TraceMOP: An Explicit-Trace Runtime Verification Tool for Java

Kevin Guan
Cornell University
Ithaca, NY, USA
kzg5@cornell.edu

Owolabi Legunsen
Cornell University
Ithaca, NY, USA
legunsen@cornell.edu

Abstract

We present TraceMOP, an explicit-trace runtime verification (RV) tool. RV monitors if execution traces—sequences of events, e.g., method calls—violate formal specifications. TraceMOP monitors each event as it occurs, but unlike prior RV tools, it also tracks monitored traces explicitly. So, TraceMOP can help researchers address several challenges of using RV in testing, e.g., (i) RV overhead is mostly wasted: 99.87% of monitored traces are duplicates of the other 0.13%; and (ii) RV violations are often flaky. We describe TraceMOP's design, implementation and use via a Maven plugin and GitHub Actions. We evaluate TraceMOP on 105 open-source projects. TraceMOP preserves the monitoring functions of a mature RV tool; it is up to 11x faster and uses up to 231.7GB less memory than our previous prototype. We use TraceMOP to debug four flaky violations; it is open sourced (https://github.com/SoftEngResearch/tracemop) and a demo is at https://tracemop.gkevin.com/demo.

ACM Reference Format:
Kevin Guan and Owolabi Legunsen. 2025. TraceMOP: An Explicit-Trace Runtime Verification Tool for Java. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728613

1 Introduction

Runtime Verification (RV) [1–3] monitors if program executions violate formal specifications (specs). An RV tool takes a program and its tests, and specs, then it instruments the program to signal relevant events to monitors, usually automata, at runtime. Monitors check if traces—sequences of events, e.g., method calls—violate specs. An RV tool outputs violations if a spec is not satisfied.

In theory, RV checks traces. But, prior RV tools use event-by-event algorithms that do not explicitly track traces. One reason is that these RV tools were designed for efficient deployment-time RV usage for improving system reliability. Such RV tools are now being adopted, e.g., at Grammatech [4] and in the Linux kernel [5].

There is a growing line of recent research on making RV more usable for finding more bugs during regression testing, before deployment. Such research include those that aim to (i) find many confirmed bugs by using RV to monitor *passing* tests in hundreds of open-source projects against specs of correct API usage [6–11]; and (ii) speed up RV by incrementally running existing RV tools only on code affected by changes during software evolution [12–15].

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728613

An explicit-trace RV tool that outputs traces can help address challenges to broader RV adoption during testing. For example, we found [16] that (i) RV's high overhead is mostly wasted—99.87% of traces cannot reveal new bugs: they are duplicates of the other 0.13%; and (ii) RV violations are often flaky, i.e., they non-deterministically appear in multiple RV runs on the same program (even when tests always pass). Also, manually inspecting each (of hundreds) spec violation takes a person hour [6], partly because JavaMOP [17]—the state-of-the-art RV tool used—does not output violating traces.

Traces are needed to speed up RV by leveraging this finding about wastefulness, debug flaky violations, and reduce manual inspection effort. Yet, the explicit-trace prototype from our study [16] is not easy to use; it is also slow and requires a lot of memory and hard-disk space. An engineering challenge that we address is how to efficiently store and retrieve the millions of traces (involving billions of events) that RV generates in open-source projects [6].

We present the design, implementation, use, and evaluation of TraceMOP, an explicit-trace RV tool for Java. We build TraceMOP on top of JavaMOP [17], a decades-old, widely-cited, mature, and well-engineered RV tool that can scalably monitor multiple specs simultaneously in one execution. To do so, we first modernize and extensively refactor several parts of JavaMOP. So, TraceMOP has all JavaMOP's features, but it enhances JavaMOP in five ways.

1. JavaMOP is not an explicit-trace RV tool, but TraceMOP is.
2. JavaMOP supports defining specs using Java syntax up to Java 8, but TraceMOP supports up to Java 17.
3. JavaMOP code is in two repositories—a frontend [17] and the RV-Monitor backend [18]. TraceMOP unifies these repositories into a multi-module Maven project and reduces duplicate code.
4. TraceMOP fixes a serious performance regression that has been in JavaMOP since 2013 [19].
5. JavaMOP in-lines two-decade old JavaParser [20] code for processing specs. But, TraceMOP uses the latest JavaParser library.

