An In-Depth Study of Runtime Verification Overheads during Software Testing

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

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

Abstract

Runtime verification (RV) monitors program executions against formal specifications (specs). Researchers showed that RV during software testing amplifies the bug-finding ability of tests, and found hundreds of new bugs by using RV to monitor passing tests in open-source projects. But, RV's runtime overhead is widely seen as a hindrance to its broad adoption, especially during continuous integration. Yet, there is no in-depth study of the prevalence, usefulness for bug finding, and components of these overheads during testing, so that researchers can better understand how to speed up RV.

We study RV overhead during testing, monitoring developer-written unit tests in 1,544 open-source projects against 160 specs of correct JDK API usage. We make four main findings. (1) RV overhead is below 12.48 seconds, which others considered acceptable, in 40.9% of projects, but up to 5,002.9x (or, 28.7 hours) in the other projects. (2) 99.87% of monitors that RV generates to dynamically check program traces are wasted; they can only find bugs that the other 0.13% find. (3) Contrary to conventional wisdom, RV overhead in most projects is dominated by instrumentation, not monitoring. (4) 36.74% of monitoring time is spent in test code or libraries.

As evidence that our study provides a new basis that future work can exploit, we perform two more experiments. First, we show that offline instrumentation (when possible) greatly reduces RV runtime overhead for single versions of many projects. Second, we show that simply amortizing high instrumentation costs across multiple program versions can outperform, by up to 4.53x, a recent evolution-aware RV technique that uses complex program analysis.

CCS Concepts

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

Keywords

Runtime Verification, software testing, software evolution

ACM Reference Format:
Kevin Guan and Owolabi Legunsen. 2024. An In-Depth Study of Runtime Verification Overheads during Software Testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680400

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680400

1 Introduction

Runtime verification (RV) is a lightweight formal method that can help find bugs early, during development, by monitoring program executions against formally specified safety properties. An RV tool takes the code under test, means of executing the code (e.g., unit tests), and formal specifications (specs, for short), and raises violations if an execution does not satisfy a spec.

An RV tool first instruments the code, tests, and third-party libraries, so that relevant program events (e.g., method calls, field accesses, lock acquisitions) are signaled to the tool at runtime. Events trigger (1) generation of monitors (usually automata, like finite-state machines) based on the spec; or (2) search for previously generated monitors that should process new events. A handler is called to deal with violations, e.g., by performing error-recovery.

RV research traditionally focused on monitoring deployed programs. The alluring idea is that, if a handler can always soundly recover from impending violations, then software will always be correct with respect to monitored specs. This powerful idea has motivated decades of research, which is now being realized in practice, such as in GrammaTech's ARTCAT [55].

Recently, researchers showed that the bug-finding ability of test suites can be amplified by using RV specs as additional oracles [84, 86, 101]. The reason is that, when run alone, unit tests typically only assert that outputs computed from given inputs are as expected. But, RV of tests finds more bugs by also checking if traces—sequences of events—satisfy formal specs. So, a test whose assertions pass, but whose execution violates an RV spec could indicate how software may fail on untested inputs. Based on this idea, these researchers discovered hundreds of bugs in open-source projects by using RV to monitor passing tests against specs of correct JDK API usage.

Broad RV adoption for everyday testing, especially in today's rapid continuous integration (CI) cycles, is hindered by the perception that it incurs high runtime overheads. This perception persists despite decades of tremendous algorithmic efficiency improvements to make RV scale [10, 16, 27, 38, 41, 75, 89, 94, 99, 100, 104, 111, 112, 135, 138]. Part of the reason is that many techniques that improved RV overhead were only evaluated on carefully curated benchmarks that have no unit tests, e.g., DaCapo [24], MiBench [56], or RV competition data [13, 14, 16, 46, 114]. Also, the recent work on RV of tests were not focused on overheads [84, 86, 101], they were evaluated on between one and 21 open-source projects [87, 89, 138], or they did not evaluate RV realistically [73] (§8 explains the latter).

Given the recent demonstration of RV's bug-finding benefits, an in-depth and large-scale study of RV overheads during testing is

needed to help (1) quantify and evaluate the spread of high RV overheads more broadly among open-source projects; (2) understand root causes, contributing factors, program locations, and usefulness for bug finding of overheads that RV incurs; and (3) obtain insights that can be leveraged to reduce these overheads.

We conduct the first in-depth study of RV overhead during testing. We measure overheads while using a state-of-the-art RV tool, JavaMOP [72], to monitor 182,547 developer written unit tests in 1,544 open-source projects (total: 10,897,631 SLOC) against 160 specs of correct JDK API usage from prior work [83, 94]. Next, we collect traces that monitors observe. To do so, we extend JavaMOP (§3.3 has rationale and details). Lastly, we use a profiler [9] to see overheads incurred by RV components, and in program parts.

We make four main findings, among several other results.

1. RV Overhead Varies Widely. The mean overhead across all 1,544 projects is 23.6x, or 249.1 seconds. A project has the maximum relative overhead of 5,002.9x. Another project has the maximum absolute overhead of 28.7 hours. Legunsen et al., claim that absolute RV overhead of up to 12.48 seconds is acceptable during software testing [86]. By that threshold, 40.9% (or, 632) of these projects can use RV with acceptable overhead during testing.

2. RV During Testing Is Very Wasteful. Only 0.13% of 3,432,878,467 collected traces are unique. The rest are wasteful repetitions of these 0.13%—each trace maps to a program path and spec pair, so monitoring the same path against the same spec after the first time cannot reveal new bugs. Repeated checking in RV of deployed systems [55] is useful to recover from violations or mitigate attacks.

3. Instrumentation Dominates. 60.5% of total RV time across all projects is spent on instrumentation, not monitoring. (JavaMOP instruments during class loading by default.) This finding goes against conventional wisdom in RV research that mostly aim to reduce overhead via faster monitoring. When RV is used in deployment, instrumentation time is a one-time startup cost, but faster instrumentation is needed during testing.

4. Monitoring Tests and Libraries Is Costly, but Necessary. Excluding instrumentation, 36.74% of monitoring time is spent in test code (21.87%) or third-party libraries (14.87%). But, arguments can be made for (excluding them could lead to false positives/negatives [89]) or against (tests are not deployed, and developers often have no control over libraries) monitoring these components (§4.4).

The computational complexity of the general RV problem can be arbitrarily hard [117]. So, our findings can provide new empirical basis for future techniques and tools that leverage the nature of RV during testing to reduce its overhead. We highlight future directions and make several suggestions to the RV research and development community. Also, to begin assessing the feasibility of realizing such techniques and tools, we perform two more experiments (§5).

First, we attempt offline, compile-time instrumentation for 1,532 projects with overhead less than an hour, and measure the time to monitor pre-instrumented code during testing. Offline instrumentation is much slower than JavaMOP's instrumentation and fails in 253 projects, e.g., because instrumentation by other tools conflicted with ours. So, even ignoring its high cost, offline instrumentation is not always possible. For 1,279 projects that we pre-instrument, RV overhead reduces by 8x, on average.

```
1 Collections_SynchronizedCollection(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 1: CSC Spec, written in an AspectJ-based DSL.

Second, offline instrumentation must cope with code and library evolution, and tight CI budgets. So, we evaluate if instrumenting only what changes per commit can reduce its cost. Compared to recent program-analysis based evolution-aware RV techniques that only re-monitor specs that are affected by code changes [89, 138], incremental offline instrumentation is up to 4.53x faster. Unlike those techniques, incremental offline instrumentation is safe: it cannot miss new violations if tests pass and tests are deterministic.

This paper makes the following contributions:

- ⋆ **Study.** We conduct the first in-depth and large-scale study of RV runtime overheads during testing, using 1,544 Java open-source projects and 160 specs of correct JDK API usage.
- ⋆ **Findings.** Our results provide insights that future work can exploit to further reduce RV overhead during testing.
- ⋆ **Proofs of Concept.** Our initial experiments to begin exploiting some of our findings show promising results.
- ⋆ **Data.** We make all tools, scripts, and raw data from our experiments publicly available to aid reproduction and further studies.

Our artifacts, including scripts, code, and data are available at this URL: https://github.com/SoftEngResearch/rv-study-artifacts.

2 Background and Examples

2.1 An Example Spec and Its Monitoring

Figure 1 shows a spec, CSC, that we monitor; it was formalized by Lee et al. [83, 94] and exemplifies three parts of RV specs: event definitions, properties, and handlers. Lines 3–10 define related events and when to signal them at runtime: (1) `sync` (lines 3–4), after calling `Collections.synchronizedCollection()` to get collection `c`; (2) `syncMakeIter` (lines 5–6), after calling `c.iterator()` from code that synchronizes on `c`; (3) `asyncMakeIter` (lines 7–8), after calling `c.iterator()` from code that is *not* synchronized on `c`; and (4) `useIter` (lines 9–10), after calling any method on iterators returned by `c.iterator()` from code that is *not* synchronized on `c`. Line 11 is the safety property, formalized as a regular expression; it matches if `c.iterator()` is called from code that is not synchronized on `c`, or if `c.iterator()` is called from code that is synchronized on `c`, but the returned iterator is later used without synchronizing on `c`. Code that violates this property can be non-deterministic [108]. So, when CSC monitors observe matching traces, the handler on line 12 is invoked; it can be any user-provided, e.g., error-recovery, code. But, for finding bugs, we print a message.

Figure 2 shows how RV amplifies tests' bug-finding ability (we elide some multi-threading code). There, `sum` (lines 1–12) adds a list of integers, and two unit tests assert that `sum` is correct on a list (line 13) and its reverse (line 14). Such tests will almost always pass

