2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Instrumentation-Driven Evolution-Aware Runtime Verification

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

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

***Abstract*—Runtime verification (RV) found hundreds of bugs by monitoring passing tests against formal specifications (specs). RV first instruments a program to obtain relevant events, e.g., method calls, to monitor. A hindrance to RV adoption, especially in continuous integration, is its high overhead. So, prior work proposed *spec-driven* evolution-aware techniques to speed up RV. They use complex analysis to re-monitor a subset of specs related to code impacted by changes. But, these techniques assume that RV overhead is dominated by monitoring time, and their designs often sacrifice safety (ability to find all new violations) for speed.**

We present iMOP, the first *instrumentation-driven* evolution-aware RV framework. iMOP leverages a recent observation that RV overhead during testing is often dominated by instrumentation, not monitoring. iMOP embodies a family of 14 techniques that aim to safely speed up RV by simply re-instrumenting only changed code. Instrumentation from the old revision is re-used for unchanged code, and all specs are re-monitored in the new revision. We implement iMOP as a Maven plugin and evaluate it on 2,028 revisions of 66 projects, using 160 specs of correct JDK API usage. iMOP is safe by design. It is up to 40.2x faster than re-running RV from scratch after each change, and 17.8x and 6.7x faster than safe and unsafe spec-driven techniques, respectively. iMOP is faster than just applying regression test selection to RV.

I. Introduction

Runtime verification (RV) [43], [60], [73] monitors executions of possibly buggy programs against formal specifications (specs) of correct or safe behavior. Practically, RV checks if *traces*—sequences of program actions, such as method calls or field accesses—satisfy the specs. To do so, RV first instruments a program to signal relevant program actions as runtime *events*. Then, *monitors* (usually automata) are dynamically synthesized [61] from the specs to check the traces. A monitor raises a *violation* if a trace does not satisfy a spec. Each spec definition includes a *handler*—user-provided, e.g., error-recovery, code to run if violations occur.

Thanks to decades of research [5], [10], [14], [21], [22], [31], [36], [46], [59], [70], [84], [95], [100], RV is now being used to monitor deployed software [6], [32]–[34], [108]. Doing so is appealing: preemptive violation detection and sound error recovery can make deployed software always satisfy the specs [27]. So, most RV research targets deployment-time RV.

Our work is on RV during testing, *before* deployment. Researchers showed that using RV during testing amplifies the bug-finding ability of tests. They used handlers that print violations during RV of passing tests against specs of correct JDK API usage. By inspecting those violations, these researchers found hundreds of confirmed bugs that testing alone missed in many projects [80], [82], [94]. Increased bug-finding ability occurs because specs provide additional test oracles that may be hard to express as test assertions. (§II has examples.)

Despite its bug-finding benefits, a main hindrance to RV adoption during testing, especially in continuous integration, is that it incurs high runtime overheads. Those overheads persist despite advances on algorithms [26], [29], [66] and data structures [28], [87], [93] for speeding up RV.

Prior work proposed 12 evolution-aware techniques to speed up RV during testing of evolving software by focusing monitoring effort on code impacted by changes [77]–[79], [126]. These techniques are *spec-driven*, use complex program analysis, and work in three main steps. Given new and old code revisions and a set of specs (multiple specs are monitored simultaneously): (i) find code impacted by changes, (ii) find a subset of affected specs that are related to impacted code, and (iii) re-monitor only affected specs in the new revision.

Spec-driven evolution-aware RV techniques have two drawbacks. First, they assume that RV overhead is dominated by monitoring time to signal events, create monitors, process events, etc., [77]. So, they may provide sub-optimal speedups if RV overhead is not dominated by monitoring. Second, 10 (of 12) of them provide good speedups but they are designed to be unsafe, i.e., they may miss some new violations after code changes [78]. The other two are safe by design, but slower.

We present iMOP, the first framework for an alternate, *instrumentation-driven* approach to evolution-aware RV. (MOP: Monitoring Oriented Programming [27], the RV style that we use.) iMOP is inspired by a recent study [52], which found that over 51% of RV overhead during testing in 1,322 of 1,544 projects is due to instrumentation. In deployment-time RV, instrumentation is a one-time start-up cost. But, during testing, these instrumentation costs should be reduced so that they are not entirely re-incurred after every code change.

The idea behind iMOP is that amortizing instrumentation costs across code revisions can speedup RV during testing [52]. iMOP embodies a family of 14 techniques that aim to speed up RV by simply re-instrumenting only changed code. The old revision's instrumentation is re-used for unchanged code, and all specs are re-monitored in the new revision. That way, iMOP reduces costs of unnecessarily re-instrumenting unchanged code in the new revision.

iMOP first identifies changed code, then it re-instruments only changed code using all specs. Lastly, an RV tool monitors the instrumented code. iMOP techniques differ in (i) the granularity level at which they find changes, (ii) whether they are specific to an instrumentation framework or work with any framework, and (iii) if they re-instrument all changed code in 3rd-party libraries or only changed library code that is used.

iMOP techniques are *agnostic* or *specific* to instrumentation frameworks. We propose 11 agnostic techniques and realize a 12th. The other two techniques are specific to AspectJ [71], which is used in JavaMOP [63], [67], the RV tool in this paper; we are the first to evaluate them for RV. Separately, eight techniques are *usage-unaware*: they re-instrument changed library classes even if the monitored program does not use them. The other six techniques are *usage-aware* and aim to avoid re-instrumenting changed but unused library classes.

We implement iMOP as a Maven plugin. Our evaluation of iMOP involves using it to monitor the execution of 17,546 developer written tests in 2,028 revisions of 66 projects, using 160 specs of correct JDK API usage protocols.

iMOP is up to 40.2x (mean: 10x) faster than *full RV* (re-running RV from scratch after each change), and up to 17.8x (mean: 5.8x) and 6.7x (mean: 2.4x) faster than the fastest safe-by-design and fastest *unsafe*-by-design spec-driven techniques, respectively. Across all projects, iMOP saves up to 40.4 hours compared to full RV, and 24.2 hours and 6.9 hours, compared to safe and unsafe spec-driven techniques, respectively.

To evaluate safety, we compare sets of new violations from iMOP, full RV, and spec-driven techniques. Usage-unaware iMOP techniques are as safe as full RV; usage-aware ones are at least as safe as safe spec-driven techniques.

Lastly, we compare and combine iMOP with regression test selection (RTS), which speeds up regression testing by only re-running tests impacted by changes [125]. We integrate two RTS tools, Ekstazi [48], [49] and STARTS [81], [83] with iMOP and compare speedups from combining RTS with (i) full RV, (ii) safe and unsafe spec-driven techniques, and (iii) the best iMOP technique in 10 projects. iMOP is faster than just applying RTS to full RV, but iMOP plus RTS provides more speedup than iMOP alone on some projects.

This paper makes the following contributions:

⋆ **Framework.** iMOP is the first instrumentation-driven evolution-aware RV framework, and realization of the idea to speed up RV by amortizing instrumentation costs [52].

⋆ **Techniques.** We propose 11 of iMOP's 14 techniques and we are the first to evaluate two of the other three for RV.

⋆ **Implementation.** We implement iMOP for Java as a Maven plugin that integrates easily with open-source projects.

⋆ **Combination.** We compare and combine RTS with instrumentation-driven evolution-aware RV techniques.

⋆ **Evaluation.** We conduct the largest evaluation of evolution-aware RV and the first evaluation of iMOP's safety and applicability to multiple instrumentation frameworks.

Our plugin, scripts, and experimental data are available at https://github.com/SoftEngResearch/imop.

```
1 StringTokenizer_HasMoreElements(StringTokenizer i) {
2 event hasnexttrue after(StringTokenizer st) returning (
      boolean b):
3  ( call(boolean StringTokenizer.hasMoreTokens()) || call(
      boolean StringTokenizer.hasMoreElements())) && target
      (st) && condition(b){}
4 event next before(StringTokenizer st):
5  ( call(* StringTokenizer.nextToken()) || call(*
      StringTokenizer.nextElement())) && target(st){}
6 ltl: [](next => (*) hasnexttrue)
7 @violation {/*print violation*/)} }
```

Fig. 1: STHM Spec, written in an AspectJ-based DSL.

```
1 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 syncMakeI after(Collection c)returning(Iterator i):
6  call(* Collection+.iterator()) && target(c) && condition(
      Thread.holdsLock(c)) {}
7 event asyncMakeI after(Collection c)returning(Iterator i):
8  call(* Collection+.iterator()) && target(c) && condition
      (!Thread.holdsLock(c)) {}
9 event useI before(Iterator i):
10 call(* Iterator.*(..)) && target(i) && condition(!Thread.
      holdsLock(this.c)) {}
11 ere : (sync asyncMakeI) | (sync syncMakeI useI)
12 @match {/*print violation*/} }
```

Fig. 2: CSC Spec, written in an AspectJ-based DSL.

II. Examples and Background

Specs and how RV monitors them. To monitor the StringTokenizer_HasMoreElements (STHM) spec in Figure 1, RV first instruments the monitored program based on events defined on lines 2–5. Each definition includes (i) an event name like hasnexttrue on line 2, that is used to specify properties; (ii) relevant program actions for each event, e.g., hasnexttrue captures calls to hasMoreTokens() or hasMoreElement() on a StringTokenizer st that return true, while next captures calls to st.nextToken() or st.nextElement(); and (iii) whether to signal events before or after these calls.

At runtime, RV synthesizes monitors to process events that are signaled from the instrumented code. STHM's monitors check if traces satisfy the linear temporal logic (LTL) safety property on line 6: "always, a next event on st implies that the previous event on st was hasnexttrue". If a trace does not satisfy this property, the handler on line 7 is invoked. Handlers can be any user-provided code. But, for RV usage during testing, we print a message to aid debugging. STHM helped find bugs in code that can crash by calling nextToken() or nextElement() on an empty st [80], [82]. The monitored tests always pass because all input st were not empty, but inspecting STHM violations helped find the bugs.

We need at least two specs to explain spec-driven techniques. So, Figure 2 shows the SynchronizedCollection (CSC) spec, whose safety property on line 11 is violated if a trace *matches* one of two cases. (i) A synchronized collection c is created (sync, lines 3–4), but an Iterator i is then created from code that does not hold the lock on c (asyncMakeI, lines 7–8). (ii) After a sync event, one correctly creates i from code that holds the lock on c (syncMakeI, lines 5–6), but then later uses i in code that does not hold the lock on c (useI, lines 9–10). Code producing such traces can be non-deterministic [30]. CSC also helped find several bugs [80], [82].

```
1  public class A {
2   public static String a(List<String> list) {
3    Collection<String> c =
4     Collections.synchronizedCollection(list);/*INSTR: CSC.sync*/
5    Iterator<String> i = c.iterator();/*INSTR: CSC.asyncMakeI*/
6    boolean b = /*INSTR: CSC.useI*/!i.hasNext();
7    return b ? null : /*INSTR: CSC.useI*/Lib1.lower(i.next()); }
8  }
9  class ATest {
10  @Test public void testA() {
11    List<String> l = Arrays.asList(new String[]{"FOO"});
12    assertEquals("foo", A.a(l)); }
13 }
14 public class B {
15  public static String b(String s) {
16  -String out = "none";
17  +String out = "nil";
18  StringTokenizer t = new StringTokenizer(s);
19  if (t.hasMoreTokens()/*INSTR: STHM.hasnexttrue*/) {
20    /*INSTR: STHM.next*/out = t.nextToken();
21  } Lib2.process(out); return out; }
22 }
23 public class BTest {
24  @Test public void testB(){assertEquals("f", B.b("f b"));}}
```

Fig. 3: Example of instrumented code and unit tests.