Table 1 shows seven new TraceMOP features that enhance our prototype [16]. We implement two optimizations in TraceMOP that improve the speed, memory usage, and disk usage of our prototype. TraceMOP supports multi-module Maven projects (MMMP) [21]; our prototype supports only single-module projects. TraceMOP tracks traces if tests spawn multiple JVMs and it natively supports comparing two traces to aid debugging; our prototype cannot. Lastly, we add a Docker image (eases installation), a plugin (integrates with Maven projects), and a GitHub Action (allows "live" usage in CI pipelines). These three ease-of-use features can be seen as enhancements over JavaMOP when trace collection is turned off.

We perform a three-part evaluation of TraceMOP. To do so, we use 105 open-source projects from our study [16] and 160 specs of correct Java API usage that are widely used in the RV literature [22].

Table 1: New TRACEMOP features enhancing our prototype; the last 3 rows also enhance JavaMOP when trace tracking is off.

Feature	Description
More efficient trace tracking	TRACEMOP tracks traces more efficiently (in time and space), using new data structures that we implement.
MMMP support	TRACEMOP supports single- and Multi-Module Maven Projects (MMMP); prototype supports only single.
Multi-JVMs trace tracking	TRACEMOP supports collecting traces even if a project spawns multiple JVMs (e.g., one per test).
Trace Comparison	TRACEMOP allows comparing RV traces across multiple runs, e.g., for debugging flaky violations.
Docker support	TRACEMOP ships with a Docker image where its required environment is pre-installed.
Maven Plugin	TRACEMOP comes with a Maven plugin that allows seamless integration with a project's build system.
GitHub Action	TRACEMOP provides a custom GitHub Action that allows users to more easily integrate RV in projects' CI pipelines.

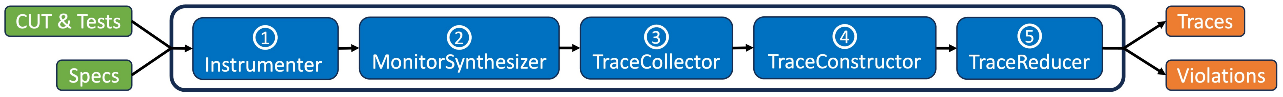

Figure 1: TRACEMOP's architecture and its main components.

```
1 CSC (Collection c, Iterator i){
2   Collection c;
3   event sync after() returning(Collection c) :
4     call(* Collections.synchronizedCollection(Collection)){ this.c = c; }
5   event syncMakeIter after(Collection c) returning(Iterator i) :
6     call(* Collection+.iterator()) && target(c) && if(Thread.holdsLock(c)) {}
7   event asyncMakeIter after(Collection c) returning(Iterator i) :
8     call(* Collection+.iterator()) && target(c) && if(!Thread.holdsLock(c)) {}
9   event useIter before(Iterator i) :
10    call(* Iterator.*(..)) && target(i) && if(!Thread.holdsLock(this.c)) {}
11  ere : (sync asyncMakeIter) | (sync syncMakeIter useIter)
12  @match {/*print violation*/} }
```

Figure 2: CSC Spec, written in TRACEMOP's AspectJ-based DSL.

```
1 public int sum(List<Integer> list) {
2   Collection<Integer> c = Collections.synchronizedCollection(list);
3   /* Generate event: Collections_SynchronizedCollection.sync */
4   int total = 0;
5   Iterator<Integer> iterator = c.iterator();
6   /* Generate event: Collections_SynchronizedCollection.asyncMakeIter */
7   while (iterator.hasNext()) total += iterator.next();
8   /* Generate event: Collections_SynchronizedCollection.useIter */
9   return total;
10 }
11 @Test public void testSum(){ assertEquals(6, sum(Arrays.asList(1, 2, 3))); }
```

Figure 3: An example (toy) monitored code and its unit test.

First, we compare TRACEMOP's violations. Excluding flaky violations, TRACEMOP finds all violations that JavaMOP finds. So, TRACEMOP preserves JavaMOP's monitoring functionality.