```
1 public int sum(List<Integer> list) {
2   Collection<Integer> collection = Collections.synchronizedCollection(list);
3   /* INSTRUMENTATION: Collections_SynchronizedCollection.sync */
4   int total = 0; Iterator<Integer> iterator;
5   synchronized(collection) {
6     iterator = collection.iterator();
7     /* INSTRUMENTATION: Collections_SynchronizedCollection.syncMakeIter */
8   }
9   while (iterator.hasNext()) total += iterator.next();
10  /* INSTRUMENTATION: Collections_SynchronizedCollection.useIter */
11  return total;
12 }
13 @Test public void testSum(){ assertEquals(6, sum(Arrays.asList(1, 2, 3))); }
14 @Test public void testSumRev(){ assertEquals(6, sum(Arrays.asList(3, 2, 1))); }
```

Figure 2: An example of monitored code and its unit tests.

Table 1: Summary statistics on 1,544 projects that we study, in terms of no. of test methods (#Tests), end-to-end test time in seconds (t), lines of code (SLOC), % statement coverage (cov^s), % branch coverage (cov^b), no. of commits (#SHAs), years since first commit (age), and no. of stars (#★).

	#Tests	t	SLOC	cov^s	cov^b	#SHAs	age	#★
Mean	118.1	8.2	7,034.6	53.8	45.8	240.1	8.9	162.8
Med	22.0	2.0	2,273.0	58.3	48.2	95.0	9.0	29.0
Min	1	1.3	21	0.0	0.0	1	0	0
Max	17,874	1,561.9	6.2×10^5	100.0	100.0	17,223	26	20,198
Sum	1.8×10^5	12,676.5	1.1×10^7	n/a	n/a	n/a	n/a	n/a

and likely never reveal a subtle bug in `sum`: line 5 synchronizes on a `Collections.synchronizedCollection()` (line 2) before obtaining `iterator` from it, but line 9 uses `iterator` in non-synchronized code. RV of these tests against CSC reveals the bug: instrumenting `sum` on lines 3, 7, and 10 (shown as comments) causes CSC monitors to observe the violating trace: `sync syncMakeIter useIter`. Both tests produce this trace, but one of them suffices to find the bug. So, RV overhead is wasted the second time w.r.t. CSC. Monitoring CSC helped find several confirmed bugs in open-source projects [86].

2.2 JavaMOP and Other Specs in Our Study

We use JavaMOP [72, 76] as the RV tool in our study because: (1) it is mature and widely cited; (2) it can monitor multiple specs simultaneously [94]; (3) it was evaluated during testing with many open-source projects [73, 84, 86, 89, 101]; (4) it ships with specs of correct JDK API usage, including CSC (Figure 1), that we use [83, 94]; and (5) it incorporates decades of RV progress [32–34, 64, 75, 77, 94, 100]. JavaMOP supports monitoring specs in different formalisms: past- and future-time linear temporal logic (LTL), extended regular expressions (ERE), finite-state machines (FSM), context-free grammars (CFG), string-rewrite systems (SRS), etc.

3 Experimental Setup

We organize our study around four questions, the answers to which could yield insights on reducing the overhead of RV during testing:

RQ1. What are the overheads of RV during testing, and how do they correlate with various program characteristics?

RQ2. How much RV monitoring is wasted during testing?

RQ3. What proportion of RV time is spent among its components?

RQ4. What proportion of RV time is spent monitoring different program components?

RQ1 aims to quantify the magnitude and prevalence of high RV runtime overhead in a realistic setting during testing in many open-source projects. RQ2 aims to measure how much RV monitoring is (un)necessary for bug finding. Lastly, RQ3 and RQ4 aim to measure where RV time is spent within JavaMOP and the studied projects. We next describe our process for finding open-source projects for our study, the set of projects that we use, our process for collecting overhead (and other) data, and our experimental settings.

3.1 Project Selection

Initial Set of Projects. We start with 7,533 open-source Java projects that use Maven [4] and whose last commit is after 1/1/2019. These projects are from (1) prior work on RV [69, 86] and testing [105], and (2) our GitHub API query that we use to find projects with greater than 10 stars. We choose Maven, like prior work on RV of test [84, 86, 87, 89, 101, 138]; future work can evaluate other build systems (e.g., Gradle [54] or Bazel [22]).

Filtering Projects. We automatically filter out all but 1,528 of 7,533 projects in our initial set. We start by filtering out 94 with no tests, 3,727 where build, compilation, or tests fail, 22 that are not cloneable (they likely went private since prior work used them), and 16 whose failure we could not quickly figure out (e.g., JVM crashes with no easy-to-debug output). JavaMOP monitors failing tests, but we only keep projects with passing tests, to more fairly compare times with and without RV. We do not control for flaky tests [23, 52, 57, 82, 93, 98, 110, 123, 125] in subsequent experiments. Next, we run JavaMOP on the 3,674 projects that remain at this point, and further filter out 2,001 where JavaMOP fails or finds no events, and 145 with no statement and branch coverage.

Manual Augmentation. Several projects on which JavaMOP failed during automated filtering merely time out after our initial 1-hour limit. So, we manually investigate whether JavaMOP would work on these projects with additional time or manual set up, finding 16 more that meet our other criteria. Adding these to the 1,528 projects that we obtain automatically, we get 1,544 projects in our study.

3.2 Characteristics of 1,544 Evaluated Projects

Due to space limits, we only provide summary statistics about these projects. Our artifact has detailed data on these 1,544 projects, their GitHub URLs, and the versions that we use. Table 1 shows the arithmetic mean (Mean), median (Med), minimum (Min), maximum (Max), and total (Sum) for eight program characteristics (see Table 1 caption); "n/a" denotes meaningless sums.

We next describe the Min row in Table 1, which may not be self-explanatory. The minimum numbers for test methods (#Tests) and test time (t) in Table 1 are small, but they reflect the fact that we do not discriminate among projects based on number or duration of tests. The minimum percentage of statement coverage (cov^s) shows up as 0.0% due to rounding; it is 0.03%. The project with 0.0% branch coverage (cov^b) has no branches. We measure project age in years, so 0 means that a project is less than 12 months old at the time of selection. Lastly, we query GitHub for projects with ≥10 stars, but some projects that we obtain from prior work have no stars, yielding a minimum number of stars of 0.

3.3 Extending JavaMOP

Theoretically, RV tools like JavaMOP check if traces violate specs. But, in practice, RV tools do not store traces. Rather, they implement online event-by-event algorithms that detect violations when monitors transition to error states. So, to answer RQ2, we extend JavaMOP with a non-default trace-collection mode that records in memory the trace that each monitor observes at runtime and persists those traces to disk during JVM shutdown. Traces that we collect are sequences of event and program location pairs, so there is a 1-to-1 mapping of unique traces to monitored program paths.

Extending JavaMOP with a trace-collection mode involved non-trivial engineering. RV is known to generate *billions* of events and hundreds of millions of monitors per project during testing [89]. So, even with an abundance of memory, trace-collection must be highly optimized and correct. Else, RV could be too slow, induce timing-related failures, or produce wrong verdicts if events are missed. We optimize our trace-tracking feature to be efficient enough in time and space for RQ2 experiments, whose times we do not report.

We validate our JavaMOP extension on a subset of projects in two ways. First, we compare violations reported in trace-collection mode with violations that default JavaMOP reports. We found no difference. Second, we compare the final values of monitor and event counters that JavaMOP optionally keeps (these are cheaper to collect than tracking traces) with the number of traces and events that are produced in trace-collection mode. The differences that we found are due to test non-determinism.

3.4 Running Experiments

Answering RQ3 and RQ4 requires profiling JavaMOP runs. So, we automate the use of async-profiler [9], an accurate, low-overhead, and widely used (including by the IntelliJ IDE [74]) profiler for Java. Profiling is inherently a statistical sampling approach, so multiple runs are typically required for performance measurements. But, our goal is not to use the profiler to measure performance (so we are not really concerned with profiler overhead), but to compute the proportion of time that RV spends in different parts of the monitoring process or the monitored program. So, after running the profiler several times on a subset of 26 projects and finding that our conclusions remain w.r.t. these proportions, we report our findings for all 1,544 projects based on only one profiler run.

We write scripts to (1) run JavaMOP and collect its data; and (2) run a profiler, dump its raw data, and post-process that data. We also write Maven extensions to (1) integrate JavaMOP with project builds, (2) measure code coverage with JaCoCo [103], (3) perform compile-time instrumentation (§5.1), and (4) integrate the profiler.

We perform all experiments in Docker containers, to aid reproducibility. Our artifact contains our Docker files and how to use them. We run all experiments involving absolute time measurement on an Intel® Xeon® Gold 6348 machine with 512GB of RAM and 112 cores, running Ubuntu 20.04.6 LTS, Java 8, Maven 3.8.8.

4 Results

4.1 RQ1: RV Overheads