Instrumentation. Figure 3 shows old and new revisions of example code (classes `A` and `B`) and tests (classes `ATest` and `BTest`). The code uses library classes `Lib1` (line 7) and `Lib2` (line 21), which have call sites for `CSC` and `STHM` events, respectively, that we elide. The new revision assigns "nil" to `out` instead of "none". Before monitoring the tests against `CSC` and `STHM` in both revisions, full RV first instruments the code based on `CSC` and `STHM` event definitions (violet comments).

Spec-driven evolution-aware RV techniques. The main idea in spec-driven techniques [78] is that specs that are not affected by changes need not be re-monitored in the new revision. Doing so yields the same monitoring outcome as in the old revision for those specs (assuming deterministic tests). So, in Figure 3, the class-level change-impact analysis used by spec-driven techniques first finds only `B` and `BTest` as impacted by the change. `CSC` is unrelated to `B` and `BTest`, so `STHM` is the only affected spec that is re-monitored in the new revision. If RV overhead is dominated by monitoring `CSC`, then RV can be much faster in the new revision. Fast spec-driven techniques are unsafe by design; their less conservative change-impact analysis can make them miss affected specs. §IV gives more details on the two spec-driven techniques in this paper.

Instrumentation-driven evolution-aware RV techniques. In Figure 3, suppose full RV instruments only `CSC` in `A` and `Lib1` and only `STHM` in `B` and `Lib2`. Also, assume that the only change is the one shown. Then, our iMOP techniques aim to only re-instrument `B` after the change. So, only 25% of classes are re-instrumented. By re-monitoring all specs in the new revision, these iMOP techniques are safe by design.

III. iMOP

A. Overview

Figure 4 is a high-level overview of iMOP; it abstracts away individual techniques' details. Conceptually, iMOP's inputs are (i) old and new revisions of the code under test (CUT), including source code and tests; (ii) a list of required 3rd-party library (e.g., jar) paths and their declared versions (e.g., 4.5.2) that the CUT uses in both revisions; (iii) checksums of `.class`

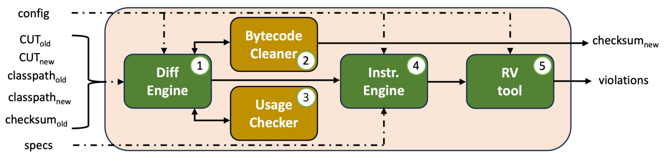

Fig. 4: iMOP's high-level workflow.

files in the old revision; (iv) the specs; and (iv) a config file with the iMOP technique to use, whether to ignore changes to debug information (line numbers, comments, space, etc.), thread count to use, whether to check if the CUT uses changed library classes, and the instrumentation framework to use.

iMOP works in five main steps. The Diff Engine (step ①) takes both revisions of the CUT and the library paths, and finds `.class` files that changed. If so configured, the Diff Engine can use Bytecode Cleaner (step ②) to find only changed `.class` files where non-debug information changed, and compute their checksums for the next revision. Files where only debug information changed need not be re-instrumented, but reasoning about debug information incurs a cost [49]. If Bytecode Cleaner is used, changed classes are `.class` files whose bytecode checksums (after ignoring debug information) in the new revision differ from those in the old revision.

Diff Engine can also be configured to call Usage Checker (step ③), which checks if changed library classes are used by the CUT. Unused changed classes need not be re-instrumented. Doing so is wasteful: the new revision's monitoring outcome cannot depend on unused classes. But, checking usage incurs a cost. Instrumentation Engine (step ④) re-instruments (in sequence or in parallel) only Diff Engine's output using the configured framework (default: AspectJ). Instrumentation from the old revision is re-used for all other (unchanged) classes.

Finally, iMOP invokes an RV tool (step ⑤) to monitor the instrumented CUT and libraries and report any violation. To bootstrap, all CUT and library classes are treated as changed in the first revision. Note that Figure 4 is conceptual, e.g., the old and new revisions may not be explicit inputs (§III-B).

B. Techniques

Design rationale. Realizing the simple idea behind iMOP requires addressing the technical challenge of simultaneously (i) speeding up RV by finding and re-instrumenting (ideally) only changed code, (ii) preserving safety, and (iii) aiming for overheads that approach a lower bound of full RV minus instrumentation costs. Meeting all three goals requires carefully balancing the tradeoffs that they induce.

In brief, the tradeoff space is as follows. More precise change identification can reduce re-instrumented classes and help goal (i), but its analysis can be more costly and hurt goal (iii). Also, analyzing libraries is necessary for safety [78] and helps goal (ii), but doing so increases analysis cost and hurts goal (iii). Coarse-grained analysis of libraries, e.g., by treating a jar as changed if any of its contents changed, can be faster and help goals (ii) and (iii). But, such analysis can lead to unnecessary re-instrumentation, hurting goal (i), if whole-jar instrumentation is costly. Lastly, using more threads can speed up iMOP, but it requires more hardware resources.

TABLE I: IMOP techniques and the notation that we subsequently use for them. Table II explains the notation.

INSTRUMENTATION TARGET	NOTATION
Usage Unaware, Agnostic of Instrumentation Framework (§III-B1)	
Whole changed jars, online, sequential	$\mathbf{UJ}^{s}_{o}$
All class files in changed jars, online, parallel	$\mathbf{UJ}^{p}_{o}$
Changed class files in changed jars, online, sequential	$\mathbf{UC}^{s}_{o}$
Changed class files in changed jars, online, parallel	$\mathbf{UC}^{p}_{o}$
Changed bytecode in changed jars, online, sequential	$\mathbf{UB}^{s}_{o}$
Changed bytecode in changed jars, online, parallel	$\mathbf{UB}^{p}_{o}$
Changed bytecode in changed jars, stored hashes, sequential	$\mathbf{UB}^{s}_{h}$
Changed bytecode in changed jars, stored hashes, parallel	$\mathbf{UB}^{p}_{h}$
Usage Aware, Agnostic of Instrumentation Framework (§III-B2)	
Changed bytecode used by CUT at runtime, sequential	$\mathbf{AB}^{sd}_{h}$
Changed bytecode used by CUT at runtime, parallel	$\mathbf{AB}^{pd}_{h}$
Changed bytecode reachable statically by CUT, sequential	$\mathbf{AB}^{ss}_{h}$
Changed bytecode reachable statically by CUT, parallel	$\mathbf{AB}^{ps}_{h}$
Usage Aware, Specific to AspectJ Instrumentation Framework (§III-B3)	
Compiler-determined changed class files, default loader	$\mathbf{ajc}^{def}$
Compiler-determined changed class files, shared loader	$\mathbf{ajc}^{one}$

Design justification. IMOP embodies a family of 14 simple techniques—11 of which we propose—at different points in the tradeoff space. Most of these techniques perform best in at least one project (§IV), suggesting that there is no one-size-fits-all technique that works for all projects, and justifying our design of IMOP to have several techniques.

Running example. We will use Figure 5 to illustrate how some IMOP techniques work. There, classes `C1`, `C2`, and `T` (a test class) are in the CUT, while classes `L1`–`L4` are in libraries. Edges show usage, e.g., the arrow from `C2` to `C1` means "`C2` uses `C1`". Purple-colored classes changed. `L1` is colored pink because its `.class` file changed but its bytecode did not. For any code change, full RV re-instruments all classes in Figure 5. IMOP aims to reduce wasted re-instrumentation.

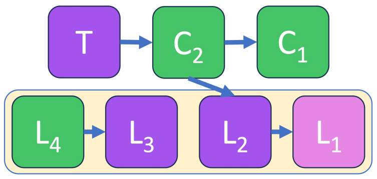

Fig. 5: Running example.

Design details. For ease of presentation, we organize IMOP's 14 techniques into three groups in Table I. These techniques primarily differ in how they handle libraries—they all re-instrument only changed CUT classes in sequence, which is fast: few CUT classes typically change at once [78]. Techniques that check for changed bytecode in changed jars also check for changed bytecode in CUT classes. We next discuss how each technique handles changes in 3rd-party libraries.

1) Usage-Unaware, Framework-Agnostic Techniques: Eight IMOP techniques are usage unaware (**U**); they re-instrument changed library code without incurring the cost to check if the CUT uses such changed code. These techniques are also agnostic, and can work with any instrumentation framework. Two of these techniques re-instrument an entire changed library (**J**). IMOP treats a library as changed if it (i) was not on the old revision's classpath; (ii) has different declared versions in both revisions; or (iii) has changed contents. One technique, $\mathbf{UJ}^{s}_{o}$, (Table II explains the notation) re-instruments changed libraries sequentially. The other technique, $\mathbf{UJ}^{p}_{o}$, re-instruments all changed library classes in parallel. In Figure 5, $\mathbf{UJ}^{s}_{o}$ and $\mathbf{UJ}^{p}_{o}$ will re-instrument classes `T`, `L1`, `L2`, `L3`, and `L4` because at least one library class changed. $\mathbf{UJ}^{s}_{o}$ and $\mathbf{UJ}^{p}_{o}$ can be fast if library changes are small or occur infrequently. Otherwise, re-instrumenting whole libraries can be expensive [52].

TABLE II: Notation key.

α	Meaning
First uppercase letter	
U	Usage Unaware
A	Usage Aware
Second uppercase letter	
J	Finds changed Jars
C	Finds changed Class files
B	Finds changed Bytecode
First superscript position	
s	Re-instruments sequentially
p	Re-instruments in parallel
Second superscript position	
d	Checks usage dynamically
s	Checks usage statically
Subscript	
o	Checks changes online
h	Checks changed hashes

To reduce wasted re-instrumentation in libraries, six usage-unaware IMOP techniques aim to re-instrument only changed library classes, without incurring the cost to reason about whether the CUT uses those classes. These techniques are as follows:

- $\mathbf{UC}^{s}_{o}$ uses `diff` to compare `.class` files in both versions of libraries and re-instruments only those that differ. In Figure 5, though the library changed, $\mathbf{UC}^{s}_{o}$ only re-instruments `T`, `L1`, `L2`, and `L3`; it does not re-instrument the unchanged `L4`.
- $\mathbf{UB}^{s}_{o}$ first cleans changed `.class` files to ignore their debug information, then it checks if the resulting bytecode was modified. Changing only debug information modifies `.class` files but not the bytecode that is run. Lastly, only modified `.class` files with changed bytecode are re-instrumented.
- $\mathbf{UB}^{s}_{h}$ is similar to $\mathbf{UB}^{s}_{o}$, but it pre-computes some steps in the old revision to reduce analysis time in the new revision. In the first run, $\mathbf{UB}^{s}_{h}$ stores a checksum of bytecode in each `.class` files on disk. Then, for each changed `.class` file in a new revision, $\mathbf{UB}^{s}_{h}$ computes and compares the checksum of its cleaned bytecode with the one it loads from disk. If the checksums differ, then $\mathbf{UB}^{s}_{h}$ re-instruments the `.class` file and updates the corresponding checksum on disk.