Second, we compare TRACEMOP's time and space overheads with those of our prototype [16]. (We do not know any other explicit-trace RV tool for simultaneously and scalably monitoring multiple specs.) TRACEMOP is up to 11x (mean: 1.2x) faster than our prototype, which crashes on 13 (of 105) projects. TRACEMOP uses up to 231.7GB (mean: 9.7GB) less memory. Lastly, TRACEMOP's compact representation saves up to 50.5GB (mean: 1.6GB) in disk space.

Finally, we perform a proof-of-concept study on debugging flaky violations by comparing traces from two TRACEMOP runs: one found a violation and another did not, for a program with no failing test. (JavaMOP and our prototype cannot compare traces.) TRACEMOP helped us find root causes of four flaky violations. So, TRACEMOP can aid future work on flaky violations (or debugging non-flaky ones). TRACEMOP and all our experimental scripts and data are open sourced: https://github.com/SoftEngResearch/tracemop.

2 Example

Fig 2 shows an example spec, CSC, that we use RV to check; it previously helped find several confirmed bugs [6, 9]. CSC was formalized by other researchers [22] to check this safety property: code that obtains a `Collections.synchronizedCollection()` must synchronize on that collection before iterating over its contents. Failure to do so can cause code to behave non-deterministically [23].

Lines 3–10 define four relevant events. The `sync` event (lines 3–4) is signaled *after* calling `Collections.synchronizedCollection()` to obtain collection `c`. Event `syncMakeIter` (lines 5–6) is signaled *after* `c.iterator()` is called from code that synchronizes on `c`, and event `asyncMakeIter` (lines 7–8) is signaled *after* such a call is made from code that does not synchronize on `c`. Lastly, event `useIter` (lines 9–10) is signaled *before* calling any method on `c.iterator()` in code that is *not* synchronized on `c`. Line 11 states CSC's safety property, formalized as an extended regular expression. Finally, line 12 is a handler that prints a violation when monitors observe traces that match the property.

Fig 3 shows toy example code that sums a list of integers and a test (line 11) that almost always misses a subtle bug in `sum`: `iterator` is used without synchronizing on `c` (lines 5 and 7). Using RV to monitor the test reveals the bug: all CSC monitors observe the violating trace, `sync asyncMakeIter`. TRACEMOP records all traces observed by monitors, whether they violate the spec or not.

3 Implementation

TRACEMOP's inputs and outputs are those listed in §1, plus monitored traces as an extra output (in addition to violations).

Architecture. Figure 1 shows TRACEMOP's high-level architecture, consisting of five steps. In step ①, `Instrumenter` uses AspectJ to instrument the code under test (CUT) and the tests based on the specs, such that relevant events are signaled to monitors at runtime. Unlike in JavaMOP, `Instrumenter` also collects the name of the current-running test, so users can re-run a test to obtain a trace or debug a violation (without re-running all tests). In step ②, monitors are dynamically synthesized to check traces. Unlike in JavaMOP, monitors notify `TraceCollector` (step ③) of each observed event.

After the program terminates but before the JVM shuts down, `TraceConstructor` (④) processes `TraceCollector`'s data structures (presented shortly) to collect monitored traces. Persisting all monitored traces to disk takes a lot of space, so `TraceReducer` (step ⑤) uses two disk-space saving optimizations: it stores only unique traces along with their frequency, and it compacts individual traces using a strategy that we discuss next.

Optimizations and Data Structures. To improve space efficiency, TRACEMOP stores only the unique traces observed across all monitors. Suppose `sum()` in Fig 3 is called a million times, each time on a three-element list. Every time line 2 is called, RV will create a new monitor. Since `list.size()` is unchanged in those million iterations, all related CSC monitors will observe identical traces. But, TRACEMOP only stores one trace and maps all other monitors

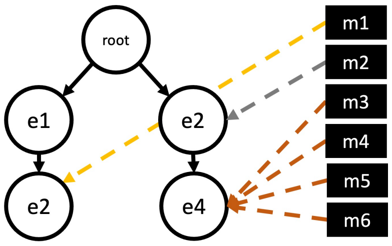

Figure 4: Data structure for storing formulaic specs' traces.