We first discuss RV overheads during testing that we found in all 1,544 projects. Then, we report on correlations between RV overhead and various program characteristics. The general RV problem can be arbitrarily hard [117], so positive correlations could reveal factors that can be leveraged to reduce RV overhead in practice or serve as good starting points for our study of causes of RV overhead.

Table 2: Overhead-related results for projects in our study: end-to-end test time in seconds without RV (t), end-to-end test time in seconds with RV (t^{rv}), absolute overhead in seconds (t^{rv}–t), relative overhead (t^{rv}/t), no. of generated monitors (#Mon), and no. of signaled events (#Event).

	t	t^{rv}	t^{rv}–t	t^{rv}/t	#Mon	#Event
Mean	8.2	257.3	249.1	23.6	5.3×10^{6}	1.4×10^{8}
Med	2.0	23.0	18.9	7.9	9,109.0	1.6×10^{5}
Min	1.3	3.0	−14.5	0.3	76	2
Max	1,561.9	1.0×10^{5}	1.0×10^{5}	5,002.9	1.4×10^{9}	6.0×10^{10}
Sum	12,676.5	4.0×10^{5}	3.8×10^{5}	n/a	8.2×10^{9}	2.2×10^{11}

Table 2 shows summary statistics for our overhead-related measurements. There, times are computed for all 1,544 projects in our study, while monitor and event information is for a subset of 1,542 projects where JavaMOP's monitor and event counters produced an output. Negative "Min" absolute overhead (t^{rv}–t) in Table 2 (tests are faster with than without RV) is not noise, and occurs in one project. Negative overheads can occur if JavaMOP's instrumentation changes garbage-collection behavior or memory layout such that the program is accidentally optimized [75, 77, 86, 99].

We discuss five findings from the data summarized in Table 2:

1. On a positive note, 632 (or 40.9%) of 1,544 projects could start using RV today, based on Legunsen et al.'s claim that an absolute RV overhead of 12.48 seconds is acceptable during testing [86].
2. Even with a larger set of projects, the "Mean" relative (23.6x) overhead is higher than the highest average overhead of 9.4x reported in prior work on RV during testing [84, 86, 89].
3. The 462 8th, 9th, and 10th decile projects have very high "Mean" absolute (59.90, 96.50, and 2,240.5 seconds, respectively) and relative (23.44x, 25.15x, and 134.10x, respectively) overheads.
4. Absolute RV overheads are ≥ 1 hour in 12 projects and between 10 minutes and an hour in 19 projects. The "Max" absolute overhead is over a day (28.7 hours), for a project whose test-running time without RV is 46.84 seconds.
5. The "Max" numbers of monitors and events generated are greater than 1.4 billion and 60 billion, respectively. But, projects with relatively few monitors and events have very high overheads. For example, some projects in Figure 3 have few events but very high overheads.

We conclude that high RV overhead is still a problem overall, and warrants continued research to reduce it further. Projects with high overheads are unlikely to adopt RV during continuous integration. Recent evolution-aware RV techniques [89, 138] are promising, but they have not yet been evaluated at this scale. Also, prior work on reducing RV overhead typically focused on reducing generated monitors or events [5, 28, 32, 34, 38, 75, 89, 111, 112], but our fifth finding suggests that there are more factors behind RV overheads.

To begin investigating factors contributing to RV overheads, we check correlation with nine program characteristics—the number of monitors (#Mon) and events (#Event), and seven in Table 1: number of test methods (#Tests), end-to-end test time in seconds without

Table 3: Pearson's correlation coefficient with several program characteristics.

	#Mon	#Event	#Tests	t	SLOC	cov^s	cov^b	#SHAs	age
Absolute overhead (with outliers)	0.1746	0.0982	0.0121	0.1444	0.1238	-0.0106	-0.0109	0.1534	-0.0034
Relative overhead (with outliers)	0.1344	0.0459	0.002	0.0083	0.0707	0.0123	0.0006	0.0799	0.0007
Absolute overhead (without outliers)	0.527	0.4365	0.176	0.478	0.1588	0.0851	0.0566	0.1701	0.0165
Relative overhead (without outliers)	0.351	0.3179	0.0845	0.2695	0.0884	0.0255	0.002	0.1104	-0.0166

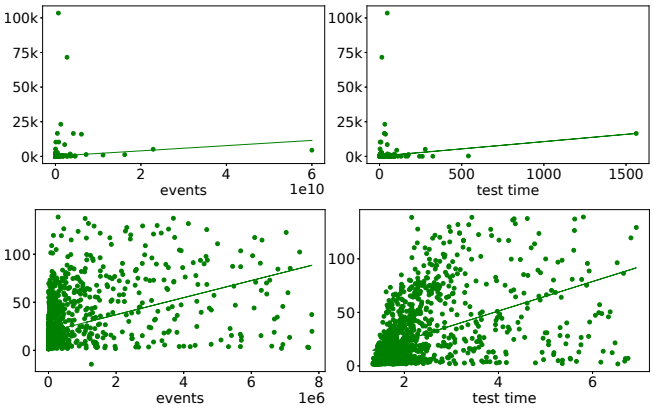


Figure 3: Correlation of absolute overhead with # of events (left) and test time without RV (right), with (top row) and without (bottom row) outliers.

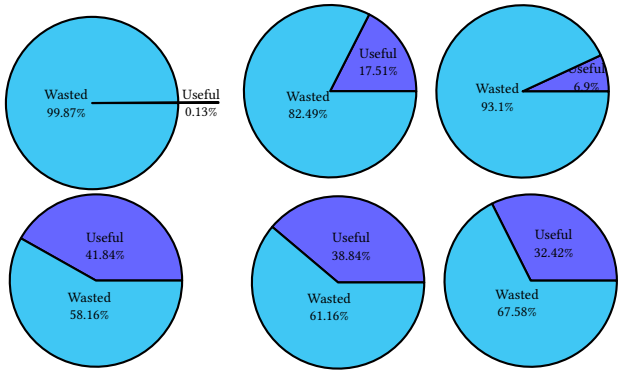


Figure 4: Proportions of sum (left), mean (middle), and median (right) of traces (top row) and events (bottom row) that are unique (dark blue) and wasted (light blue).

RV (t), lines of code (SLOC), branch coverage (cov^b), statement coverage (cov^s), number of commits (#SHAs), and age.

Table 3 shows the Pearson's correlation coefficients (computed using Matplotlib [97]) between RV overhead and these characteristics. The top two rows show the coefficients with outliers *included* and the bottom two rows show them with outliers *excluded*. Overall, there is weak correlation with all considered characteristics when outliers are included. With outliers excluded, we find that numbers of monitors and events, and test time have weak to moderate positive correlation with overheads. So, overall there seems to be more factors behind RV overheads than these characteristics, but more work to speed up monitoring could help reduce these overheads.

Due to space limits, we show only scatter plots for events and test time without RV, with and without outliers, in Figure 3. The plot for correlation with number of monitors is similar to that of events; other plots show even weaker correlation than that of time.

This lack of correlation means that a deeper analysis of RV overhead is needed. So, in the other research questions (§4.2 – §4.4), we investigate RV overhead during testing from other angles.

4.2 RQ2: RV's Wastefulness during Testing

We hypothesize that repetitive, wasteful monitoring is a factor in RV overhead during testing for two reasons. First, multiple tests often validate the same program path using different inputs. But, checking a path multiple times against a spec is wasteful (§1 and §2). Second, older prior work [111] showed that repeated monitoring inside program loops contributes to RV overhead when monitoring one spec against single executions in the DaCapo benchmarks. So, our more realistic multi-execution and simultaneous multi-spec monitoring could compound the impact of loops.

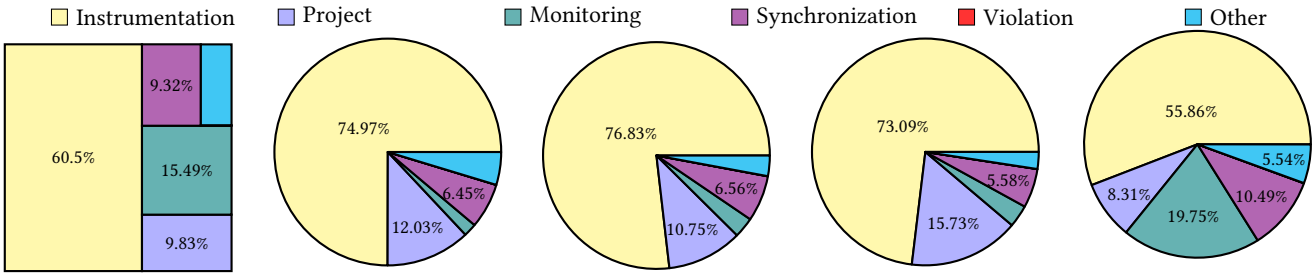
We test our hypothesis by conducting the first study of repetitive and wasteful monitoring during RV of tests. To do so, we first run the trace-collection mode in our extended JavaMOP implementation (§3.3) to obtain traces that monitors observe. (Existing JavaMOP counters report numbers of monitors and events, not observed traces.) Then, we analyze traces from 1,454 projects[1] in two ways. First, we compute the number of useful and wasted traces and events across all these projects and within each project. If a trace for a spec is observed x times, only one of these traces is useful; the other $x - 1$ traces are wasted. Only events in useful traces are useful; the others are wasted. Second, we qualitatively analyze the locations and nature of wasted traces to obtain new insights that could be exploited by future work on reducing RV overhead.

The top row in Figure 4 shows the proportion of useful and wasted traces. There, the leftmost pie chart shows that, considering all traces in all 1,454 projects, only 4,590,494 (or 0.13%) of all 3,432,878,467 are useful. There is a one-to-one mapping of traces to monitors, so one implication is that only 0.13% of monitors that RV generates during testing of these projects are necessary for bug finding. The middle and rightmost pie charts in the top row of Figure 4 show, respectively, the proportions for the arithmetic mean across all 1,454 projects and the median project. So, high proportions of wasted traces are widespread among these projects.

The bottom row in Figure 4 shows the proportion of useful and wasted events, in the same order as for traces. Interestingly, the proportion of useful events is much higher than the proportion of useful traces. To put this comparison in perspective, across these projects, a hypothetically perfect future technique that only generates the 0.13% of monitors that are necessary for bug finding would still process 38.84% of 51,203,201,000 events.

Analysis of Repetitive Traces. Our post-processing of the trace data reveals that the average length of useful traces is over 5,200

[1]We could not obtain traces from the remaining 90 projects because traces exceeded our disk space, trace-collection ran out of memory, or tests failed due to timeout.



(a) All projects **(b) First-quartile** **(c) Second quartile** **(d) Third quartile** **(e) Fourth quartile**

Figure 5: Percentage of time that JavaMOP spends performing load-time instrumentation (Instrumentation), running the code under test (Project), handling monitors and events (Monitoring), waiting for locks (Synchronization), printing violations (Violation), and other processes (Other). We only show percentages that are above 5%.

events long. Also, we find that testing does compound the impact of loops. But, during our analysis, we also found an insight that could be exploited to reduce wasted traces and events. Our data showed that more than 50% of traces in some loop-heavy programs come from five or fewer methods. So, future work may be able to use a method-based analysis to reduce wasted traces in a manner that could be more tractable than existing loop-based analysis which is promising, but expensive, and took hours on the single-version programs with no unit tests in the DaCapo benchmarks [111]. Also, we believe it is now timely for research to revisit that loop-based analysis with a view to speeding it up, e.g., by making it incremental during software evolution. Finally, we find that $1.4{\times}10^6$ useful traces are in the code under test (CUT) or unit tests, $1.7{\times}10^6$ are in libraries, and only 51,702 of them cross the code-and-library boundary. When the frequencies of occurrence of these useful traces is included, these numbers are $2.2{\times}10^9$, $5.7{\times}10^8$, $2.5{\times}10^6$. This finding about the partitioning of traces is important: it can be a basis for source-only or binary-only program analysis to be separately designed and used to reduce wasted traces (and their events) that are only in the CUT and unit tests, or only in the library, respectively.