$\mathbf{UB}^{s}_{o}$ and $\mathbf{UB}^{s}_{h}$ only re-instrument `T`, `L2`, and `L3`; they do not re-instrument `L1`, whose `.class` file changed but its cleaned bytecode did not. $\mathbf{UC}^{s}_{o}$, $\mathbf{UB}^{s}_{o}$, and $\mathbf{UB}^{s}_{h}$ re-instrument all such `.class` files in sequence. Their parallel analogs—$\mathbf{UC}^{p}_{o}$, $\mathbf{UB}^{p}_{o}$, and $\mathbf{UB}^{p}_{h}$, respectively—use multiple threads.

2) Usage-Aware, Framework-Agnostic Techniques: Four agnostic techniques are usage aware (**A**): they aim to only re-instrument bytecode that changed *and* that the CUT uses. Such used `.class` files with changed bytecode can be fewer than those in §III-B1 (saving re-instrumentation time). But, to achieve end-to-end speedup, checking for usage must be fast.

- $\mathbf{AB}^{sd}_{h}$ finds used classes, `Loaded`, as those that the JVM loads. Then it re-instruments only `.class` files in `Loaded` whose cleaned bytecode changed. Using `Loaded` from an old revision to choose what to re-instrument in a new revision is unsafe: changes can cause more classes to be loaded in the new revision. So, $\mathbf{AB}^{sd}_{h}$ runs tests twice in the new revision: (i) run tests without RV to update `Loaded`; (ii) find, clean,

TABLE III: Features in each IMOP technique. ✓: feature is present. ✗: feature is absent. “n/a”: not applicable.

	$\mathbf{UJ}^{\mathsf{s}}_{\mathsf{o}}$	$\mathbf{UJ}^{\mathsf{p}}_{\mathsf{o}}$	$\mathbf{UC}^{\mathsf{s}}_{\mathsf{o}}$	$\mathbf{UC}^{\mathsf{p}}_{\mathsf{o}}$	$\mathbf{UB}^{\mathsf{s}}_{\mathsf{o}}$	$\mathbf{UB}^{\mathsf{p}}_{\mathsf{o}}$	$\mathbf{UB}^{\mathsf{s}}_{\mathsf{h}}$	$\mathbf{UB}^{\mathsf{p}}_{\mathsf{h}}$	$\mathbf{AB}^{\mathsf{sd}}_{\mathsf{h}}$	$\mathbf{AB}^{\mathsf{pd}}_{\mathsf{h}}$	$\mathbf{AB}^{\mathsf{ss}}_{\mathsf{h}}$	$\mathbf{AB}^{\mathsf{ps}}_{\mathsf{h}}$	$\mathbf{ajc}^{\mathsf{def}}$	$\mathbf{ajc}^{\mathsf{one}}$
Incremental CUT instrumentation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Incremental lib instrumentation	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Framework agnostic	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
One cache per loader	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	✓	✗
Usage aware	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
Check usage statically	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
Parallel re-instrumentation (lib)	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✗
Check bytecode difference	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
Check class file difference	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Store old hashes	n/a	n/a	n/a	n/a	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓

and re-instrument changed `.class` files in `Loaded` whose bytecode changed, and (iii) re-run tests with RV.

- $\mathbf{AB}^{\mathsf{ss}}_{\mathsf{h}}$ finds used classes, `Reachable`, as those reachable from CUT nodes in a statically computed class dependency graph (CDG). In brief, CDG nodes are types (`.class` files) in code, tests, or libraries. There is a directed edge from node `A` to node `B` if `A` can use `B`. Also, `B` is reachable from `A` if there is a path from `B` to `A`. First, $\mathbf{AB}^{\mathsf{ss}}_{\mathsf{h}}$ generates CDG. For new jars, $\mathbf{AB}^{\mathsf{ss}}_{\mathsf{h}}$ instruments `.class` files that are in `Reachable`. If a jar is updated, then the `.class` files to re-instrument are obtained as those in `Reachable` whose cleaned bytecode changed. Lastly, even if a jar did not change, changes to the CUT can alter the control flow such that previously unused (and un-instrumented) library classes are now used in the new revision. In this case, $\mathbf{AB}^{\mathsf{ss}}_{\mathsf{h}}$ finds and instruments all `.class` files that are not in the old CDG's `Reachable` but are in the new CDG's `Reachable`. The new CDG is saved to disk.

In Figure 5, $\mathbf{AB}^{\mathsf{ss}}_{\mathsf{h}}$ and $\mathbf{AB}^{\mathsf{sd}}_{\mathsf{h}}$ will only re-instrument `T` and `L2`; they will not re-instrument `L3`, which changed but is not used by any CUT class. $\mathbf{AB}^{\mathsf{sd}}_{\mathsf{h}}$ and $\mathbf{AB}^{\mathsf{ss}}_{\mathsf{h}}$ re-instrument all identified `.class` files in sequence. Their parallel analogs, $\mathbf{AB}^{\mathsf{pd}}_{\mathsf{h}}$ and $\mathbf{AB}^{\mathsf{ps}}_{\mathsf{h}}$, respectively, use multiple threads. $\mathbf{AB}^{\mathsf{ss}}_{\mathsf{h}}$ and $\mathbf{AB}^{\mathsf{ps}}_{\mathsf{h}}$ are safe by design. But, in practice, their safety depends on the soundness of the static analysis used to build the CDG.

3) Usage-Aware, Framework-Specific Techniques: Some instrumentation frameworks (e.g., AspectJ's `ajc`) are compilers. So, they may already support incremental instrumentation that can be exploited for evolution-aware RV. Native, framework-specific incremental instrumentation can be fast. But, there are disadvantages:

1. Supporting incremental compilation is hard [90], [91], [110] and can make IMOP slower (if there are many classes to re-instrument [52]) or unsafe (if bugs in the compiler make it fail to re-instrument some changed classes).
2. RV tool developers may not have expertise to fix compiler bugs or integrate better with RV, so the resulting instrumentation-driven techniques may become reliant on compiler developers who are already pressed for time.
3. Some general-purpose (e.g., ASM [7], BCEL [16], DiSL [89], Javassist [64]) and RV-inspired (e.g., BISM [115]–[118]) frameworks do not support incremental instrumentation. Others, e.g., ByteBuddy [24] support incremental instrumentation of CUT, but not libraries.
4. Compiler-specific incremental instrumentation could make it harder for RV tool developers to switch among these instrumentation tools, leading to “vendor lock-in”.

Two IMOP techniques are AspectJ-specific; JavaMOP (the RV tool that we use in this paper) uses AspectJ. Also, 11 of 18 RV tools for Java in a publicly-available list [109] that is crowd-sourced by the RV research community also use AspectJ for instrumentation. So, our results may generalize beyond JavaMOP. Lastly, we discovered experimental support for incremental instrumentation in the AspectJ compiler's (**ajc**) source code [4]. This support is not part of AspectJ's public or advertised APIs or options. The advertised support [1] does not work for libraries, so it is unfit for IMOP. We next describe these two **ajc**-specific IMOP techniques.

- $\mathbf{ajc}^{\mathsf{def}}$ stores checksums of all loaded `.class` files (CUT plus libraries) that it instruments in the old revision. Then, in the new revision, it compares checksums of loaded `.class` files with those from the old revision. If a `.class` file's checksums in both revisions are the same, then the instrumented `.class` file from the old revision is fetched from a cache and re-used in the new revision. If the checksums differ, then the `.class` file in the new revision is re-instrumented, added to the cache, and its checksum is updated for use in the next revision. $\mathbf{ajc}^{\mathsf{def}}$ uses one cache per classloader.
- $\mathbf{ajc}^{\mathsf{one}}$ works like $\mathbf{ajc}^{\mathsf{def}}$, but it uses one cache for all classloaders. $\mathbf{ajc}^{\mathsf{one}}$'s caching also uses a slower data structure.

In Figure 5, $\mathbf{ajc}^{\mathsf{def}}$ and $\mathbf{ajc}^{\mathsf{one}}$ will re-instrument `T`, `L1`, and `L2` even though `L2`'s cleaned bytecode did not change. That is, these framework-specific techniques would imprecisely re-instrument a changed `.class` file even if its bytecode has not changed since the old revision.

Summary of IMOP techniques. Table III summarizes IMOP techniques' main features, for ease of reference.

C. Implementation

We implement IMOP's workflow (Figure 4) and techniques (§III-B) in a Maven plugin, for easier integration with Maven-based Java projects. After installation, users only need to change a few lines in a Maven configuration file (`pom.xml`). We only implement IMOP for Maven to focus our evaluation on the techniques, which are not Maven specific. Future work can support other build systems.

We choose JavaMOP because it was evaluated at scale during testing of open-source projects [65], [78], [80], [82], [94], [126]. It is not clear that other RV tools can simultaneously monitor 160 specs during software testing. Some RV tools

TABLE IV: Summary statistics on 66 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	265.8	9.1	12,658.5	66.6	57.7	581.6	9.8	621.3
Med	79.5	3.5	5,294.0	70.8	60.9	244.5	10.0	59.5
Min	2	1.5	559	0.4	0.3	38	1	1
Max	4,232	74.7	$2.1{\times}10^5$	99.2	99.3	5,144	22	11,993
Sum	17,546	598.4	$8.4{\times}10^5$	n/a	n/a	n/a	n/a	41,009

can only check one spec at a time [65], which would mean at least a 160x overhead in our case. Others require some manual setup that would not be feasible at our scale.

IMOP uses the same code for checking bytecode-level changes that is used in several open-source RTS tools [38], [48], [83], [85], [119] and eMOP [126]. Also, IMOP uses `jdeps` and STARTS to build the CDG (§III-B2). But, we extend STARTS to also include library classes in the CDG—STARTS only reasons about changed libraries at the coarse-grained level of jars.

To find loaded classes ($\mathbf{AB}_{\mathsf{h}}^{\mathsf{sd}}$ and $\mathbf{AB}_{\mathsf{h}}^{\mathsf{pd}}$), IMOP first runs tests with the `-verbose:class` JVM flag and post-processes the output. IMOP only invokes JavaMOP for monitoring, preventing it from re-instrumenting the code. Lastly, to integrate IMOP with **ajc** (§III-B3), we run `ajc` with the `aj.weaving.cache.enabled = true` and `aj.weaving.cache.impl = shared` options.

IV. Evaluation

We answer the following research questions:

RQ1. How do the runtime overheads of IMOP's techniques compare with those of existing approaches?

RQ2. How does the safety of IMOP's techniques compare with those of existing techniques?

RQ3. How do IMOP's speedups compare to those from just applying regression test selection (RTS) to full RV?