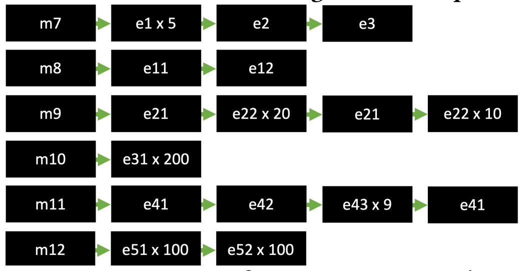

Figure 5: Data structure for storing raw specs' traces.

to that unique trace, effectively reducing the number of recordable traces and events from millions (which is typical) to just a handful. This design is justified by our previous finding [16]: only 0.13% of traces monitored during testing are unique.

Internally, TRACEMOP uses two data structures to optimize trace collection, one per category of specs. CSC is in the first category of *formulaic* specs: the formula that expresses its safety property drives its monitor synthesis algorithm and one monitor will be created for each set of related objects that are its parameter types at runtime. The second category are *raw* specs that users program themselves, without using any logical formalism to express a safety property. Only one monitor is ever created for a raw spec.

Fig 4 shows a data structure used by the `TraceCollector` to store traces for formulaic specs; it is inspired by prefix trees. This example (not related to `sum()`), shows six monitors, `m1` to `m6`, and three unique traces. Monitor `m1` so far observed trace `e1 e2`, monitor `m2` observed the trace `e2`, and monitors `m3` to `m6` observed the trace `e2 e4`. Monitors `m3` to `m6` observed the same trace, so they point to the same node. When a monitor observes a new event, `TracesCollector` first searches the children of the node that the monitor currently points to. If the received event is not a child of the node that the monitor currently points to, `TraceCollector` creates a child node and maps the monitor to that new node. Otherwise, `TraceCollector` maps the monitor to the existing child node that represents the received event.

For raw specs, where there is only one trace, the prefix-tree like structure is inefficient. So, `TraceCollector` uses a different data structure, shown in Fig 5. There, monitors maintain their own list of events, and each element in the list includes information on frequency of consecutive occurrence. For example, monitor `m7` has the trace `e1 e1 e1 e1 e1 e2 e3`, but `TraceCollector` only stores the events `e1`, `e2`, and `e3`, along with their frequencies.

4 Usage

Setting up. A TRACEMOP environment can be set up in three ways. First, if Docker support is available, then users can simply set up the environment like so:

```
~/tracemop$ docker build -f scripts/Dockerfile . -t tracemop
~/tracemop$ docker run -it tracemop /bin/bash
```

Second, users can simply use our pre-built Docker image like so:

```
~/tracemop$ docker pull softengresearch/tracemop
~/tracemop$ docker run -it softengresearch/tracemop
```

Finally, if Docker support is not available, users can manually run the commands in our Dockerfile [24]. Any of these three approaches should set up the environment needed to use TRACEMOP (with, e.g., Java and Maven installed).

Using TRACEMOP as a Maven plugin. TRACEMOP's Maven plugin makes it easier to integrate with Maven-based projects. To use the plugin, first run the Maven install command from TRACEMOP's directory and then run the plugin from a target project's directory:

```
~/tracemop$ mvn install
~/tracemop$ cd ~/project
~/project$ mvn edu.cornell:tracemop-maven-plugin:1.0:run
```

Alternately, after running the above Maven install command and changing to the project's directory, users can add TRACEMOP's Maven plugin to the project's config (i.e., `pom.xml`) file by following our instructions in [25], and then running this shorter command:

```
~/project$ mvn tracemop-maven-plugin:run
```

With either TRACEMOP Maven plugin command, users can optionally specify (i) whether to collect traces (default: `true`) and (2) the output directory to store traces (default: `target/tracemop`). The first of these next two commands stores traces in a non-default location; the second runs TRACEMOP *without* trace tracking:

```
~/project$ mvn tracemop-maven-plugin:run -DoutputDirectory=another-dir
~/project$ mvn tracemop-maven-plugin:run -DcollectTraces=false
```