We conclude from RQ2 that RV overhead is incurred on wasteful and repetitive monitoring that is not useful for bug finding. Future techniques that aim to reduce wasted monitoring during testing could target the reduction of this waste.

4.3 RQ3: Time Spent in RV Components

Here, we turn our attention inwards: what proportion of time is spent in the different components of an RV tool? Answering this question is important to (1) help identify parts of the RV process that future work should target; and (2) provide perspective on how well prior algorithmic advances on RV perform during testing.

To answer RQ3, we run experiments in which we attach a profiler to the JVM where RV is monitoring tests, and then process the raw profiling data. We only report profiling results for 1,525 projects, after excluding projects where raw profiling data was too large for our custom processor to handle, or where the profiler failed.

Figure 5 shows proportions of time in different RV tool components. There, all plots are based on sums over groups of projects—Figure 5a is for all 1,525 projects in RQ3, Figures 5b–5e are for subsets of projects in the first, second, third, and fourth quartiles, respectively, in increasing order of absolute RV overheads (§4.1).

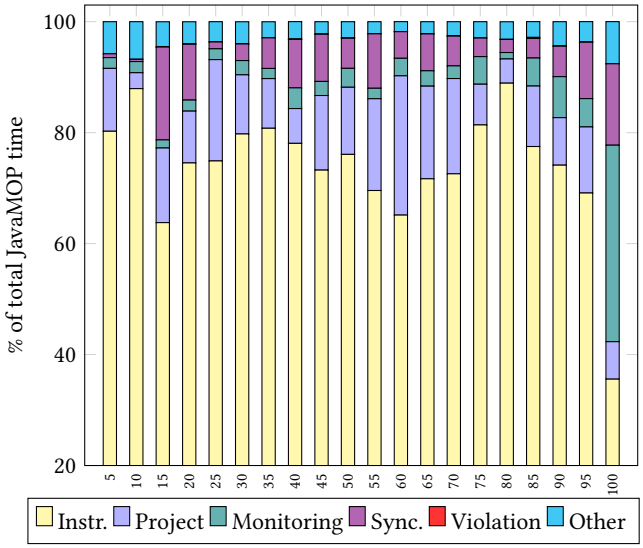


Figure 6: % of JavaMOP time spent per component, aggregated for projects in every fifth percentile; y-axis starts at 20%.

Surprisingly, we find that JavaMOP spends most of its time on instrumentation rather than on monitoring traces. Across all 1,525 projects (Figure 5a), 60.5% of RV time is spent instrumenting the code under test, test code, and the parts of third-party libraries that a project uses. The proportion of time spent on instrumentation is even higher for projects in the first, second, and third quartiles—74.97%, 76.83%, and 73.09%, respectively. JavaMOP uses AspectJ [8, 79], a mature and well-engineered tool. We confirmed that JavaMOP does not use any experimental AspectJ settings. In fact, we use JavaMOP (and AspectJ) as prior work did [84, 86, 89, 101].

Surveys, discussions, and techniques for improving RV instrumentation exist [15, 31, 104, 115, 127–130], but we do not know any prior work that quantifies the proportion of RV overhead that is spent on instrumentation during testing of open-source projects. Also, prior work on RV during testing [84, 86, 89, 101] did not break down where RV spends its time. Yet, the findings in Figure 5 suggests that lowering instrumentation costs should be one of the next main frontiers of research on reducing RV overhead during testing. Offline instrumentation at compile time is an alternative to the load-time instrumentation that JavaMOP performs by default. In §5,

we find even higher costs and other problems with offline instrumentation, and propose practical ways to mitigate them pending future research and development on faster instrumentation for RV.

Figure 5a also shows that RV spends 24.84% of its time on runtime checking: managing monitor generation and event handling (Monitoring, 15.49%), waiting for locks to avoid concurrency problems (Synchronization, 9.32%), and on handling violations (Violation, 0.03%, not visible in Figure 5a). These relatively small proportions are made possible by tremendous progress made in prior work on speeding up RV. However, research on faster but correct synchronization could be productive to further reduce RV overhead. Handling violations is negligible. But, as we show in §5, we find and fix a performance bug in JavaMOP's violation handling during our study. Without this fix, violation handling would have inflated the RV overheads that we report in this paper.

The coarse-grained aggregation in Figure 5 may occlude finer-granularity observations. Given space limits, we partially address this problem in Figure 6, showing the breakdown of RV time along the same categories as in Figure 5 per five percentiles. There, it can be seen that 5.1% of RV time is spent on monitoring (not instrumentation) for projects between the 91st and 95th percentiles. However, for projects above the 96th percentile, monitoring dominates the RV overhead. We make two observations about these projects with high monitoring costs. First, more algorithmic advances are likely needed to scale RV beyond the projects and specs that we evaluate. Second, our manual analysis of monitoring dominated projects confirm that it is the cost of handling events and monitors that are costly. In particular, some projects spent more than 50% of the time monitoring a spec, including the max-overhead project in our study, which spent a day monitoring one spec.

Figure 5 does not show any information about monitor garbage collection [75]. But, on average, JavaMOP spends only 1.7% of its time on garbage collection (for monitors and regular Java objects). So, this omission does not affect our conclusions. Note, though, that we run experiments on a server with a lot of memory, so garbage-collection may be costlier in resource-constrained settings.

4.4 RQ4: Time Spent in Program Components

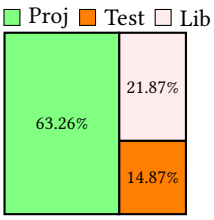


Figure 7: % of JavaMOP time spent in code under test (Proj), test code (Test), and 3rd-party libraries (Lib).

Ignoring instrumentation, we here investigate where in an open-source projects monitoring time is spent. This investigation is important: the time RV spends in test code or third-party libraries was often not measured in prior work on RV, which often was evaluated on benchmarks that have no libraries or unit tests. But, RV during testing of open-source projects must cope with tests and libraries, which may need to be monitored (to avoid false positives or false negatives [89]) and the performance costs of doing so. To answer RQ4, we run similar profiler-based experiments as we did for RQ3. RQ3 concerns time spent in JavaMOP components. Here, we are concerned with, "what part of the program was running when calls to JavaMOP components are made: tests, library code, or code under test?". We use the 1,525 projects in RQ3 to answer RQ4.

Figure 7 shows the results, where the percentages are computed from the sums across RQ4 projects. 36.74% of RV overhead is spent on monitoring test code (21.87%) or third-party libraries (14.87%).

Monitoring test code or libraries may (not) be a waste. Developers can monitor them if they care about spec violations in libraries they rely on (those violations can cause harm in deployment, and several confirmed bugs that Legunsen et al. used RV to find are in libraries [84, 86]). Also bugs in test code are important (e.g., such bugs can reduce tests' bug-finding effectiveness). Lastly, excluding tests and libraries from monitoring can lead to false positives or negatives during RV [89]. On the other hand, developers may not want to monitor libraries that they have no control over (for legacy, legal, or contractual reasons), or tests, which are often not deployed.

Regardless of the view about the wastefulness of monitoring test code or libraries, our RQ4 results quantify at scale what parts of a program RV spends its time in, and provides a data point that developers and researchers could use to decide on what to monitor.

Table 4: Data on 1,279 projects on which we evaluate compile-time instrumentation. Table 2 describes column headers.

	t	t^{rv}	t^{rv}–t	t^{rv}/t	#Mon	#Event
Mean	6.3	54.6	48.3	12.3	3.9×10^6	6.4×10^7
Med	1.9	18.3	14.6	6.5	5,427	94,261
Min	1.3	3.0	1.6	1.1	76	2
Max	324.5	2,811.1	2,798.0	493.4	1.4×10^9	1.1×10^{10}
Sum	8,076.7	69,845.3	61,768.5	n/a	4.9×10^9	8.2×10^{10}

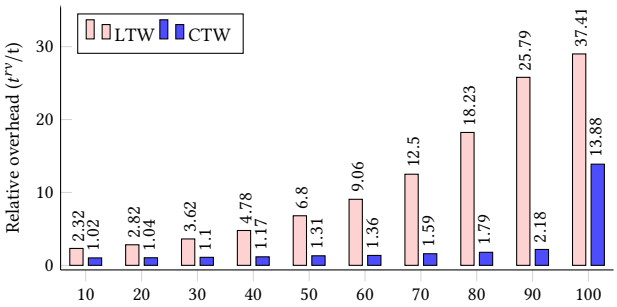


Figure 8: Relative RV runtime overheads with load-time (LTW) vs. after offline (CTW) instrumentation, for 1,279 projects, aggregated for every tenth percentile.

5 Discussion

5.1 Reducing Instrumentation Costs

Load-time instrumentation dominates RV overhead in most projects (§4.3), so we evaluate the potential benefits and challenges of performing compile-time instrumentation offline before running tests. We do so using 1,532 projects where absolute RV overhead is less than 1 hour. We exclude projects with over 1 hour overhead because they are not dominated by instrumentation and faster or offline instrumentation cannot provide much speedup for them. Also, their profiler output is often too large to precisely analyze.

One challenge is that offline instrumentation often fails; it fails for 253 of 1,532 projects because of an assortment of AspectJ errors: incompatible bytecode, typing issues, etc. Table 4 shows data, based

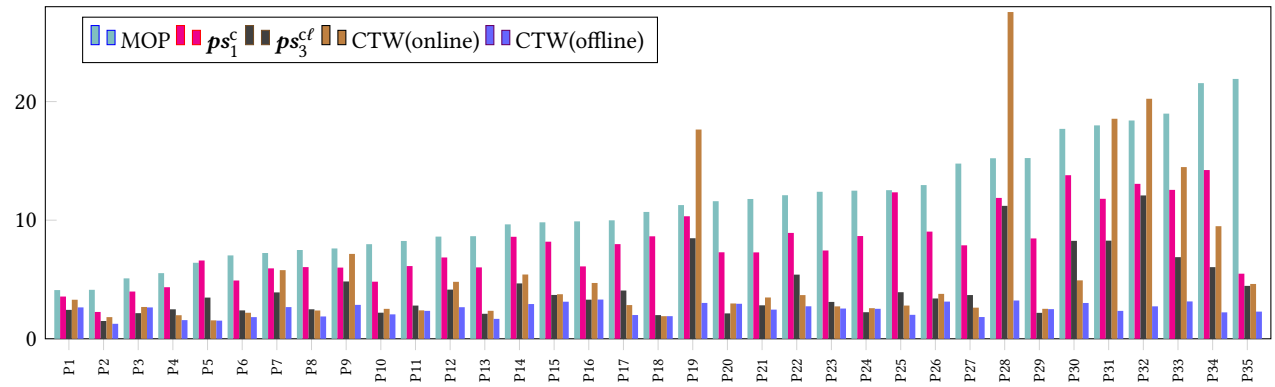


Figure 9: Average relative RV overhead across several versions of 35 projects: JavaMOP (MOP), evolution-aware RV (ps_1^c and $ps_3^{c\ell}$), and two incremental offline instrumentation strategies (CTW(online) and CTW(offline)).

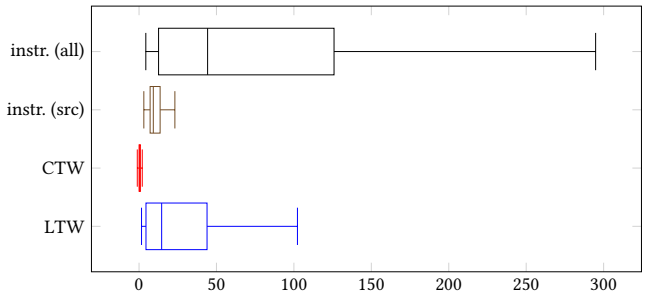


Figure 10: Distribution of times (in seconds) for offline instrumentation (instr.), RV after offline instrumentation (CTW), and RV with load-time instrumentation (outliers removed).

on load-time instrumentation about the 1,279 projects where our offline instrumentation succeeds.

