = new StringBuilder(byteSize *2);
  for (int i = 0; i < byteSize; i++) {
    if (Math.random() * 100 > 98) {
      // appends a terminating character to b
```

Trace: [onethread_use, onethread_use, …]

RV signals **260,000,000 non-parametric events** that check the same logic!

[1] Legunsen et al., "How Good are the Specs? A Study of the Bug-Finding Effectiveness of Existing Java API Specifications," ASE 2016

[2] Legunsen et al., "How Effective are Existing Java API Specifications for Finding Bugs during Runtime Verification?," JASE 2019

[3] Miranda et al., "Prioritizing Runtime Verification Violations," ICST 2020

[4] Guan and Legunsen, "An In-Depth Study of Runtime Verification Overheads During Software Testing," ISSTA 2024

2. Our Approach: Valg

Key Idea: <u>Selective</u> Monitoring!

- Use **reinforcement learning** for selective parametric monitor creation
- Use violation feedback to selectively signal non-parametric events

Selective Parametric Monitor Creation

Learning objective: Reduce **redundant** monitors and preserve **unique** ones

■: "Hyperparameters"

1. Formulation as a **two-armed bandit** problem
   - **Actions**: {'create', 'ncreate'} // creating a monitor or not
   - Binary **reward** for 'create' and continuous **reward** for 'ncreate'

$$R_{\text{create},t} \doteq 0 \text{ if } (\text{trace}_t \text{ is redundant}) \text{ else } 1 \quad R_{\text{ncreate},t} \doteq \frac{\sum_{k=0}^{t-1} 1(\text{trace}_k \text{ is redundant})}{\sum_{k=0}^{t-1} 1(\text{trace}_k \text{ is observed})}$$