RQ4. How do internal metrics of IMOP's techniques compare with those of existing techniques?

RQ1 measures IMOP's overheads, comparing it with those of full RV and two spec-driven techniques. RQ2 measures IMOP's safety and compares it with those of two spec-driven techniques. RQ3 combines and compares IMOP with regression test selection (RTS). RQ4 assesses IMOP's internals.

A. Experimental setup

Evaluated projects and specs. We evaluate IMOP on 2,028 revisions of 66 open-source projects, using 160 specs of correct JDK API usage. Table IV shows summary statistics on the evaluated projects; the caption describes the columns and "n/a" are meaningless sums. (Full per-project statistics are in our appendix.) 36/66 projects are from the 37 instrumentation-dominated (having the largest differences in total RV time minus monitoring time) ones used in Guan and Legunsen's study [52] to evaluate a proof-of-concept (§VII compares with that work in more detail). We exclude one of the 37 projects from Guan and Legunsen's study because of a limitation which caused our bytecode-cleaning infrastructure to fail on that excluded project. 12/66 evaluated projects are from the instrumentation-dominated ones from eMOP's evaluation [126] that we can run; we exclude others because their tests failed during our experiments. Finally, we select an additional 18 projects from Guan and Legunsen's study [52]; they are the next most instrumentation-dominated ones that were not used to evaluate the prototype in that study.

For all 48 projects from prior work [52], [126], we use the same GitHub revisions as those prior works per project. For the other 18, we select up to 25 historical revisions from GitHub where at least one Java file changed, code compiles, and tests pass with and without JavaMOP and IMOP.

The 160 specs that we use are from prior work [76], [87]; they were used to evaluate RV during testing [52], [65], [77], [78], [80], [82], [94], [126] and helped find many bugs that testing alone missed.

Baselines. We compare IMOP to full RV (re-running JavaMOP's evolution-*unaware* RV from scratch after every change), two spec-driven techniques, and tRV—the best possible theoretical lower bound on RV overhead which assumes an ideal instrumentation time of zero, measured as full RV time minus instrumentation time. We next summarize the two spec-driven techniques that we compare with, using the names—$\boldsymbol{ps}_1^{\mathsf{c}}$ and $\boldsymbol{ps}_3^{\mathsf{c}\ell}$—that their authors use; see full details in [78].

$\boldsymbol{ps}_1^{\mathsf{c}}$ is the faster of two safe-by-design spec-driven techniques. $\boldsymbol{ps}_1^{\mathsf{c}}$ first finds classes impacted by code changes as those (i) whose cleaned `.class` file changed (Δ), (ii) that transitively depend on Δ (dependents), (iii) that Δ transitively depends on, and (iv) that Δ and its dependents can pass data to. Then, $\boldsymbol{ps}_1^{\mathsf{c}}$ finds affected specs as those related to impacted classes. Lastly, $\boldsymbol{ps}_1^{\mathsf{c}}$ re-monitors only affected specs, but it does not instrument them in un-impacted classes. The idea is that if the set of impacted classes is small, then affected specs are likely to be a small subset of all available specs.

In contrast to $\boldsymbol{ps}_1^{\mathsf{c}}$, $\boldsymbol{ps}_3^{\mathsf{c}\ell}$ is the fastest of 10 unsafe-by-design techniques that are deliberately designed to trade safety for speed. An evolution-aware RV technique is safe if it finds all new violations after a code change [78]. $\boldsymbol{ps}_3^{\mathsf{c}\ell}$ can miss new violations after a code change because: it (i) only finds impacted classes as Δ and its dependents; and (ii) does not instrument 3rd-party libraries (not even used library classes). We use the $\boldsymbol{ps}_1^{\mathsf{c}}$ and $\boldsymbol{ps}_3^{\mathsf{c}\ell}$ implementations in eMOP [126].

Running experiments. We write Maven extensions to integrate JavaMOP, IMOP, eMOP, and a profiler (RQ4) into evaluated projects. We write scripts to run tests, run RV, and analyze results. We run all experiments in Docker containers, to aid reproducibility. Our artifact has our Docker files and how to use them. We use an Intel® Xeon® Gold 6348 machine (112 cores) with 512GB of RAM running Ubuntu 20.04.4 LTS and Java 8 to run experiments that report absolute time.

B. RQ1: Runtime overheads

Aggregated results. Figure 6 shows the results of our performance evaluation. There, we show absolute times and relative

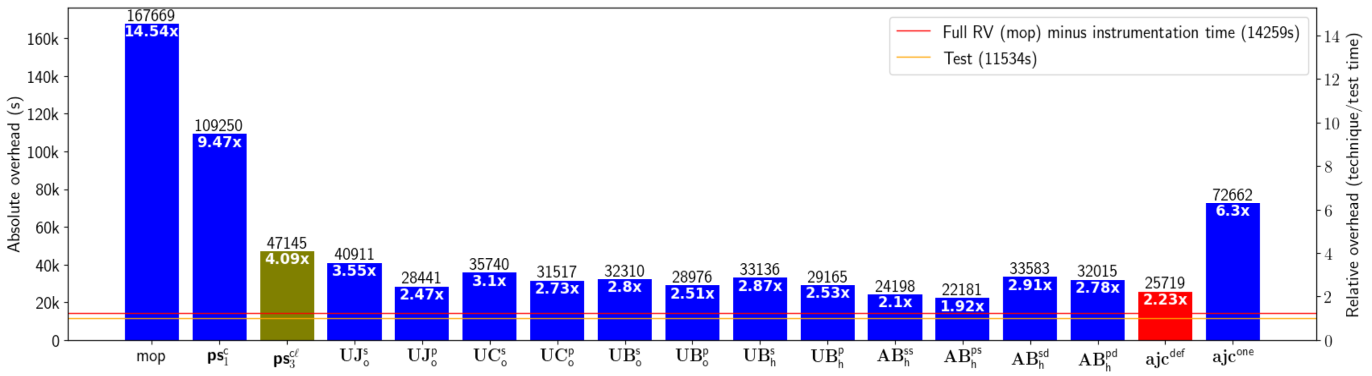

Fig. 6: Absolute and relative overheads across all 66 projects. Blue bars: safe-by-design techniques. Olive bar: unsafe-by-design spec-driven $\boldsymbol{ps}_3^{c\ell}$. Red bar: a safe-by-design framework-specific **ajc**^{def} that fails in 19 of 66 projects.

TABLE V: Summary statistics on speedups per project by the best-performing framework-specifc and framwork agnostic IMOP technique, relative to full RV (mop), $\boldsymbol{ps}_1^c$, $\boldsymbol{ps}_3^{c\ell}$, and full RV minus instrumentation time (tRV).

	$\frac{\text{mop}}{\text{agnostic}}$	$\frac{\text{mop}}{\text{specific}}$	$\frac{\boldsymbol{ps}_1^c}{\text{agnostic}}$	$\frac{\boldsymbol{ps}_1^c}{\text{specific}}$	$\frac{\boldsymbol{ps}_3^{c\ell}}{\text{agnostic}}$	$\frac{\boldsymbol{ps}_3^{c\ell}}{\text{specific}}$	$\frac{\text{agnostic}}{\text{tRV}}$	$\frac{\text{specific}}{\text{tRV}}$
Mean	10.0	6.2	5.8	3.9	2.4	1.5	1.9	4.1
Med	8.3	4.8	5.8	2.8	2.1	1.4	1.6	2.5
Min	2.4	0.9	1.4	0.1	0.5	0.1	0.7	0.8
Max	40.2	20.8	17.8	13.8	6.7	4.4	5.2	16.8

overheads when times are summed across all 66 projects, excluding the initial revision in each project. Absolute time is the end-to-end time from Maven invocation to termination. Relative overhead is the ratio of time to run (evolution-aware) RV to time to run tests without RV; the lower, the better. The yellow line in Figure 6 marks the time to run tests without RV. The red line marks tRV time. The relatively small time difference (see legend) between the red and yellow lines shows that instrumentation, not monitoring, dominates RV overhead during testing in these projects.

Figure 6 shows that all simple, framework-agnostic, safe-by-design IMOP techniques speedup full RV (mop) and outperform both $\boldsymbol{ps}_1^c$ and $\boldsymbol{ps}_3^{c\ell}$. In aggregate, these IMOP techniques are 4.1x–7.6x faster than full RV, 2.7x–4.9x faster than safe $\boldsymbol{ps}_1^c$, and 1.2x–2.1x faster than unsafe $\boldsymbol{ps}_3^{c\ell}$, despite the fact that we did not sacrifice safety in their design. Across all projects and their revisions, IMOP saves up to 40.4 hours compared to full RV and saves 24.2 hours and 6.9 hours, compared to safe and unsafe spec-driven techniques, respectively.

Figure 6 also seems to show that framework-specific **ajc**^{def} provides good speedups compared to full RV, $\boldsymbol{ps}_1^c$ and $\boldsymbol{ps}_3^{c\ell}$, and that **ajc**^{def} seems to be faster than 10 of 12 agnostic IMOP techniques. But, these results are inconclusive: **ajc**^{def}'s instrumentation fails in 19/66 projects. **ajc**^{one}, the other safe, framework-specific technique does not fail in any project but it is slower than all IMOP's agnostic techniques. However, **ajc**^{one} is 1.5x faster than $\boldsymbol{ps}_1^c$ and 2.3x faster than full RV.

We asked `ajc` developers if **ajc**^{def} failures are due to bugs [3]. Their response reinforces our sense that framework-specific incremental instrumentation is hard to get right and may be unreliable. An `ajc` maintainer (i) pointed us to a similar bug report whose solution did not fix our issues, (ii) confirmed that the problem was in `ajc` since at least 2017, and (iii) said if incremental instrumentation "*ever worked ... and just broke a long time ago or if it never worked ..., is yet to be established*". Worse, the maintainer has now quit the AspectJ project, and may no longer contribute without financial support [2]. We therefore exclude **ajc**^{def}'s result from timed experiments if its instrumentation fails on a project.

More detailed results. Table V shows summary statistics on speedups per-project (our appendix has more data, e.g., best-performing technique per project). There, agnostic techniques are up to 40.2x (mean: 10x), 17.8x (mean: 5.8x), and 6.7x (mean: 2.4x) faster than full RV, $\boldsymbol{ps}_1^c$, and $\boldsymbol{ps}_3^{c\ell}$, respectively. The corresponding numbers for framework-specific techniques are 20.8x (mean: 6.2x), 13.8x (mean: 3.9x), and 4.4x (mean: 1.5x). So, on a per-project basis, agnostic techniques tend to provide more speedups than framework-specific ones. The 0.1x minima in Table V are for a project where $\boldsymbol{ps}_1^c$ and $\boldsymbol{ps}_3^{c\ell}$ outperform the best framework-specific technique (**ajc**^{def} fails and **ajc**^{one} is very slow on that project).