Figure 8 shows the potential benefits of using offline instrumentation, whose times are excluded, over using load-time instrumentation during testing. There, we plot the relative overhead (t^{rv}/t) for every decile among these 1,279 projects, ordered by their absolute overhead in §4.1. All groups in Figure 8 show reduction in relative RV overhead; the maximum is the 9th decile, which sees a 11.83x aggregate speedup. Individually, the project with the maximum speedup (not shown in Figure 8) sees a 40.78x speedup. Lastly, we observe an average speedup of 8x across all these projects.

Figure 10 shows the distribution of times to (1) instrument the code under test, unit tests, and libraries (instr. (all)); (2) instrument only the code under test and unit tests (instr. (src)); (3) run JavaMOP after offline instrumentation (CTW); and (4) run JavaMOP with load-time instrumentation (LTW). We make two main observations from Figure 10. First, comparing "instr. (all)" and "instr. (src)" shows that offline instrumentation of libraries is very costly, compared to offline instrumentation of only the code under test and unit tests. Second, running JavaMOP after offline instrumentation is much faster than running JavaMOP with load-time instrumentation. But, the end-to-end time of offline instrumentation ("instr. (all)" plus "CTW") is very slow, and unlikely to fit in tight CI budgets.

We conclude that offline instrumentation can reduce RV overhead for a large subset of projects in our study. But, realizing these speedups requires longer-term research to speed up offline instrumentation, especially for libraries. We next evaluate the feasibility and potential benefits of incremental instrumentation as software evolves, which could be a shorter-term engineering solution.

5.2 Amortizing Instrumentation Costs

To evaluate the feasibility of incremental offline instrumentation as a way to amortize its costs during evolution, we select 35 of the projects with the highest LTW minus CTW times in Figure 8. (The greater the LTW minus CTW time, the more RV overhead is dominated by instrumentation.) Then, we select up to 50 historical versions of these projects from GitHub where at least one Java file changed, code compiles, and tests pass with and without JavaMOP.

For all 1,210 versions of these 35 projects that we obtain, we measure the overhead of five approaches, relative to the time to run tests without RV: (1) run JavaMOP from scratch with load-time instrumentation (MOP); (2) run safe but slow evolution-aware RV (ps_1^c); (3) run the fastest but unsafe evolution-aware RV ($ps_3^{c\ell}$); (4) run JavaMOP after offline re-instrumentation of changed code or libraries (CTW(online)); and (5) run JavaMOP after offline re-instrumentation of only changed sources (CTW(offline)).

CTW(offline) assumes that changed libraries are instrumented ahead of time, e.g., as part of the library update, or by downloading them from a (hypothetical) repository of pre-instrumented jars (§6).

We use the names (ps_1^c and $ps_3^{c\ell}$) of evolution-aware techniques as in prior work [89, 138]. They work as follows. Given a set of changed bytecode files, the set of impacted classes (IC)—those whose behavior can differ after the changes—is first computed using static change-impact analysis [88, 131]. Then, the set of affected specs (AS)—those whose events may be generated in IC—is computed using AspectJ. Finally, only AS is re-monitored in the new program version (JavaMOP re-monitors all). Evolution-aware RV is *safe* if it finds all violations that are new after the change. ps_1^c is safer than $ps_3^{c\ell}$: its IC is more comprehensive and it instruments AS everywhere (including libraries) except classes in the complement of IC. However, $ps_3^{c\ell}$ is faster than ps_1^c: its IC is smaller, it does not instrument AS in some IC (so it is unsafe by design), it does not instrument AS in libraries, and it does not instrument AS in classes that are in the complement of IC. Note that $ps_3^{c\ell}$ is the fastest evolution-aware RV technique that selects among specs. Also, JavaMOP, CTW(online), and CTW(offline) are safe.

Figure 9 shows the results. There, the trends among JavaMOP and the evolution-aware RV techniques are similar to those from prior

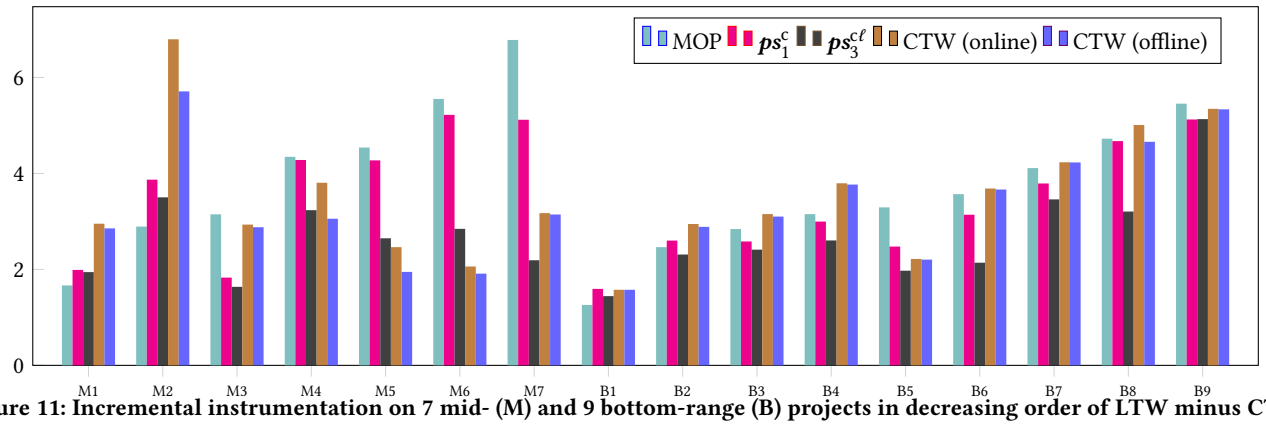


Figure 11: Incremental instrumentation on 7 mid- (M) and 9 bottom-range (B) projects in decreasing order of LTW minus CTW.

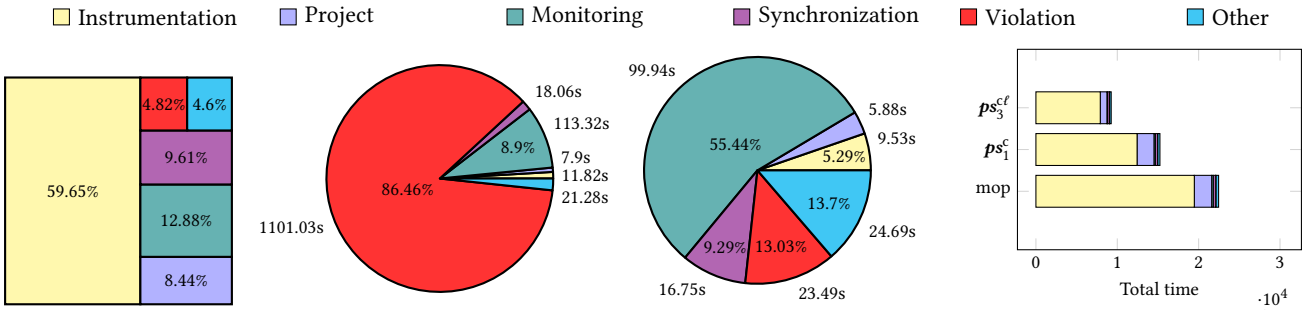


Figure 12: Before we fixed a bug in JavaMOP, handling violations was a sizeable portion of RV time across all projects (square chart). That bug dominated overheads in projects like lexburner/consistent-hash-algorithm before (left pie chart) but not after (rightmost pie chart) we fixed it. The bar graph breaks down where evolution-aware RV spends time.

work [89, 138] (we use a disjoint set projects with higher overhead). Overall, the results are promising. Surprisingly (as instrumenting libraries is very expensive—§5.1), even CTW(online) performs very well: it is faster than $ps_3^{c\ell}$ in 12 projects, faster than ps_1^c in 29 projects, slower than ps_1^c but faster than MOP in 2 projects, and slower than MOP in only 4 projects. CTW(offline) performs even better: it is always faster than ps_1^c and faster than, or equal to $ps_3^{c\ell}$ in 29 projects. Figure 11 shows results of incremental instrumentation on 344 versions of 16 other projects; seven are near the middle when LTW minus CTW is sorted in descending order; others are near the bottom. As expected, incremental instrumentation works better when LTW minus CTW is higher. Bottom-range projects do not benefit as much and analysis costs make them slower.

We conclude that it is worthwhile to develop instrumentation-driven evolution-aware RV techniques and tools in the short term. But, longer-term research on faster instrumentation seems to be needed to speed up RV during testing.

5.3 More Benefits of Profiling RV

Profiling RV overhead helped us in two other ways. First, it helped us discover a performance regression that had been in JavaMOP since 2013 [119]. The bug causes JavaMOP to obtain location information for violations (to aid debugging) by throwing an exception and parsing the stack trace for each violation, instead of reading from AspectJ, which already has precise location information. This bug is costly, especially when there are many violations, and could have caused us to report inflated RV overheads if we had not found and fixed it. To see how, consider the square chart on the left of Figure 12, which shows the sum across all 1,544 projects of the time that JavaMOP spends in it own components *before* we fixed the bug; 4.82% of all RV time is spent on handling violations, compared to 0.03% *after* we fixed the bug.

We noticed this bug while inspecting per-project profiling results, which showed that violation handling dominated overhead in some projects. For example, the leftmost circular pie chart in Figure 12 shows that 86.46% (or, 1,101.0 seconds) of RV time in `lexburner/consistent-hash-algorithm` was due to violation handling. That overhead reduced to 13.03% (or, 23.49 seconds) after we fixed the bug, as seen in rightmost pie chart in Figure 12. . We do not know which (if any) of the many papers that used JavaMOP since 2013 have results that are affected by this bug.

The second way profiling helped us is for interpreting the results of evolution-aware RV techniques. The bar plot in Figure 12, which is a sum of where JavaMOP, ps_1^c, and $ps_3^{c\ell}$ spend time among RV components, for 26 of 35 projects that we evaluated incremental offline instrumentation on. There, ps_1^c does not save much monitoring time, and it incurs a lot of instrumentation costs. Also, $ps_3^{c\ell}$'s monitoring-time reduction is somewhat obscured by instrumentation time. This, as far as we know, is the first component-level explanation for these evolution-aware RV techniques' savings.

5.4 Assumptions and Other Considerations

This paper assumes that users will likely want to use RV during continuous integration (CI), which is increasingly widely used today [66, 67]. An alternative is to only run RV periodically. We assume the CI setting for two reasons. First, like with regression testing [58, 59, 81, 85, 91, 92, 109, 124, 126, 137, 139–141], running RV during CI can help find/fix bugs earlier. Finding/fixing bugs

often grows costlier as time-to-discovery grows, and running RV only periodically risks finding bugs later. Second, using lightweight formal methods during CI has helped bring other techniques closer to everyday use [107], and we hope that doing so will help RV make similar impact. But, even for periodic runs, reducing RV overheads can still be helpful, especially when (rapidly) evolving code is deployed multiple times daily, e.g., [102].

This paper also assumes that users will likely want to check all available specs, especially if those specs ship with the RV tool like in JavaMOP. The alternative is to run our experiments, or for users to run RV, only with a fewer number of important specs (all 160 specs are not equally important [84, 86]). But, our experience shows that it is hard for researchers to determine which specs are more important than others per project. For example, [83] classifies specs in our study in decreasing order of violation severity as "error", "warning", or "suggestion". Yet, "suggestion" specs helped find bugs [84, 86, 101] at similar rates as "error" specs [84]. Also, developers are often not formal-methods experts and may find it even harder than researchers to determine important specs. So, we think that developers are likely to just run all available specs.