2. Selection based on **action-value method** [Learning Rate] [Exploration Prob.]
   - Estimate rewards using **exponential-recency weighted average**

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

   - Enable stochastic exploration (vs. exploitation) using **ϵ-greedy**

$$A_t \leftarrow \begin{cases} \arg\max_a Q_t(a) & \text{with probability } 1-\epsilon \\ \text{random action } a & \text{with probability } \epsilon \end{cases}$$

3. **Convergence logic** for the learning [Convergence Threshold]
   - Heuristic: If the absolute difference in estimated values is close to 1
4. **Initial value** selection [Initial Values]
   - Optimistic value for 'create' to encourage monitor creation at early stages

State-of-the-Art (SoTA)		Our Technique (Valg)
@ iteration 1 **Trace**: [safe_append, safe_append]	[create]	@ iteration 1 **Trace**: [safe_append, safe_append]
@ iteration 2 **Trace**: [safe_append, safe_append]	[create]	@ iteration 2 **Trace**: [safe_append, safe_append]
@ iteration 3 **Trace**: [safe_append, safe_append]	[ncreate]	@ iteration 3 **Trace**: [safe_append, safe_append]
…		…
@ iteration 68,000,157 **Trace**: [safe_append, safe_append]	[ncreate]	@ iteration 68,000,157 **Trace**: [safe_append, safe_append]

Valg reduces **68,000,157 created monitors** to **2 monitors**

Selective Non-Parametric Event Signaling

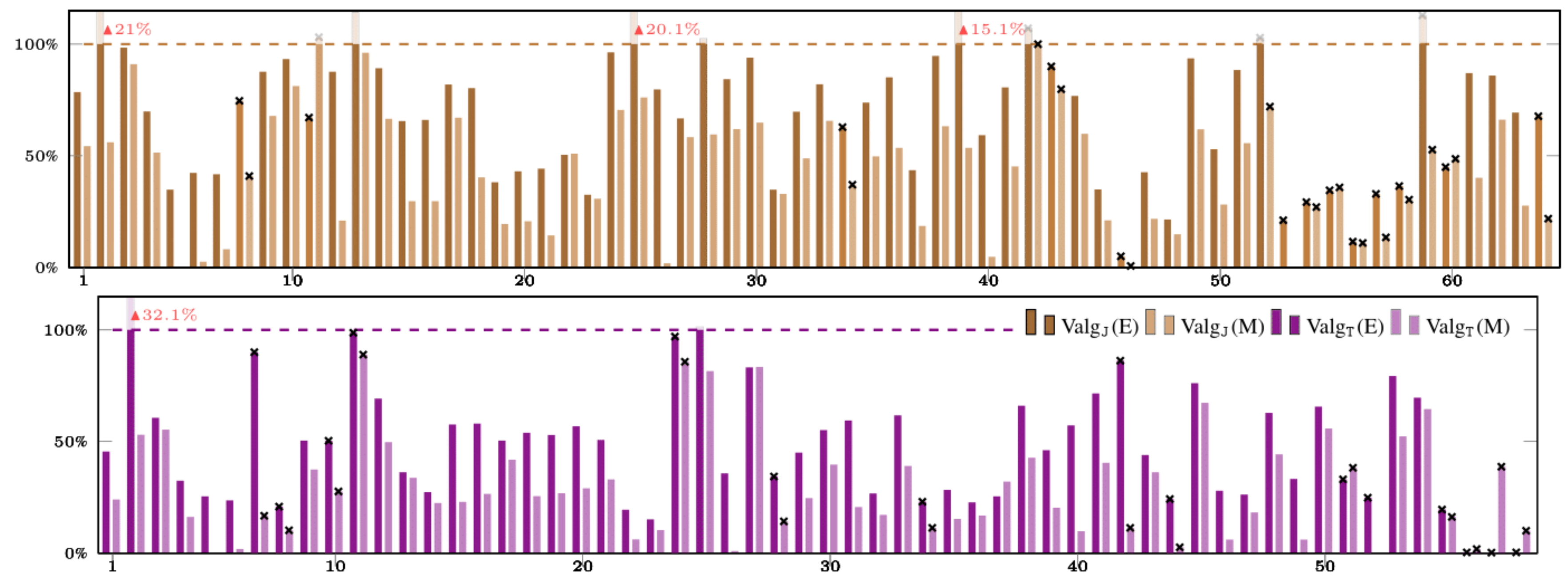