The two rightmost columns in Table V show how much *slower* IMOP's best-performing techniques are, compared to tRV (the best theoretical lower bound on RV overhead that assumes zero instrumentation cost). There, "min" is how closely IMOP approaches tRV. At best, IMOP is 30% faster than tRV, which is evolution-*unaware* and may run tests with monitoring even if changes do not modify cleaned bytecode. On average, the best-performing agnostic IMOP technique is 88% (max: 5.2x) slower than tRV. These tRV-related results show how closely IMOP approaches theoretical evolution-unaware RV with no instrumentation cost.

Figure 7 shows how often the speedup achieved by each technique is among the top three; darker colors mean higher rank and each row sums up to the number of evaluated projects (66). Such ranking is not visible from the aggregated results (Figure 6). Our appendix has a full 17×17 ranking and all techniques' rank per project. We make five main observations:

1. There is no universally best IMOP technique: 11/14 IMOP techniques provide the best speedup in at least one project. Also, each IMOP technique except **ajc**^{one} is in the top-three at least three times. This result justifies having several IMOP techniques, and exploring the tradeoff space (§III).

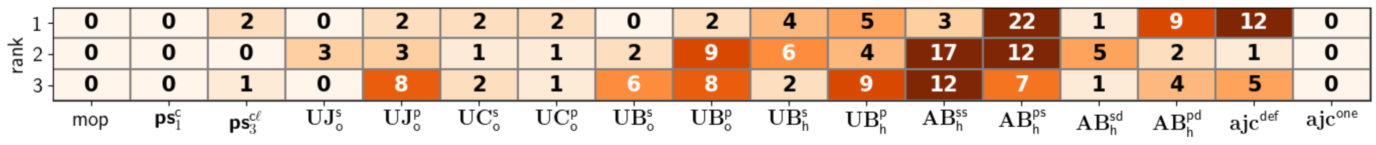

Fig. 7: Heat map showing how often a technique provides one of the top-three speedups per project. The top, middle, and bottom rows show no. of projects where a technique provides the best, 2nd-best, and 3rd-best speedup, respectively.

TABLE VI: Safety results for two spec-driven techniques and iMOP before (**pre**) and after (**post**) our manual inspection.

	ps_1^{c}	$ps_3^{c\ell}$	$\mathbf{UJ}_o^s$	$\mathbf{UJ}_o^p$	$\mathbf{UC}_o^s$	$\mathbf{UC}_o^p$	$\mathbf{UB}_o^s$	$\mathbf{UB}_o^p$	$\mathbf{UB}_h^s$	$\mathbf{UB}_h^p$	$\mathbf{AB}_h^{ss}$	$\mathbf{AB}_h^{ps}$	$\mathbf{AB}_h^{sd}$	$\mathbf{AB}_h^{pd}$	$\mathbf{ajc}^{def}$	$\mathbf{ajc}^{one}$
Missed new violations (**pre**, of 1579)	168	403	72	75	72	71	147	148	145	149	191	187	117	117	135	69
Missed new violations (**post**, of 1160)	14	44	0	0	0	0	0	0	0	0	17	17	0	0	40	0
Unsafe revisions (**pre**, of 2028)	70	173	35	38	34	35	66	68	66	68	117	115	59	58	75	34
Unsafe revisions (**post**, of 2028)	10	26	0	0	0	0	0	0	0	0	9	9	0	0	15	0
Unsafe projects (**pre**, of 66)	19	23	11	10	10	11	23	23	22	23	24	24	21	20	17	10
Unsafe projects (**post**, of 66)	7	14	0	0	0	0	0	0	0	0	7	7	0	0	6	0

2. Agnostic techniques provide the best speedups in 52/66 projects. This result may be affected by $\mathbf{ajc}^{def}$'s failures, but it suggests that continued improvement of agnostic techniques would be worthwhile.
3. When agnostic techniques provide the best speedup, usage-*unaware* ones perform best in 17/52 projects. So, the added complexity of checking if changed library code is used by the CUT is not needed in a good fraction of projects.
4. Surprisingly, $\mathbf{AB}_h^{sd}$ and $\mathbf{AB}_h^{pd}$, which run tests twice, provide the best speedup in 10 projects.
5. iMOP outperforms spec-driven techniques in all but two projects where iMOP is just two and six seconds slower.

The important question of how users should choose what iMOP technique to use requires more research. §V discusses initial observations from our qualitative analysis.

C. RQ2: Safety

Table VI shows the results of our safety evaluation. A safe evolution-aware RV technique finds all new violations after a code change [78]. In coming up with this definition of safety, Legunsen et al. assume a setting where users are aware of violations in the old revision, and are more interested to see new violations after code changes [78]. To find new violations, we run Violation Message Suppression (VMS) [78], an evolution-aware RV technique that reduces human time to inspect violations; it does not reduce runtime overhead. VMS hides violations in unchanged lines of code; users can view hidden violations, but only new ones are shown by default.

We run eMOP's VMS implementation [126] on full RV in a separate un-timed experiment and report numbers of missed VMS-reported violations before ("**pre**" rows) and after ("**post**" rows) our manual inspection. The 1st, 3rd, and 5th rows show numbers of missed VMS-reported violations, number of revisions with a missed violation, and number of projects with an unsafe revision before inspection, respectively. The 2nd, 4th, and 6th rows show these numbers after inspection.

Our inspection shows that 419 of 1,579 VMS-reported violations are not related to changes (difference between the 1st and 2nd rows in the first column of Table VI). Rather, they are due to non-deterministic test executions (80 violations), false positives or bugs in VMS (287 violations), or false positives due to bytecode cleaning in iMOP misleading VMS about line numbers (52 violations). Precisely matching lines of code locations across revisions is hard [53], [86], [102], leading to VMS' false positives. We have reported two related VMS bugs to the eMOP developers [121], [122].

After inspection, we find that 11 iMOP techniques—all eight usage-*unaware* iMOP techniques, and three of six usage-aware ones—are safe in our experiments. Among the three usage-aware techniques that miss a new violation, the overall ratio of unsafe revisions is small (0.44%–0.74%). These safety results are encouraging. Also, iMOP finds 35 violations that spec-driven techniques miss. If all iMOP's techniques find a violation but at least one spec-driven technique does not and we find no evidence of test non-determinism, we count that violation as being missed by that spec-driven technique.

We analyze why new violations are missed, to better understand how to make iMOP's techniques safer in practice. $\mathbf{AB}_h^{ss}$ and $\mathbf{AB}_h^{ps}$ use static analysis and miss 17 new violations related to using reflection to load classes. $\mathbf{ajc}^{def}$ misses 40 new violations that are due to `ajc` failures during instrumentation. Note that these techniques are safe by design (unlike the unsafe-by-design techniques in [78], we did not deliberately design them to be unsafe). Future work is needed to also make the implementations of three (of 14) techniques safe.

Unsurprisingly, $ps_3^{c\ell}$ (spec driven and unsafe by design) is the most unsafe in our experiments; it misses 44 new violations (4%). 37/44 are due to $ps_3^{c\ell}$'s use of a less conservative (than ps_1^{c}) static analysis to find affected specs to re-monitor. The other seven are due to two bugs in eMOP's implementation that we have reported [106], [107]. ps_1^{c} (safe by design) misses 14 violations due to the same eMOP bugs that affect $ps_3^{c\ell}$.

We conclude that usage-*unaware* iMOP techniques should be used when safety is critical: they are safe, faster than unsafe $ps_3^{c\ell}$, simple, and often provide the best speedups. Other settings can use faster usage-aware techniques; two of them are iMOP's fastest techniques (Figure 6), but they show very small degrees of unsafety that should be improved in the future.

D. RQ3: Comparison with RTS

Evolution-aware RV re-runs all tests in a new revision, while regression test selection (RTS)—a regression testing

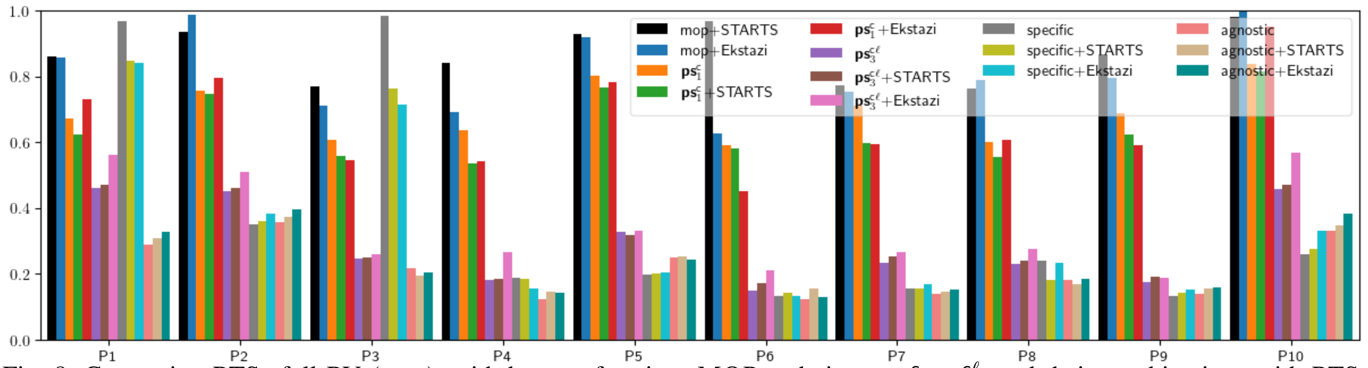

Fig. 8: Comparing RTS+full RV (mop), with best-performing IMOP technique, $\boldsymbol{ps}_1^{\mathrm{c}}$, $\boldsymbol{ps}_3^{\mathrm{c}\ell}$, and their combinations with RTS.

technique—aims to speed up testing by only re-running affected tests [125]. We compare the runtime overhead and safety of combining RTS with full RV with those of the best-performing agnostic and framework-specific IMOP technique per project, $\boldsymbol{ps}_1^{\mathrm{c}}$, $\boldsymbol{ps}_3^{\mathrm{c}\ell}$, and their combinations with RTS. We are the first to evaluate RTS with instrumentation-driven techniques; others evaluated RTS with spec-driven techniques [80].

Due to the many combinations, we run RTS experiments only on the top 10 Table IV projects (in decreasing order of speedups) that we could successfully run with Ekstazi [38], [48] and STARTS [83], [119]—two open-source RTS tools that use dynamic and static analysis, respectively, to find affected tests. We could not simply pick the top 10 projects because Ekstazi failed for some of them due to a known problem [37]. The 10 projects that we use in RQ3 have a total of 365 revisions. We seamlessly integrate both Ekstazi and STARTS into IMOP, so running RTS with IMOP does not require additional setup from the users.

Figure 8 shows the RTS-related results. Speedups from these techniques with and without RTS are shown as ratios of each technique's time to full RV time (the 1.0 line); the further below 1.0, the better. STARTS selects 33%–83% (mean: 62%) of all tests, while Ekstazi selects 26%–78% (mean: 45%).