6 Suggestions for Future RV Research

Based on our study, we suggest several directions for research on further reducing RV overhead, to increase the chance of adoption.

Speeding Up Instrumentation. Future research and development should be invested into speeding up instrumentation, which is high and constitutes the main proportion of RV overhead during testing in many open-source projects (§4.3). Speeding up instrumentation may not have been a major concern in RV research, which often targets RV in deployment where instrumentation cost is incurred once during startup. But, RV during testing in today's rapid CI cycles will re-incur instrumentation costs from scratch, and could become a hindrance to broad developer adoption of RV.

Longer term, new frameworks that are fast or algorithms to speed up current instrumentation frameworks like AspectJ are needed. The RV community already started working in this direction. The recent work on BISM [127–130] is one example, but it still lacks robust tool support (e.g., it is not open sourced at this time of writing) and it was not yet evaluated at scale during testing of open-source projects. More work in this direction is needed.

In the interim, two directions can help speed up current instrumentation frameworks that have well-engineered tool support (like AspectJ). First, our proofs of concept (§5.1 and §5.2) motivate (incremental) compile-time instrumentation. But, several other problems with compile-time instrumentation must be addressed and better tool support for incremented compile-time instrumentation during CI is needed. Second, incremental compile-time instrumentation will not reduce the high costs of instrumenting libraries, and different RV users will wastefully re-instrument the same libraries. We next suggest an engineering solution to the second problem.

Open Repository of Instrumented Libraries. A public service and repository of pre-instrumented libraries (e.g., jars) would help. Several models are possible; we discuss two. (1) If an instrumented library is not in the repository, a user instruments it and uploads it to the repository with the list of specs used. The repository assures security and trust in the uploaded libraries. (2) The repository exposes a list of available specs for users to select from, the user provides a URL to the library (e.g., on Maven Central), and the repository produces an instrumented version of the jar and stores it for future download. A community process is needed to work out the details and ensure success. But, the gains can surpass the cost of setting up and maintaining such a repository.

New RV Techniques for Software Testing. Future techniques for making the monitoring aspects of RV (handling monitors and events, synchronization) scale better could leverage findings in this paper (e.g., the repetitiveness of checked program paths or the concentration of monitoring effort in few methods). In a sense, following recent work on RV of tests, this paper evaluates RV during testing using a technique that was designed for production runs. It may be possible to develop new RV techniques that are designed from ground up for use during software testing and CI.

Reporting RV Performance Improvement Results. Future work on speeding up RV during testing should be evaluated on open-source projects (not just benchmarks that have no unit tests or use no libraries) and report results that are accompanied by information obtained from the kinds of profiling that we do in this paper. Doing so would help to better evaluate impact on practice and to better interpret the results (in terms of what RV components' cost is being reduced), as we show for evolution-aware RV in §5.4.

7 Threats to Validity and Limitations

External. Our study results may not generalize beyond projects that we evaluate. To mitigate this threat, we study a set of open-source projects with more than 5 times the number used in prior work on RV during testing. Further, there was wide diversity among the projects that we evaluate in terms of various program characteristics (see Table 1). The specs that we use are about standard Java library API usage (allowing us to evaluate many projects); different results may be obtained for project-specific specs or other JDK API usage specs. The set of specs that we use is the largest publicly available one, and prior work [84, 86] showed that they are better than automatically mined specs. Future work can use more specs or project-specific specs.

We study RV overhead using developer written tests; different results may be obtained for automatically generated tests. Many RV tools and techniques other than JavaMOP exist; RV overheads may differ for others. But, several RV tools cannot check multiple specs simultaneously or have not been evaluated large sets of open-source projects. Also, JavaMOP is one of the most widely cited RV tools. Regardless of RV tool, instrumentation is a prerequisite for RV, so speeding it up should be beneficial more broadly. Finally, our results may not generalize to other programming languages beyond Java, or to other build systems.

Internal. We write scripts and Maven extensions to automate our experiments. Those artifacts and their output were reviewed several times, and we will release them for external validation. We extended JavaMOP to obtain traces (§3.3). Our changes do not modify instrumentation code (JavaMOP is very modular), and is not responsible for the high instrumentation costs that we observe.

Limitations. We only study RV overheads, not any other potential hindrances to its use in practice. Usability of RV and its spec languages is a subject of other research [86, 101, 132]. We do not

re-litigate spec quality [84, 86]: RV finds 10,733 violations during our experiments, but inspecting them is out of scope of this paper.

8 Related Work

RV during Software Testing. The potential to use RV during testing was previously recognized [6, 7], and is often mentioned in RV papers. But, only with relatively recent advancements in RV technology that enabled efficient simultaneous monitoring of multiple specs [94] did researchers start focusing on problems that arise when RV is used during testing in modern software development environments. Legunsen et al. demonstrated that RV amplifies the bug-finding ability of test suites [84, 86] and that focusing RV on code changes could help scale RV better as software evolves [87, 89, 138]. Our study is complementary and orthogonal: we perform a first-principles examination of the prevalence, usefulness, and breakdowns of RV overhead during testing.

Legunsen et al. [84, 86] show that RV lacks high-quality specs and generates many false alarms during testing. So, Miranda et al. [101] use machine learning to rank violations in order of likelihood of being true bugs. These works focus on human time for inspecting violations, but our focus is on runtime overheads. Some works integrate RV directly into unit testing frameworks, e.g., for Python [116] and for JUnit [39]; future work can evaluate them.

RV Research More Broadly. There has been decades of active research on RV, and several surveys, introductions, and competitions exist [14–16, 44–46, 90, 114]. Beyond integration with testing, some major directions in RV research (1) develop monitoring algorithms for spec languages [19, 21, 63, 64, 68, 80, 99, 100, 118, 121, 133], (2) theoretically analyze what properties are monitorable [1, 2, 20, 40, 43, 47, 60, 65, 117, 120], (3) develop techniques, algorithms, and data structures to speed up RV [27, 28, 32, 38, 41, 75, 77, 94, 111, 112, 135], (4) develop RV for different application domains [17, 37, 38, 48–51, 53, 71, 122, 134, 136], (5) investigate other styles of RV that differ from JavaMOP's [3, 10–12, 18, 30, 35, 36, 42, 61, 70, 96, 106, 113], and (6) develop frameworks and tools [5, 25, 33, 62, 76, 78, 138].

Our study is enabled by these advancements, but our goal is different: we seek to evaluate and understand the overheads of a popular style of RV during testing today. Our work hopefully encourages other researchers to perform similar evaluations for other styles and application domains of RV. Some RV techniques probabilistically sample monitors and events [5, 29] in long-running production environments. The degree of repetition among traces that we found suggests that future work could investigate sampling approaches during testing as well.

Other Studies of RV during Testing. Javed and Binder [73] evaluate JavaMOP and two other RV tools on 1,775 Maven open-source projects. But, they monitor one spec at a time and only use two specs. They do not analyze RV overhead in depth as we do, and their evaluation setting is not realistic: one would have to repeatedly run all tests for each spec. So, using their setting we would have seen at least 160x overhead per project (since we monitor 160 specs). On the flip side, they evaluate memory overhead, which is not a concern for us: tests in modern open-source projects are typically not run in memory-constrained environments. Although Javed and Binder evaluate more projects than we do, the product of specs and projects in this paper (1,544 x 160) is larger than theirs (1,775 x 2). Unlike theirs, our study also evaluates multiple versions of some projects and develops proofs-of-concept for reducing RV overhead.