Using TRACEMOP with GitHub Actions. If a Maven project has already set up CI pipelines using GitHub Actions, all that a user needs to do to enable TRACEMOP is add these lines to their GitHub Actions workflow file (no need to modify `pom.xml`):

```
- uses: SoftEngResearch/tracemop@master
  with:
    collect-traces: true
    output-directory: traces-output-directory
```

Both parameters are optional. By default in GitHub Actions pipelines, TRACEMOP collects traces, saves them to a `traces` directory, and uploads that directory to GitHub. Our single-line change [26] integrates TRACEMOP with GitHub Actions on our fork of an open-source project, and (within three months of this paper's submission) readers can download the traces [27] or see the violations (after clicking "Post Run SoftEngResearch/tracemop@master") [28]. Our README has screenshots of these pages.

Using TRACEMOP as a Java agent (for non-Maven projects). TRACEMOP's Java agent allows to integrate it with arbitrary Java programs, with or without Maven. Versioned TRACEMOP Java agents can be obtained from its release page [29]. Users can also build the agent using our instructions [30]. One can use TRACEMOP's agent to monitor a Java program whose `main` method is in `Main.java` like so (after compiling `Main.java`):

```
java -javaagent:${PATH-TO-AGENT}/tracemop-agent.jar Main
```

TRACEMOP's Java agent can also be used with build systems, e.g., one can follow our instructions [31] to add TRACEMOP to a Maven project's `pom.xml` file, and then run "`mvn test`" to monitor tests.

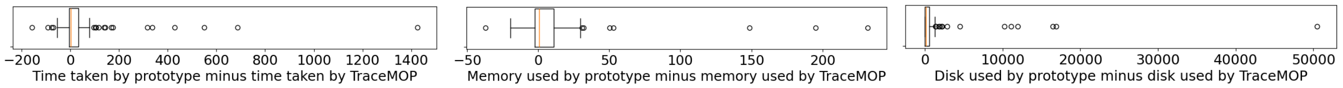

Figure 6: Left: distribution of prototype minus TraceMOP times (seconds). Middle and right: distributions of prototype minus TraceMOP memory (GB) and disk (MB) usage. Positive values mean TraceMOP uses less.

Table 2: Summary statistics on 105 projects that we evaluate: no. of test methods (#Tests), test time w/o RV in seconds (t), lines of code (SLOC), % statement coverage (cov^s), % branch coverage (cov^b), no. of GitHub commits (#SHAs), years since first commit (age), and no. of stars (#★).

	#Tests	t	SLOC	cov^s	cov^b	#SHAs	age	#★
Mean	175.3	14.2	15,867.0	58.4	50.7	513.8	11.3	199
Med	43	4.5	5,191	61.1	53.1	176	11	48
Min	1	1.5	193	0.1	0.0	3	3	1
Max	2,708	159.2	4.3×10^5	96.5	100.0	4,860	27	2,448
Sum	18,407	1,489.2	1.7×10^6	n/a	n/a	n/a	n/a	20,895

5 Evaluation

Experimental Setup. Table 2 summarizes our 105 evaluation subjects. "n/a" is a meaningless sum. Min cov^b is 0%: `Kuangcp/NettyBook2` does not test a branch. We randomly select these projects from our corpus of 1,544 projects [16]; (i) our prototype crashes on 13 due to running out of memory, being MMMP, or spawning multiple JVMs; (ii) RV generates over one million events for 87; and (iii) RV generates fewer than 10,000 events for five. (These characteristics do not correlate with RV overhead [16].) We run experiments on an Intel® Xeon® Gold 6348 machine with 512GB of RAM and 112 cores, Ubuntu 20.04.6 LTS, Java 8, and Maven 3.8.8.

5.1 Comparison with JavaMOP

We validate that TraceMOP preserves the JavaMOP's monitoring functionality by comparing unique violations found by JavaMOP, TraceMOP, and our prototype for all 86 formulaic specs. We exclude raw specs as many of them had flaky violations, making comparison hard. All three tools find the same set of 1,379 unique violations in all 105 projects. So, we have initial confidence TraceMOP's implementation, relative to JavaMOP, whose monitoring functionality it preserves. On average, TraceMOP is 2.4x (max: 13.1x) slower than JavaMOP, as expected: TraceMOP does more work to track traces. Without trace tracking, (i) TraceMOP is up to 8.0x (mean: 1.2x) faster than JavaMOP due to the performance regression in JavaMOP that TraceMOP fixes; and (ii) TraceMOP's location-tracking makes it sometimes slower than JavaMOP.