We make several observations from Figure 8. (i) In all 10 projects; RTS speeds up full RV, but RTS+full RV never yields the best speedup. (ii) RTS speeds up $\boldsymbol{ps}_1^{\mathrm{c}}$ in all 10 projects (Ekstazi+$\boldsymbol{ps}_1^{\mathrm{c}}$ is slower than $\boldsymbol{ps}_1^{\mathrm{c}}$ in four projects). But, RTS+$\boldsymbol{ps}_1^{\mathrm{c}}$ never yields the best speedup. (iii) RTS (STARTS) speeds up $\boldsymbol{ps}_3^{\mathrm{c}\ell}$ in only 1/10 project. $\boldsymbol{ps}_3^{\mathrm{c}\ell}$ is already fast, so RTS analysis time becomes an overhead. Also, RTS+$\boldsymbol{ps}_3^{\mathrm{c}\ell}$ never provides the best speedup. (iv) For the same reason as with $\boldsymbol{ps}_3^{\mathrm{c}\ell}$, combining RTS with IMOP often causes slowdowns. But, RTS speeds up an IMOP technique in 5/10 projects.

Combining IMOP and $\boldsymbol{ps}_1^{\mathrm{c}}$ with STARTS did not miss any new violation. But, $\boldsymbol{ps}_3^{\mathrm{c}\ell}$ misses two violations when combined with Ekstazi because of limitations of $\boldsymbol{ps}_3^{\mathrm{c}\ell}$'s static analysis.

Overall, IMOP outperforms RTS+full RV, but RTS+IMOP is faster than IMOP alone in 5/10 evaluated projects. So, we conclude that IMOP and RTS are orthogonal but complementary approaches to speeding up RV during testing of projects where instrumentation time dominates RV overhead.

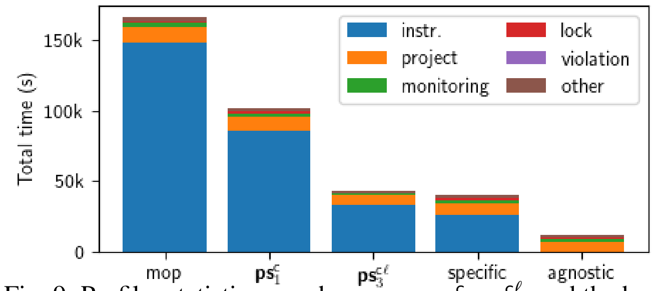

Fig. 9: Profiler statistics on where mop, $\boldsymbol{ps}_1^{\mathrm{c}}$, $\boldsymbol{ps}_3^{\mathrm{c}\ell}$, and the best-performing framework-specific and agnostic IMOP techniques spend their runtimes across all 2,028 project revisions.

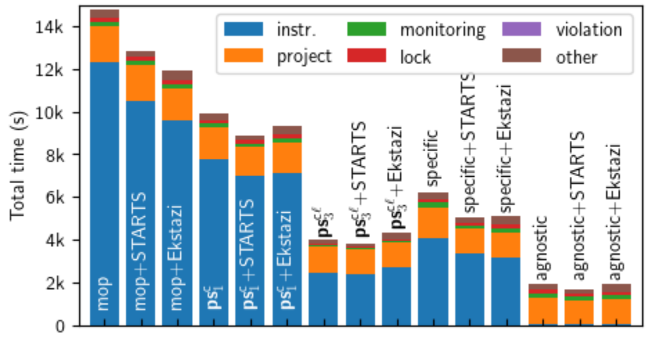

Fig. 10: Results from profiling RTS runs in 10 projects.

E. *RQ4: Internal* IMOP *metrics*

Profiler data. Figure 9 shows the profiler-reported proportions of time spent on instrumentation (instr.), running the CUT (project) and RV—monitoring, synchronization ("lock") to process events without races, and printing violations (which can be costly [52]). We use the same procedure as [52], using async-profiler [8], [62]. Profiler-related runs are not timed.

Figure 9 shows that (i) monitoring accounts for very little proportion of RV time across all 66 evaluated projects; (ii) IMOP techniques provide more speedups than $\boldsymbol{ps}_1^{\mathrm{c}}$ and $\boldsymbol{ps}_3^{\mathrm{c}\ell}$ because they reduce more instrumentation time; and (iii) agnostic techniques spend no runtime on instrumentation. Agnostic techniques perform instrumentation at compile-time; that time is included in RQ1. Framework-specific techniques and full RV perform instrumentation during class loading.

Figure 10 shows the results of profiling the RTS runs (colors mean the same as Figure 9). There, some reasons for the

TABLE VII: No. of CUT classes, jars, and library classes re-instrumented by five iMOP techniques across all projects.

	$\mathbf{UJ}^{\mathsf{s}}_{\mathsf{o}}$	$\mathbf{UC}^{\mathsf{s}}_{\mathsf{o}}$	$\mathbf{UB}^{\mathsf{s}}_{\mathsf{o}}$	$\mathbf{AB}^{\mathsf{ss}}_{\mathsf{h}}$	$\mathbf{AB}^{\mathsf{sd}}_{\mathsf{h}}$
# CUT classes	23101	23101	20655	20655	20655
# jars	1933	1933	1933	1352	1307
# library classes	776172	516711	422775	314869	156000

trends in RTS results (RQ3) is clearer. For example, RTS+full RV is faster than full RV because it spends less time on CUT and instrumentation. But RTS+full RV still spends more time on instrumentation than iMOP techniques (alone or in combination with RTS), explaining why it is slower. Similar trends hold for other combinations with RTS.

Re-instrumented class counts. Table VII shows counts of CUT classes, jars, and library classes that $\mathbf{UJ}^{\mathsf{s}}_{\mathsf{o}}$, $\mathbf{UC}^{\mathsf{s}}_{\mathsf{o}}$, $\mathbf{UB}^{\mathsf{s}}_{\mathsf{o}}$, $\mathbf{AB}^{\mathsf{ss}}_{\mathsf{h}}$, and $\mathbf{AB}^{\mathsf{sd}}_{\mathsf{h}}$ re-instrument across all 66 projects. We only show these techniques to save space; they showcase differences among (i) usage-unawareness (**U**) and usage-awareness (**A**); (ii) finding changes at jar (**J**), `.class` (**C**), and bytecode (**B**) levels; and (iii) checking usage statically (2nd superscript is s) and dynamically (2nd superscript is d). We add a jar to the second row if one of its classes is re-instrumented.

In Table VII, $\mathbf{UJ}^{\mathsf{s}}_{\mathsf{o}}$ re-instruments the most classes (it re-instruments all classes in a jar if any of the jar's contents changed). By re-instrumenting only changed `.class` files in changed jars, $\mathbf{UC}^{\mathsf{s}}_{\mathsf{o}}$ re-instruments much less than $\mathbf{UJ}^{\mathsf{s}}_{\mathsf{o}}$. Also, by re-instrumenting only changed `.class` files whose bytecode changed, $\mathbf{UB}^{\mathsf{s}}_{\mathsf{o}}$ re-instruments fewer CUT and library classes than $\mathbf{UC}^{\mathsf{s}}_{\mathsf{o}}$. $\mathbf{AB}^{\mathsf{ss}}_{\mathsf{h}}$ and $\mathbf{AB}^{\mathsf{sd}}_{\mathsf{h}}$ are usage-aware, so they re-instrument even fewer library classes than these usage-unaware techniques. Overall, $\mathbf{AB}^{\mathsf{sd}}_{\mathsf{h}}$ re-instruments the least; it is the most precise (but it incurs overhead to run tests twice).

V. Discussion

When does each iMOP technique tend to perform best? We perform an initial qualitative analysis of program characteristics that tend to hold when each iMOP technique performs best. We find that framework-specific $\mathbf{ajc}^{\mathsf{def}}$ tends to perform better than agnostic techniques when few classes change. When libraries change frequently, $\mathbf{AB}^{\mathsf{ps}}_{\mathsf{h}}$ tends to perform best if the test-running time is long. For such projects, if the test-running time is short, then $\mathbf{AB}^{\mathsf{pd}}_{\mathsf{h}}$'s higher precision enables it to outperform others. $\mathbf{UJ}^{\mathsf{p}}_{\mathsf{o}}$ tends to perform better than $\mathbf{UC}^{\mathsf{p}}_{\mathsf{o}}$ and $\mathbf{UB}^{\mathsf{p}}_{\mathsf{o}}$ when a project switches major versions (in the semantic versioning sense) of libraries often, and each such switch modifies many library classes. In such cases, all three techniques re-instrument almost all library classes involved. But, $\mathbf{UJ}^{\mathsf{p}}_{\mathsf{o}}$ does not have the overhead of comparing each `.class` file or cleaning the bytecode as $\mathbf{UC}^{\mathsf{p}}_{\mathsf{o}}$ and $\mathbf{UB}^{\mathsf{p}}_{\mathsf{o}}$ do.

For projects that frequently change the CUT (but not the libraries), $\mathbf{UB}^{\mathsf{s}}_{\mathsf{h}}$ and $\mathbf{UB}^{\mathsf{p}}_{\mathsf{h}}$ tend to be the best techniques. This initial analysis can be the first step towards more detailed qualitative analysis and future work on automated prediction of which iMOP technique to use for a project or code change.

Can iMOP work beyond AspectJ? iMOP also supports BISM [115]–[118]; a recently proposed instrumentation framework for RV. BISM aims to address some drawbacks (e.g., runtime overhead and learning curve) of AspectJ, the dominant Java instrumentation framework in the RV literature.

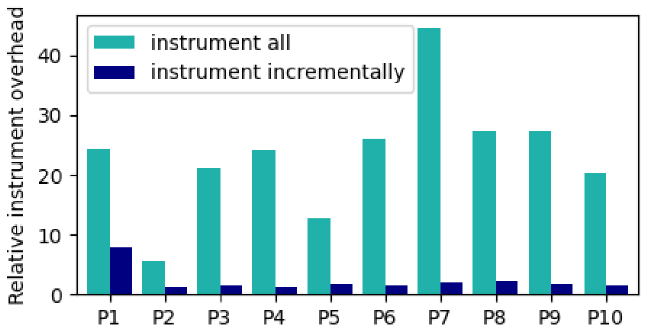

Fig. 11: iMOP overheads relative to test time when using BISM with and without evolution awareness in 10 projects.

We evaluate iMOP's BISM support on the 10 RQ3 projects by comparing (i) the overhead (relative to running tests without RV) of using BISM to re-instrument all CUT and library classes in each revision with (ii) the overhead of using BISM to re-instrument only the changed `.class` files or bytecode that the best-performing agnostic iMOP technique finds as changed. We cannot evaluate BISM with monitoring as it was not yet integrated with an RV tool (to our knowledge) and re-engineering JavaMOP to use BISM could take years.

We observe from the BISM results in Figure 11 that: (i) BISM also incurs high overheads to instrument all classes in these projects, so high instrumentation overheads are not specific to AspectJ. (ii) Evolution-awareness reduces BISM overhead. But, BISM crashes on 118/364 jars that it encountered. BISM was only recently proposed and it is not yet open sourced. So, we report these crashes to the BISM authors to help improve the tool.