Instrumentation for RV. Cassar et al. [31] survey instrumentation strategies in RV. Navabpour et al. [104] propose an instrumentation approach for a sampling-based RV technique. Bodden et al. [26] propose an approach to lower RV overhead in deployment by having many users run partially instrumented code. Marek et al. [95] develop a domain-specific language for instrumentation that incurs less overhead in a non-RV setting than AspectJ, but the overall instrumentation costs remain high. None of these prior works concern testing and they do not quantify, at scale, the proportion of RV overhead that is due to instrumentation. But, it may be possible for future work to learn from these works to reduce instrumentation overheads during RV of tests.

9 Conclusions and Future Work

RV's high runtime overhead is widely seen as a major hindrance to its adoption. Our large-scale and in-depth study quantifies and analyzes those overheads during testing, in terms of their prevalence, usefulness for bug finding, and components. Among other things, we find that the cost of instrumentation (not monitoring) is responsible for RV's overheads in most evaluated projects, and that monitoring effort is repetitively wasted in a manner that is not useful for finding bugs. Further, we investigate proofs of concept for reducing these overheads in the short term, and suggest longer-term future directions.

Data Availability

Our accompanying artifact contains an appendix with additional plots, raw data from our experiments, and our experimental infrastructure (see link at bottom of Section 1).

Acknowledgments

We thank Marcelo d'Amorim, Saikat Dutta, Alan Han, Pengyue Jiang, Yu Liu, Steven Long, Valeria Marquez, Sasa Misailovic, Ayaka Yorihiro, and the anonymous reviewers for their help, comments, and 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 and CCF-2319473.

References

[1] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. 2019. Adventures in monitorability: from branching to linear time and back again. *PACMPL* 3, POPL (2019).
[2] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. 2019. An operational guide to monitorability. In *SEFM*.
[3] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. 2005. Adding Trace Matching with Free Variables to AspectJ. In *OOPSLA*.
[4] Apache Software Foundation 2024. Apache Maven. http://maven.apache.org/.
[5] Matthew Arnold, Martin Vechev, and Eran Yahav. 2008. QVM: An Efficient Runtime for Detecting Defects in Deployed Systems. In *OOPSLA*.
[6] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Roşu, Koushik Sen, Willem Visser, et al. 2005. Combining test case generation and runtime verification. *TCS* 336, 2-3 (2005).
[7] Cyrille Artho, Doron Drusinksy, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Grigore Roşu, and Willem Visser. 2003. Experiments with test case generation and runtime analysis. In *Abstract State Machines*.

[8] AspectJ Guide 2005. Chapter 5. Load-Time Weaving. https://eclipse.dev/aspectj/doc/released/devguide/ltw.html.
[9] Async-Profiler Team 2024. Sampling CPU and HEAP profiler for Java. https://github.com/async-profiler/async-profiler.
[10] Pavel Avgustinov, Julian Tibble, and Oege de Moor. 2007. Making trace monitors feasible. In *OOPSLA*.
[11] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. 2004. Rule-Based Runtime Verification. In *VMCAI*.
[12] Howard Barringer, David Rydeheard, and Klaus Havelund. 2010. Rule Systems for Run-time Monitoring: From Eagle to RuleR. *Journal of Logic and Computation* 20, 3 (2010).
[13] Ezio Bartocci, Borzoo Bonakdarpour, and Yliès Falcone. 2014. First International Competition on Software for Runtime Verification. In *RV*.
[14] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Roşu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. 2019. First international Competition on Runtime Verification: Rules, benchmarks, tools, and final results. *IJSTTT* 21, 1 (2019).
[15] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on Runtime Verification*.
[16] Ezio Bartocci, Yliès Falcone, and Giles Reger. 2019. International Competition on Runtime Verification. In *TACAS*.
[17] David Basin, Felix Klaedtke, and Eugen Zălinescu. 2015. Failure-aware runtime verification of distributed systems. In *FSTTCS*.
[18] David Basin, Felix Klaedtke, and Eugen Zălinescu. 2017. Runtime verification of temporal properties over out-of-order data streams. In *CAV*.
[19] Omar Bataineh, David S Rosenblum, and Mark Reynolds. 2019. Efficient decentralized LTL monitoring framework using tableau technique. *TECS* 18, 5s (2019).
[20] Andreas Bauer. 2010. Monitorability of Omega-regular languages. *arXiv preprint arXiv:1006.3638* (2010).
[21] Andreas Bauer and Ylies Falcone. 2012. Decentralised LTL monitoring. In *FM*.
[22] Bazel 2024. Bazel Home Page. https://bazel.build.
[23] Jon Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *ICSE*.
[24] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*.
[25] Eric Bodden. 2011. MOPBox: A Library Approach to Runtime Verification. In *RV*.
[26] Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem. 2007. Collaborative Runtime Verification with Tracematches. In *RV*.
[27] Eric Bodden, Laurie Hendren, and Ondrej Lhoták. 2007. A Staged Static Program Analysis to Improve the Performance of Runtime Monitoring. In *ECOOP*.
[28] Eric Bodden, Patrick Lam, and Laurie Hendren. 2008. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-time. In *FSE*.
[29] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. 2013. Time-triggered runtime verification. In *FMSD*, Vol. 43.
[30] Glenn Bruns and Patrice Godefroid. 2001. Temporal logic query checking. In *LICS*.
[31] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. 2017. A survey of runtime monitoring instrumentation techniques. *arXiv preprint arXiv:1708.07229* (2017).
[32] Feng Chen, Patrick O'Neil Meredith, Dongyun Jin, and Grigore Roşu. 2009. Efficient formalism-independent monitoring of parametric properties. In *ASE*.
[33] Feng Chen and Grigore Roşu. 2003. Towards Monitoring-Oriented Programming: A paradigm combining specification and implementation. In *RV*.
[34] Feng Chen and Grigore Roşu. 2009. Parametric trace slicing and monitoring. In *TACAS*.
[35] Marcelo d'Amorim and Klaus Havelund. 2005. Event-based runtime verification of Java programs. In *WODA*.
[36] Ben d'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. LOLA: runtime monitoring of synchronous systems. In *Temporal Representation and Reasoning*.
[37] Luis Miguel Danielsson and César Sánchez. 2019. Decentralized stream runtime verification. In *RV*.
[38] Normann Decker, Jannis Harder, Torben Scheffel, Malte Schmitz, and Daniel Thoma. 2016. Runtime Monitoring with Union-Find Structures. In *TACAS*.
[39] Normann Decker, Martin Leucker, and Daniel Thoma. 2013. jUnit RV–adding runtime verification to jUnit. In *FM*.
[40] Volker Diekert and Martin Leucker. 2014. Topology, monitorable properties and runtime verification. *TCS* 537 (2014).
[41] Matthew B. Dwyer, Rahul Purandare, and Suzette Person. 2010. Runtime Verification in Context: Can Optimizing Error Detection Improve Fault Diagnosis?. In *RV*.
[42] U. Erlingsson and F. B. Schneider. 2000. IRM enforcement of Java stack inspection. In *IEEE S&P*.
[43] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. 2012. What can you verify and enforce at runtime? *IJSTTT* 14 (2012).
[44] Yliès Falcone, Klaus Havelund, and Giles Reger. 2013. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems*.
[45] Yliès Falcone, Srđan Krstić, Giles Reger, and Dmitriy Traytel. 2018. A Taxonomy for Classifying Runtime Verification Tools. In *RV*.
[46] Yliès Falcone, Dejan Ničković, Giles Reger, and Daniel Thoma. 2015. Second International Competition on Runtime Verification. In *RV*.
[47] Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. 2017. A foundation for runtime monitoring. In *RV*.
[48] Adrian Francalanza, Jorge A Pérez, and César Sánchez. 2018. Runtime verification for decentralised and distributed systems. *Lectures on Runtime Verification* (2018).
[49] Ritam Ganguly, Anik Momtaz, and Borzoo Bonakdarpour. 2021. Distributed runtime verification under partial synchrony. In *OPODIS*.
[50] Ritam Ganguly, Yingjie Xue, Aaron Jonckheere, Parker Ljung, Benjamin Schornstein, Borzoo Bonakdarpour, and Maurice Herlihy. 2022. Distributed Runtime Verification of Metric Temporal Properties for Cross-Chain Protocols. In *ICDCS*.
[51] Ritam Ganguly, Yingjie Xue, Aaron Jonckheere, Parker Ljung, Benjamin Schornstein, Borzoo Bonakdarpour, and Maurice Herlihy. 2024. Distributed runtime verification of metric temporal properties. *J. Parallel and Distrib. Comput.* 185 (2024).
[52] Google Testing Blog 2008. TotT: Avoiding Flakey Tests. http://goo.gl/vHE47r.
[53] Felipe Gorostiaga and César Sánchez. 2018. Striver: Stream runtime verification for real-time event-streams. In *RV*.
[54] Gradle 2024. Gradle Home Page. https://gradle.org.
[55] Grammatech 2024. ARTCAT: Autonomic Response To Cyber-Attack. https://grammatech.github.io/prj/artcat.
[56] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Workshop on Workload Characterization*.
[57] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. 2016. NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications. In *FSE Demo*.
[58] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating regression test selection opportunities in a very large open-source ecosystem. In *ISSRE*.
[59] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression Test Selection for Java Software. In *OOPSLA*.
[60] Klaud Havelund and Doron Peled. 2023. Monitorability for Runtime Verification. In *RV*.
[61] Klaus Havelund, Doron Peled, and Dogan Ulus. 2017. First order temporal logic monitoring with BDDs. In *FMSD*.
[62] Klaus Havelund and Grigore Roşu. 2001. Monitoring Java Programs with Java PathExplorer. In *RV*.
[63] Klaus Havelund and Grigore Roşu. 2001. Monitoring Programs Using Rewriting. In *ASE*.