5.2 Comparison with prototype

Figure 6 compares TraceMOP's time, maximum memory usage, and disk usage on only 92 projects where our prototype [16] runs.

Time Comparison. TraceMOP is often faster than our prototype. The max relative speedup is 999.9% (equivalently, 11x). The average (median) is 19.1% (1.3%). The max absolute speedup saves 23.8 minutes (Figure 6, left); the average (median) is 51.0 (1.1) seconds. Excluding outliers, roughly 25% of projects see no change in time, 25% see a slowdown, and around 50% see a speed-up. With outliers TraceMOP is faster on 53.3% of these projects. Note that users can turn off TraceMOP's optimizations to get the effect of using the prototype when TraceMOP is slower.

Maximum Memory Usage. TraceMOP often uses less memory than our prototype. The max relative reduction is 377% (equivalently, 4.8x). The average (median) is 14.5% (4.5%). The max absolute reduction is 231.7GB (Figure 6, middle); the average (median) is 9.7 (0.7) GB. Excluding outliers, roughly 25% of projects see no change in memory usage, 25% see an increase, and around 50% see a reduction in memory usage.

Disk Usage. TraceMOP *always* uses less disk space than our prototype. The max relative reduction is 6.7×10^5% (equivalently, 6,707.7x). The average (median) is 19,959.4% (587.5%). The max absolute reduction saves 50.5GB (Figure 6, right); the average is 1.6GB.

Overall, TraceMOP is faster in 49 (53.3%) and uses less memory in 53 (57.6%) of these projects; it always uses less disk space. Users can turn off TraceMOP's optimizations to get the effect of using the prototype when TraceMOP is slower or more memory intensive.

5.3 Debugging flaky violations

To begin demonstrating the usefulness of TraceMOP for addressing challenges of using RV during testing, we use it in a small proof-of-concept study to debug four flaky violations. Here, we describe one of those flaky violations and how TraceMOP helped us. Our artifacts contain descriptions of the others [32]. Project `contentful/contentful.java` violates a safety property (not CSC): a collection from which an iterator is obtained must not be modified while that iterator is in use. Such violation can escape Java's ConcurrentModificationException if the iterator is being used in a different thread than the one in which the collection is modified [33, 34].

We obtain a trace like this from the run where the violation appeared—`create.1 useiter.1 modify.3 useiter.2 useiter.1`. In this trace, location 1 creates an iterator from a collection and also uses that iterator at the same location. Then, at location 3, the collection is modified. Next, execution proceeds back to location 2 and then 1. Comparing this trace with a non-violating one, we find why the violation is flaky. The collection is modified during iteration at location 3 in some but not all thread interleaving. Future work could build on TraceMOP to better deal with flaky violations.

6 CONCLUSION

TraceMOP is an explicit trace RV tool for Java that can support research on addressing several challenges of using RV for testing. TraceMOP enhances JavaMOP and our earlier prototype. TraceMOP finds the same set of non-flaky violations as JavaMOP. Compared to our prototype, TraceMOP is often faster, uses less memory and disk space, and helped us understand flaky violations.

Acknowledgments. We thank the anonymous reviewers for their feedback. This work is partially supported by an Intel Rising Star Faculty Award, a Google Cyber NYC Institutional Research Award, and the United States National Science Foundation (NSF) under Grant Nos. CCF-2045596, CCF-2319473, and CCF-2403035.