What if iMOP trades safety for speed? $\boldsymbol{ps}^{\mathsf{c}\ell}_{3}$ is unsafe but fast and can perform well on some projects (Table VI, [78], [126]). So, we experiment with four techniques in iMOP that trade safety for speed by design and evaluate them on all 66 projects. (These experimental techniques are not part of iMOP; details about them are in our artifact). They are as unsafe as $\boldsymbol{ps}^{\mathsf{c}\ell}_{3}$, but much slower than the best-performing safe iMOP technique per project, and barely faster than $\boldsymbol{ps}^{\mathsf{c}\ell}_{3}$.

Sensitivity to thread counts. By default, iMOP uses 20 threads in its parallel techniques, but users can change this thread count via the command line. We use 20 after performing a preliminary analysis of the sensitivity of these iMOP techniques to the thread counts. For this analysis, we run the best-performing, parallel usage-aware and usage-unaware iMOP technique per project on all revisions of the 10 RQ3 projects, using 10, 20, and 40 threads. Table VIII shows the results: using 20 threads saves more time than 10 threads, but using 40 instead of 20 threads does not save much.

VI. Threats to Validity and Limitations

Threats to validity. To reduce the threat of poor generalization to other instrumentation frameworks, iMOP supports

TABLE VIII: iMOP times with 10, 20, and 40 threads.

	10 threads	20 threads	40 threads
Usage unaware (best)	3492.6s	3391.7s	3399.7s
Usage aware (best)	3323.1s	3279.6s	3268.9s

AspectJ and BISM. Our results may not generalize beyond the projects and their historical GitHub revisions that we evaluate. But, we show that amortizing instrumentation costs can provide more savings than existing approaches on 66 projects, which is the largest set evaluated with evolution-aware RV to date. Our manual inspection to check safety may miscategorize some violations. To reduce this threat, a co-author checked the results multiple times. These manual inspections were needed in the first place because of test non-determinism, as well as VMS bugs and limitations. Future work can target how to mitigate these problems.

Limitations. We address high RV overheads, which is only one of several problems to be solved towards broader RV adoption. We do not, e.g., address the difficulty of finding or improving specs, reduce tedium of violation inspection, or check if violations are true bugs. These problems are subjects of other research [47], [51], [80], [82], [94], [103], [120].

Our work may be limited to the kind of API-level specs we use, but these are the largest publicly available set. We make no claims beyond the popular RV style in JavaMOP (other styles exist [11], [23], [43], [58], [100]). Future work can investigate instrumentation-driven techniques for other kinds of specs, e.g., project-specific ones, and RV styles.

The projects on which we evaluate iMOP are those where instrumentation dominates the RV time. Prior work [52] shows that RV overhead in more than 85% of 1,544 evaluated projects is dominated by instrumentation, not monitoring. We do not expect iMOP to provide similar speedups in projects where monitoring time dominates RV overhead.

VII. Related Work

The most relevant work are spec-driven evolution-aware RV techniques [77], [78], [126], which we discuss and compare with in several parts of the paper. Here, we put iMOP in context with respect to the empirical study that inspired it, and discuss other related work.

Motivating study. iMOP is inspired by Guan and Legunsen's recent study which found that RV overheads during testing is often dominated by instrumentation [52]. They also provided preliminary evidence that amortizing instrumentation costs can speed up RV, $\boldsymbol{ps}_1^{c}$ and $\boldsymbol{ps}_3^{c\ell}$. But, we are the first to realize instrumentation-driven evolution-aware RV techniques in a tool, evaluate its performance on a larger scale, compare and combine instrumentation-driven techniques with RTS, and evaluate its safety. Their study evaluates two proofs-of-concept, one of which assumes a non-existent repository where all jars are pre-instrumented for RV. We refine and implement their other proof-of-concept into $\mathbf{UJ}_o^s$, and add 13 techniques.

Instrumentation in RV and other program analyses. Cassar et al. [25] survey instrumentation approaches in RV. Other works distribute instrumentation costs across users [20], [88] or develop instrumentation for sampling events [95]. These works are not concerned with RV during testing of evolving software. Reducing instrumentation costs has not received a lot of RV research attention, perhaps because most prior work (i) target deployed software, where instrumentation is a one-time cost; or (ii) were evaluated on DaCapo benchmarks [18] and measured performance after system warm-up. For RV during testing, especially in continuous integration, reducing high instrumentation overhead as iMOP does is critical.

Reducing RV overhead. Many works reduce RV's overhead [5], [9], [12], [13], [15], [19]–[23], [26], [29], [35], [44]–[46], [66], [68], [78], [87], [95], [97], [98], [101], [124], [126], but they often target the reduction of monitoring, not instrumentation, costs. iMOP is the first framework that reduces instrumentation costs in an evolution-aware manner.

Regression testing. iMOP is related to regression testing [41], [57], [125] which aims to speed up testing during software evolution. RTS [42], [49], [50], [54], [56], [75], [81], [85], [96], [104], [105], [112], [114], [123], [128]–[130] is a regression testing technique that aims to speed up regression testing by re-running only affected tests after a code change. We compare and combine iMOP with RTS (§IV-D). In the future we plan to further speed up this integration by building the dependency graph only once in some cases where certain iMOP techniques build their own dependency graph, and RTS separately builds its own graph. Also, we leave as future work the evaluation of RV with other regression testing techniques, such as test-suite minimization [17], [55], [69], [74], [92], [99], [111], [113] or test-suite prioritization [39], [40], [72], [127].

VIII. Conclusions

We present the first instrumentation-driven evolution-aware RV framework, iMOP, and its in-depth evaluation. The idea behind iMOP is simple: in a new code revision, re-instrument only changed code and re-use the old revision's instrumentation for unchanged code. On 66 projects where instrumentation dominates RV overhead, iMOP provides more speedup and is safe, compared to the state-of-the-art spec-driven evolution-aware RV techniques that use complex program analysis. iMOP is faster than just combining regression test selection (RTS) with RV, but iMOP and RTS are complementary.

Acknowledgments

We thank Marcelo d'Amorim, Saikat Dutta, Milos Gligoric, Pengyue Jiang, Shinhae Kim, Yu Liu, Sasa Misailovic, and the anonymous reviewers for their help, comments, and feedback on an earlier draft of this paper. 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] "AspectJ's CTW incremental instrumentation," https://eclipse.dev/aspectj/doc/latest/devguide/ajc.html.

[2] "AspectJ's maintenance status," https://www.eclipse.org/lists/aspectj-users/msg15598.html.

[3] "AspectJ's incremental instrumentation bug report," https://github.com/eclipse-aspectj/aspectj/issues/314.
[4] "AspectJ's LTW incremental instrumentation," https://github.com/eclipse-aspectj/aspectj/blob/3c6e30bdd9a8cda263dbfc73a529af96f2b01e40/weaver/src/main/java/org/aspectj/weaver/tools/cache/WeavedClassCache.java#L31.
[5] M. Arnold, M. Vechev, and E. Yahav, "QVM: An efficient runtime for detecting defects in deployed systems," in *OOPSLA*, 2008.
[6] "ARTCAT: Autonomic Response To Cyber-Attack," https://grammatech.github.io/prj/artcat.
[7] "ASM," http://asm.ow2.org.
[8] "Sampling CPU and HEAP profiler for Java," https://github.com/async-profiler/async-profiler.
[9] P. Avgustinov, J. Tibble, and O. de Moor, "Making trace monitors feasible," in *OOPSLA*, 2007.
[10] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-based runtime verification," in *VMCAI*, 2004.
[11] H. Barringer, D. Rydeheard, and K. Havelund, "Rule systems for run-time monitoring: From Eagle to RuleR," *Journal of Logic and Computation*, vol. 20, no. 3, 2010.
[12] E. Bartocci, B. Bonakdarpour, and Y. Falcone, "First international competition on software for runtime verification," in *RV*, 2014.
[13] E. Bartocci, Y. Falcone, B. Bonakdarpour, C. Colombo, N. Decker, K. Havelund, Y. Joshi, F. Klaedtke, R. Milewicz, G. Reger, G. Roşu, J. Signoles, D. Thoma, E. Zalinescu, and Y. Zhang, "First international competition on runtime verification: Rules, benchmarks, tools, and final results," *IJSTTT*, vol. 21, no. 1, 2019.
[14] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, "Introduction to runtime verification," in *Lectures on Runtime Verification*, 2018.
[15] E. Bartocci, Y. Falcone, and G. Reger, "International competition on runtime verification," in *TACAS*, 2019.
[16] "Commons BCEL," https://commons.apache.org/proper/commons-bcel.
[17] J. Black, E. Melachrinoudis, and D. Kaeli, "Bi-criteria models for all-uses test suite reduction," in *ICSE*, 2004.
[18] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA*, 2006.
[19] E. Bodden, "MOPBox: A library approach to runtime verification," in *RV*, 2011.
[20] E. Bodden, L. Hendren, P. Lam, O. Lhoták, and N. A. Naeem, "Collaborative runtime verification with Tracematches," in *RV*, 2007.
[21] E. Bodden, L. Hendren, and O. Lhoták, "A staged static program analysis to improve the performance of runtime monitoring," in *ECOOP*, 2007.
[22] E. Bodden, P. Lam, and L. Hendren, "Finding programming errors earlier by evaluating runtime monitors ahead-of-time," in *FSE*, 2008.
[23] B. Bonakdarpour, S. Navabpour, and S. Fischmeister, "Time-triggered runtime verification," in *FMSD*, vol. 43, 2013.
[24] "Byte Buddy - runtime code generation for the Java virtual machine," https://bytebuddy.net.
[25] I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir, "A survey of runtime monitoring instrumentation techniques," *arXiv preprint arXiv:1708.07229*, 2017.
[26] F. Chen, P. O. Meredith, D. Jin, and G. Roşu, "Efficient formalism-independent monitoring of parametric properties," in *ASE*, 2009.
[27] F. Chen and G. Roşu, "Towards monitoring-oriented programming: A paradigm combining specification and implementation," in *RV*, 2003.
[28] ——, "Mop: an efficient and generic runtime verification framework," in *OOPSLA*, 2007, pp. 569–588.
[29] ——, "Parametric trace slicing and monitoring," in *TACAS*, 2009.
[30] "Collections_SynchronizedCollection," https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collections.html#synchronizedCollection(java.util.Collection).
[31] M. d'Amorim and K. Havelund, "Event-based runtime verification of Java programs," in *WODA*, 2005.
[32] D. B. de Oliveira, "Efficient runtime verification for the Linux kernel," https://research.redhat.com/blog/article/efficient-runtime-verification-for-the-linux-kernel.
[33] ——, "Automata-based formal analysis and verification of the real-time Linux kernel," Ph.D. dissertation, Universidade Federal de Santa Catarina, Brazil, 2019.
[34] D. B. de Oliveira, T. Cucinotta, and R. S. de Oliveira, "Efficient formal verification for the Linux kernel," in *SEFM*, 2019, pp. 315–332.
[35] N. Decker, J. Harder, T. Scheffel, M. Schmitz, and D. Thoma, "Runtime monitoring with union-find structures," in *TACAS*, 2016.
[36] M. B. Dwyer, R. Purandare, and S. Person, "Runtime verification in context: Can optimizing error detection improve fault diagnosis?" in *RV*, 2010.
[37] "Mockito is not working," https://github.com/gliga/ekstazi/issues/61.
[38] "Ekstazi," http://ekstazi.org/.
[39] S. Elbaum, A. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *ISSTA*, 2000.
[40] S. Elbaum, G. Rothermel, S. Kanduri, and A. Malishevsky, "Selecting a cost-effective test case prioritization technique," *SQJ*, vol. 12, no. 3, 2004.
[41] E. Engström and P. Runeson, "A qualitative survey of regression testing practices," in *Product-Focused Software Process Improvement*, 2010.
[42] E. Engström, M. Skoglund, and P. Runeson, "Empirical evaluations of regression test selection techniques: A systematic review," in *ESEM*, 2008.
[43] U. Erlingsson and F. B. Schneider, "IRM enforcement of Java stack inspection," in *IEEE S&P*, 2000.
[44] Y. Falcone, S. Krstić, G. Reger, and D. Traytel, "A taxonomy for classifying runtime verification tools," in *RV*, 2018.
[45] Y. Falcone, D. Ničković, G. Reger, and D. Thoma, "Second international competition on runtime verification," in *RV*, 2015.
[46] V. Forejt, M. Kwiatkowska, D. Parker, H. Qu, and M. Ujma, "Incremental runtime verification of probabilistic systems," in *RV*, 2012.
[47] M. Gabel and Z. Su, "Testing mined specifications," in *FSE*, 2012.
[48] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *ICSE Demo*, 2015.
[49] ——, "Practical regression test selection with dynamic file dependencies," in *ISSTA*, 2015.
[50] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov, "An empirical evaluation and comparison of manual and automated test selection," in *ASE*, 2014.
[51] E. Goldweber, W. Yu, S. A. V. Ghahani, and M. Kapritsos, "{IronSpec}: Increasing the reliability of formal specifications," in *OSDI*, 2024.
[52] K. Guan and O. Legunsen, "An in-depth study of runtime verification overheads during software testing," in *ISSTA*, 2024.
[53] A. Gyori, S. K. Lahiri, and N. Partush, "Refining interprocedural change-impact analysis using equivalence relations," in *ISSTA*, 2017.
[54] A. Gyori, O. Legunsen, F. Hariri, and D. Marinov, "Evaluating regression test selection opportunities in a very large open-source ecosystem," in *ISSRE*, 2018.
[55] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel, "On-demand test suite reduction," in *ICSE*, 2012.
[56] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression test selection for Java software," in *OOPSLA*, 2001.
[57] J. Hartmann, "30 years of regression testing: Past, present and future," in *PNSQC*, 2012.
[58] K. Havelund, D. Peled, and D. Ulus, "First order temporal logic monitoring with BDDs," in *FMCAD*, 2017.
[59] K. Havelund and G. Roşu, "Monitoring Java programs with Java PathExplorer," in *RV*, 2001.
[60] ——, "Monitoring programs using rewriting," in *ASE*, 2001.
[61] ——, "Synthesizing monitors for safety properties," in *TACAS*, 2002.
[62] "How IntelliJ IDEA profiler works," https://www.jetbrains.com/help/idea/cpu-and-allocation-profiling-basic-concepts.html#how-profiler-works.
[63] "JavaMOP," https://github.com/runtimeverification/javamop.
[64] "Javassist," https://www.javassist.org.
[65] O. Javed and W. Binder, "Large-scale evaluation of the efficiency of runtime-verification tools in the wild," in *APSEC*, 2018.
[66] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu, "Garbage collection for monitoring parametric properties," in *PLDI*, 2011.
[67] D. Jin, P. O. Meredith, C. Lee, and G. Roşu, "JavaMOP: Efficient parametric runtime monitoring framework," in *ICSE Demo*, 2012.
[68] D. Jin, P. O. Meredith, and G. Roşu, "Scalable parametric runtime monitoring," Computer Science Dept., UIUC, Tech. Rep., 2012.