[64] Klaus Havelund and Grigore Roşu. 2002. Synthesizing Monitors for Safety Properties. In *TACAS*.
[65] Thomas A Henzinger and N Ege Saraç. 2020. Monitorability under assumptions. In *RV*.
[66] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *FSE*.
[67] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *ASE*.
[68] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. 2014. Online monitoring of metric temporal logic. In *RV*.
[69] How good are Specs? 2016. Supplementary Material for Paper [86]. https://www.cs.cornell.edu/~legunsen/spec-eval.
[70] Samuel Huang and Rance Cleaveland. 2022. Temporal-logic query checking over finite data streams. *IJSTTT* 24, 3 (2022).
[71] Soha Hussein, Patrick Meredith, and Grigore Roşu. 2012. Security-Policy Monitoring and Enforcement with JavaMOP. In *PLAS*.
[72] JavaMOP 2024. JavaMOP Team. https://github.com/runtimeverification/javamop.
[73] O. Javed and W. Binder. 2018. Large-Scale Evaluation of the Efficiency of Runtime-Verification Tools in the Wild. In *APSEC*.
[74] JetBrains 2024. How IntelliJ IDEA profiler works. https://www.jetbrains.com/help/idea/cpu-and-allocation-profiling-basic-concepts.html.
[75] Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Roşu. 2011. Garbage Collection for Monitoring Parametric Properties. In *PLDI*.

[76] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. 2012. JavaMOP: Efficient Parametric Runtime Monitoring Framework. In *ICSE Demo*.

[77] Dongyun Jin, Patrick O'Neil Meredith, and Grigore Roşu. 2012. *Scalable Parametric Runtime Monitoring*. Technical Report. Computer Science Dept., UIUC.

[78] Murat Karaorman and Jay Freeman. 2004. jMonitor: Java runtime event specification and monitoring library. In *RV*.

[79] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. 2001. An overview of AspectJ. In *ECOOP*.

[80] Moonjoo Kim, Mahesh Viswanathan, Hanene Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. 1999. Formally specified monitoring of temporal properties. In *ECRTS*.

[81] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. 1995. Class firewall, test order, and regression testing of object-oriented programs. *JOOP* 8, 2 (1995).

[82] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST*.

[83] Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, and Grigore Roşu. 2012. *Towards Categorizing and Formalizing the JDK API*. Technical Report. Computer Science Dept., UIUC.

[84] 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).

[85] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *FSE*.

[86] 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*.

[87] Owolabi Legunsen, Darko Marinov, and Grigore Roşu. 2015. Evolution-aware monitoring-oriented programming. In *ICSE NIER*.

[88] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic Regression Test Selection. In *ASE*.

[89] Owolabi Legunsen, Yi Zhang, Milica Hadzi-Tanovic, Grigore Roşu, and Darko Marinov. 2019. Techniques for Evolution-Aware Runtime Verification. In *ICST*.

[90] Martin Leucker and Christian Schallhart. 2007. A brief account of runtime verification. In *FLACOS*.

[91] Hareton K.N. Leung and Lee White. 1990. A study of integration testing and software regression at the integration level. In *ICSM*.

[92] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. 2023. More precise regression test selection via reasoning about semantics-modifying changes. In *ISSTA*.

[93] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE*.

[94] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu. 2014. RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties. In *RV*.

[95] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A domain-specific language for bytecode instrumentation. In *AOSD*.

[96] Michael Martin, Benjamin Livshits, and Monica S Lam. 2005. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*.

[97] Matplotlib Team 2024. Matplotlib: Visualization with Python. https://matplotlib.org.

[98] Atif M. Memon and Myra B. Cohen. 2013. Automated testing of GUI applications: models, tools, and controlling flakiness. In *ICSE*.

[99] P.O. Meredith, Dongyun Jin, Feng Chen, and G. Roşu. 2008. Efficient Monitoring of Parametric Context-Free Patterns. In *ASE*.

[100] Patrick Meredith and Grigore Roşu. 2013. Efficient Parametric Runtime Verification with Deterministic String Rewriting. In *ASE*.

[101] Breno Miranda, Igor Lima, Owolabi Legunsen, and Marcelo d'Amorim. 2020. Prioritizing Runtime Verification Violations. In *ICST*.

[102] Miranda, João 2014. How Etsy Deploys More Than 50 Times a Day. https://www.infoq.com/news/2014/03/etsy-deploy-50-times-a-day.

[103] Mountainminds GmbH & Co. KG and Contributors 2017. JaCoCo Java Code Coverage Library. http://www.eclemma.org/jacoco/.

[104] Samaneh Navabpour, Chun Wah Wallace Wu, Borzoo Bonakdarpour, and Sebastian Fischmeister. 2011. Efficient Techniques for Near-Optimal Instrumentation in Time-Triggered Runtime Verification. In *RV*.

[105] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. In *ICSE*.

[106] Vinit A Ogale and Vijay K Garg. 2007. Detecting temporal logic predicates on distributed computations. In *DISC*.

[107] Peter W O'Hearn. 2018. Continuous reasoning: Scaling the impact of formal methods. In *LICS*.

[108] Oracle 2024. Collections API. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collections.html#synchronizedCollection(java.util.Collection).

[109] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *FSE*.

[110] Fabio Palomba and Andy Zaidman. 2017. Does refactoring of test smells induce fixing flaky tests?. In *ICSME*.

[111] Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. 2010. Monitor Optimization via Stutter-equivalent Loop Transformation. In *OOPSLA*.

[112] Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. 2013. Optimizing Monitoring of Finite State Properties Through Monitor Compaction. In *ISSTA*.

[113] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. 2015. MarQ: Monitoring at Runtime with QEA. In *TACAS*.

[114] Giles Reger, Sylvain Hallé, and Yliès Falcone. 2016. Third International Competition on Runtime Verification. In *RV*.

[115] David Georg Reichelt, Lubomír Bulej, Reiner Jung, and André van Hoorn. 2024. Overhead Comparison of Instrumentation Frameworks. In *ICPE*.

[116] Adam Renberg. 2014. *Test-inspired runtime verification: Using a unit test-like specification syntax for runtime verification*. Master's thesis. KTH, Sweden.

[117] Grigore Rosu. 2012. On safety properties and their monitoring. *Scientific Annals of Computer Science* 22, 2 (2012).

[118] Grigore Roşu and Saddek Bensalem. 2006. Allen Linear (Interval) Temporal Logic – Translation to LTL and Monitor Synthesis. In *CAV*.

[119] Runtime Verification Inc. 2024. Performance regression that we find in JavaMOP. https://github.com/runtimeverification/rv-monitor/commit/884f9622f.

[120] Fred B. Schneider. 2000. Enforceable Security Policies. *TISSEC* 3, 1 (2000).

[121] Koushik Sen, Grigore Roşu, and Gul Agha. 2003. Generating Optimal Linear Temporal Logic Monitors by Coinduction. In *Advances in Computing Science*.

[122] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. 2004. Efficient decentralized monitoring of safety in distributed systems. In *ICSE*.

[123] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications. In *ICST*.

[124] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-Aware Static Regression Test Selection. In *OOPSLA*.

[125] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *FSE*.

[126] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *ISSRE*.

[127] Chukri Soueidi and Yliès Falcone. 2023. Bridging the Gap: A Focused DSL for RV-Oriented Instrumentation with BISM. In *RV*.

[128] Chukri Soueidi and Yliès Falcone. 2023. Instrumentation for RV: From Basic Monitoring to Advanced Use Cases. In *RV*.

[129] Chukri Soueidi, Yliès Falcone, and Sylvain Hallé. 2023. Dynamic Program Analysis with Flexible Instrumentation and Complex Event Processing. In *ISSRE*.

[130] Chukri Soueidi, Marius Monnier, and Yliès Falcone. 2023. Efficient and expressive bytecode-level instrumentation for Java programs. *IJSTTT* 25, 4 (2023).

[131] STARTS Team 2024. STARTS—A tool for STAtic Regression Test Selection. https://github.com/TestingResearchIllinois/starts.

[132] Leopoldo Teixeira, Breno Miranda, Henrique Rebêlo, and Marcelo d'Amorim. 2021. Demystifying the challenges of formally specifying API properties for runtime verification. In *ICST*.

[133] Prasanna Thati and Grigore Rosu. 2004. Monitoring Algorithms for Metric Temporal Logic Specifications. In *RV*.

[134] Adriano Torres, Pedro Costa, Luis Amaral, Jonata Pastro, Rodrigo Bonifácio, Marcelo d'Amorim, Owolabi Legunsen, Eric Bodden, and Edna Dias Canedo. 2023. Runtime Verification of Crypto APIs: An Empirical Study. *TSE* 49, 10 (2023).

[135] Chun Wah Wallace Wu, Deepak Kumar, Borzoo Bonakdarpour, and Sebastian Fischmeister. 2013. Reducing Monitoring Overhead by Integrating Event- and Time-Triggered Techniques. In *RV*.

[136] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. 2020. Aragog: Scalable runtime verification of shardable networked systems. In *OSDI*.

[137] S. Yoo and M. Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *STVR* 22, 2 (2012).

[138] Ayaka Yorihiro, Pengyue Jiang, Valeria Marques, Benjamin Carleton, and Owolabi Legunsen. 2023. eMOP: A Maven Plugin for Evolution-Aware Runtime Verification. In *RV*.

[139] Jiyang Zhang, Yu Liu, Milos Gligoric, Owolabi Legunsen, and August Shi. 2022. Comparing and combining analysis-based and learning-based regression test selection. In *ASE*.

[140] Lingming Zhang. 2018. Hybrid Regression Test Selection. In *ICSE*.

[141] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *ICSE*.

Received 2024-04-12; accepted 2024-07-03