References

[1] K. Havelund and G. Roşu, "Monitoring programs using rewriting," in *ASE*, 2001.
[2] U. Erlingsson and F. B. Schneider, "IRM enforcement of Java stack inspection," in *IEEE S&P*, 2000.
[3] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky, "Formally specified monitoring of temporal properties," in *ECRTS*, 1999.
[4] "ARTCAT: Autonomic Response To Cyber-Attack," https://grammatech.github.io/prj/artcat.
[5] D. B. de Oliveira, "Efficient runtime verification for the Linux kernel," https://research.redhat.com/blog/article/efficient-runtime-verification-for-the-linux-kernel.
[6] 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.
[7] B. Miranda, I. Lima, O. Legunsen, and M. d'Amorim, "Prioritizing runtime verification violations," in *ICST*, 2020.
[8] O. Javed and W. Binder, "Large-scale evaluation of the efficiency of runtime-verification tools in the wild," in *APSEC*, 2018.
[9] 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.
[10] F. Burk, "A dynamic analysis-based linter for python," Master's thesis, University of Stuttgart, Germany, 2023.
[11] A. Eghbali, F. Burk, and M. Pradel, "DyLin: A Dynamic Linter for Python," in *FSE*, 2025.
[12] O. Legunsen, Y. Zhang, M. Hadzi-Tanovic, G. Roşu, and D. Marinov, "Techniques for evolution-aware runtime verification," in *ICST*, 2019.
[13] O. Legunsen, D. Marinov, and G. Roşu, "Evolution-aware monitoring-oriented programming," in *ICSE NIER*, 2015.
[14] A. Yorihiro, P. Jiang, V. Marques, B. Carleton, and O. Legunsen, "eMOP: A Maven plugin for evolution-aware runtime verification," in *RV*, 2023.
[15] K. Guan and O. Legunsen, "Instrumentation-driven evolution-aware runtime verification," in *ICSE*, 2024.
[16] ——, "An in-depth study of runtime verification overheads during software testing," in *ISSTA*, 2024.
[17] "JavaMOP," https://github.com/runtimeverification/javamop.
[18] "RV-Monitor," https://github.com/runtimeverification/rv-monitor.
[19] "Performance regression that we find in JavaMOP," https://github.com/runtimeverification/rv-monitor/commit/884f9622f.
[20] "JavaParser - Home," https://javaparser.org.
[21] "Guide to Working with Multiple Modules – Maven," https://maven.apache.org/guides/mini/guide-multiple-modules.html.
[22] C. Lee, D. Jin, P. O. Meredith, and G. Roşu, "Towards categorizing and formalizing the JDK API," Computer Science Dept., UIUC, Tech. Rep., 2012.
[23] "Collections_SynchronizedCollection," https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collections.html#synchronizedCollection(java.util.Collection).
[24] "Dockerfile for setting up TraceMOP," https://github.com/SoftEngResearch/tracemop/tree/master/docs/OutsideDocker.md.
[25] "Adding TraceMOP Maven Plugin To `pom.xml` files," https://github.com/SoftEngResearch/tracemop/tree/master/docs/AddPluginToPom.md.
[26] "Adding TraceMOP to our Functional Utils fork," https://github.com/guan-kevin/functional-utils/blob/d6cdc789b00b941bd7a9cf962a9947d5d0336085/.github/workflows/package.yml#L27.
[27] "A sample GitHub Actions run with TraceMOP in our Functional Utils fork," https://github.com/guan-kevin/functional-utils/actions/runs/12739129442.
[28] "A sample GitHub Actions run with TraceMOP in our Functional Utils fork," https://github.com/guan-kevin/functional-utils/blob/d6cdc789b00b941bd7a9cf962a9947d5d0336085/.github/workflows/package.yml#L27.
[29] "TraceMOP GitHub releases," https://github.com/SoftEngResearch/tracemop/releases.
[30] "Building a TraceMOP Java Agent," https://github.com/SoftEngResearch/tracemop/tree/master/docs/BuildAgent.md.
[31] "Adding TraceMOP's Java Agent to a Maven project," https://github.com/SoftEngResearch/tracemop/tree/master/docs/AddAgent.md.
[32] "Debugging Flaky Violations," https://github.com/SoftEngResearch/tracemop/blob/master/docs/FlakyViolations.md.
[33] F. Chen and G. Roşu, "MOP: an efficient and generic runtime verification framework," in *OOPSLA*, 2007.
[34] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu, "Garbage collection for monitoring parametric properties," in *PLDI*, 2011.