[69] J. A. Jones and M. J. Harrold, "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage," in *ICSM*, vol. 29, 2001.
[70] M. Karaorman and J. Freeman, "jMonitor: Java runtime event specification and monitoring library," in *RV*, 2004.
[71] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *ECOOP*, 2001.
[72] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *ICSE*, 2002.
[73] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky, "Formally specified monitoring of temporal properties," in *ECRTS*, 1999.
[74] B. Korel, L. Tahat, and B. Vaysburg, "Model based regression test reduction using dependence analysis," 2002.
[75] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, "Change impact identification in object oriented software maintenance," in *ICSM*, 1994.
[76] C. Lee, D. Jin, P. O. Meredith, and G. Roşu, "Towards categorizing and formalizing the JDK API," Computer Science Dept., UIUC, Tech. Rep., 2012.
[77] O. Legunsen, D. Marinov, and G. Roşu, "Evolution-aware monitoring-oriented programming," in *ICSE NIER*, 2015.
[78] O. Legunsen, Y. Zhang, M. Hadzi-Tanovic, G. Roşu, and D. Marinov, "Techniques for evolution-aware runtime verification," in *ICST*, 2019.
[79] O. Legunsen, "Evolution-Aware Runtime Verification," Ph.D. dissertation, University of Illinois at Urbana-Champaign, USA, 2019.
[80] 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.
[81] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *FSE*, 2016.
[82] 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.
[83] O. Legunsen, A. Shi, and D. Marinov, "STARTS: STAtic Regression Test Selection," in *ASE*, 2017.
[84] M. Leucker and C. Schallhart, "A brief account of runtime verification," in *FLACOS*, 2007.
[85] Y. Liu, J. Zhang, P. Nie, M. Gligoric, and O. Legunsen, "More precise regression test selection via reasoning about semantics-modifying changes," in *ISSTA*, 2023.
[86] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear, "Verification modulo versions: Towards usable verification," in *PLDI*, 2014.
[87] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Şerbănuţă, and G. Roşu, "RV-Monitor: Efficient parametric runtime verification with simultaneous properties," in *RV*, 2014.
[88] M. Madsen, F. Tip, E. Andreasen, K. Sen, and A. Møller, "Feedback-directed instrumentation for deployed JavaScript applications," in *ICSE*, 2016.
[89] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi, "DiSL: A domain-specific language for bytecode instrumentation," in *AOSD*, 2012.
[90] "Maven is broken by design," https://blog.ltgt.net/maven-is-broken-by-design/.
[91] "Incremental compilation doesn't work unless useIncrementalCompilation is set to 'false'," https://issues.apache.org/jira/browse/MCOMPILER-209.
[92] S. McMaster and A. Memon, "Fault detection probability analysis for coverage-based test suite reductionb," in *ICSM*, 2007.
[93] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the mop runtime verification framework," *IJSTTT*, vol. 14, no. 3, 2012.
[94] B. Miranda, I. Lima, O. Legunsen, and M. d'Amorim, "Prioritizing runtime verification violations," in *ICST*, 2020.
[95] S. Navabpour, C. W. W. Wu, B. Bonakdarpour, and S. Fischmeister, "Efficient techniques for near-optimal instrumentation in time-triggered runtime verification," in *RV*, 2011.
[96] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *FSE*, 2004.
[97] R. Purandare, M. B. Dwyer, and S. Elbaum, "Monitor optimization via stutter-equivalent loop transformation," in *OOPSLA*, 2010.
[98] ——, "Optimizing monitoring of finite state properties through monitor compaction," in *ISSTA*, 2013.
[99] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs."
[100] G. Reger, H. C. Cruz, and D. Rydeheard, "MarQ: Monitoring at runtime with QEA," in *TACAS*, 2015.
[101] G. Reger, S. Hallé, and Y. Falcone, "Third international competition on runtime verification," in *RV*, 2016.
[102] S. P. Reiss, "Tracking source locations," in *ICSE*, 2008.
[103] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *TSE*, vol. 39, no. 5, 2013.
[104] G. Rothermel and M. J. Harrold, "A safe, efficient algorithm for regression test selection," in *ICSM*, 1993.
[105] ——, "A safe, efficient regression test selection technique," *TOSEM*, vol. 6, no. 2, 1997.
[106] "eMOP RPS's no impacted classes bug report," https://github.com/SoftEngResearch/emop/issues/98.
[107] "eMOP RPS's safety issue bug report," https://github.com/SoftEngResearch/emop/issues/97.
[108] "Runtime Verification The Linux Kernel documentation," https://docs.kernel.org/trace/rv/runtime-verification.html.
[109] "A list of RV tools," https://tinyurl.com/yc2bzcs6.
[110] "incremental compiles seems broke in 4.2.0," https://github.com/davidB/scala-maven-plugin/issues/364.
[111] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *FSE*, 2014.
[112] A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen, "Reflection-aware static regression test selection," in *OOPSLA*, 2019.
[113] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *FSE*, 2015.
[114] A. Shi, P. Zhao, and D. Marinov, "Understanding and improving regression test selection in continuous integration," in *ISSRE*, 2019.
[115] C. Soueidi and Y. Falcone, "Bridging the gap: A focused DSL for RV-oriented instrumentation with BISM," in *RV*, 2023.
[116] ——, "Instrumentation for RV: From basic monitoring to advanced use cases," in *RV*, 2023.
[117] C. Soueidi, Y. Falcone, and S. Hallé, "Dynamic program analysis with flexible instrumentation and complex event processing," in *ISSRE*, 2023.
[118] C. Soueidi, M. Monnier, and Y. Falcone, "Efficient and expressive bytecode-level instrumentation for Java programs," *IJSTTT*, vol. 25, no. 4, 2023.
[119] "STARTS—A tool for STAtic Regression Test Selection," https://github.com/TestingResearchIllinois/starts.
[120] L. Teixeira, B. Miranda, H. Rebêlo, and M. d'Amorim, "Demystifying the challenges of formally specifying API properties for runtime verification," in *ICST*, 2021.
[121] "eMOP VMS's unknown line of code bug report," https://github.com/SoftEngResearch/emop/issues/99.
[122] "eMOP VMS's violation in libraries bug report," https://github.com/SoftEngResearch/emop/issues/100.
[123] D. Willmor and S. M. Embury, "A safe regression test selection technique for database driven applications," in *ICSM*, 2005, iCSM.
[124] C. W. W. Wu, D. Kumar, B. Bonakdarpour, and S. Fischmeister, "Reducing monitoring overhead by integrating event- and time-triggered techniques," in *RV*, 2013.
[125] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *STVR*, vol. 22, no. 2, 2012.
[126] A. Yorihiro, P. Jiang, V. Marques, B. Carleton, and O. Legunsen, "eMOP: A Maven plugin for evolution-aware runtime verification," in *RV*, 2023.
[127] K. Zhai, B. Jiang, and W. K. Chan, "Prioritizing test cases for regression testing of location-based services: Metrics, techniques, and case study," *TSC*, vol. 7, no. 1, 2014.
[128] J. Zhang, Y. Liu, M. Gligoric, O. Legunsen, and A. Shi, "Comparing and combining analysis-based and learning-based regression test selection," in *AST*, 2022.
[129] L. Zhang, "Hybrid regression test selection," in *ICSE*, 2018.
[130] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric, "A framework for checking regression test selection tools," in *ICSE*, 2019.