- Valg tracks the violation status of each event location
- Valg does not signal the event if a violation was already detected

Valg reduces **260,000,000 signaled events** to **1 event**

3. Evaluation Results

RQ1 Valg vs. SoTA Techniques (JavaMOP and TraceMOP [5])

- **Setup**: 64 Java open-source projects, 160 JDK API specs



Compared to JavaMOP and TraceMOP,

Overhead. Valg is up to **20.2x (4.3 hrs)** and **551.5x (24.3 hrs)** faster
Valg reduces **3.02 days** down to **11.6 minutes** for three projects

Violations. Valg preserves **99.6%** of the original violations

[5] Guan and Legunsen, "TraceMOP: An Explicit-Trace Runtime Verification Tool for Java," FSE Demo, 2025

RQ2 Valg vs. {10%, 50%} Random Sampling

- **Setup**: 20 Java open-source projects, 160 JDK API specs

Compared to {10%, 50%} random sampling,

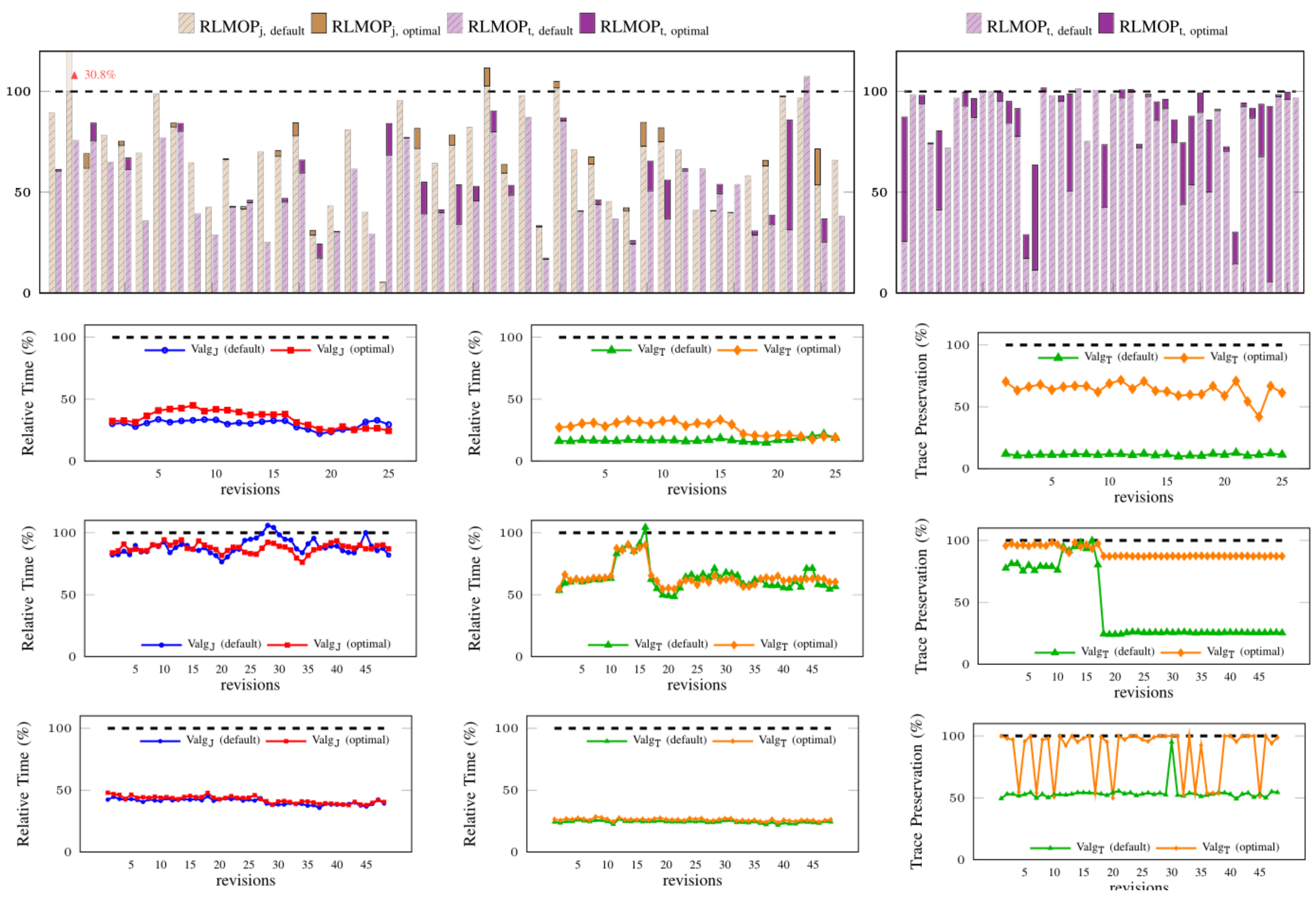
Overhead. Valg is up to **11.8x (4.3 mins)** and **20.1x (22.0 mins)** faster

Violations. Valg preserves **66.1%** and **14.4%** more violations

RQ3 What is the impact of hyperparameter tuning?

Unique Traces. Valg's preservation ratio improves from **76.7%** to **95.1%**

RQ4 How effective and efficient is Valg as code evolves?



4. Discussion and Future Work

Discussion: Comparison with evolution-aware RV, ablation study, memory overhead

Future Work: Valg opens up a new research direction for **learning-based RV**

- Different algorithms, further study on hyperparameter tuning, hyperparameter learning