eMOP: A Maven Plugin for Evolution-Aware Runtime Verification

Ayaka Yorihiro, Pengyue Jiang, Valeria Marqués,
Benjamin Carleton, Owolabi Legunsen

Cornell University, USA

`{ay436, pj257, vmm49, bc534, legunsen}@cornell.edu`

Abstract. We present eMOP, a tool for incremental runtime verification (RV) of test executions during software evolution. We previously used RV to find hundreds of bugs in open-source projects by monitoring passing tests against formal specifications of Java APIs. We also proposed evolution-aware techniques to reduce RV's runtime overhead and human time to inspect specification violations. eMOP brings these benefits to developers in a tool that seamlessly integrates with the Maven build system. We describe eMOP's design, implementation, and usage. We evaluate eMOP on 676 versions of 21 projects, including those from our earlier prototypes' evaluation. eMOP is up to 8.4× faster and shows up to 31.3× fewer violations, compared to running RV from scratch after each code change. eMOP also does not miss new violations in our evaluation, and it is open-sourced at `https://github.com/SoftEngResearch/emop`.

1 Introduction

The prevalence of costly and harmful bugs in deployed software underscores the need for techniques to help find more bugs during testing. Runtime verification (RV) [3, 13, 14, 26, 30, 38, 44] is such a technique; it monitors executions against formal specifications and produces violations if a specification is not satisfied.

We previously used RV to amplify the bug-finding ability of tests [32, 34, 40]. We found hundreds of bugs by monitoring passing tests in hundreds of open-source projects against Java API behavioral specifications [31]. Such specifications should not change as client programs evolve. For example, monitoring the `Collections_SynchronizedCollection` specification [7] revealed several bugs, e.g., [8, 9]: possible "non-deterministic behavior" caused by not synchronizing on iterators over `Collection.synchronizedCollection()`'s output [27]. Developers confirmed and fixed these and many other bugs that RV helped us find.

We also found that RV incurs runtime overhead and requires a lot of human time to inspect violations. To reduce RV costs, we proposed three evolution-aware techniques that focus RV and its users on code affected by changes [35, 37]:

(1) **Regression Property Selection (RPS)** re-checks, in a new code version, a subset of specifications that may be violated in code affected by changes.
(2) **Violation Message Suppression (VMS)** displays new violations—users are more likely to deal with violations that are related to their changes [41].
(3) **Regression Property Prioritization (RPP)** monitors important specifications on developers' critical path; others are monitored in the background.

These evolution-aware techniques reduce RV costs, but our proof-of-concept prototypes are hard to integrate with open-source projects. Also, these techniques can be used together but our prototypes do not allow users to easily do so.

We present eMOP, a Maven plugin for incremental RV of tests during software evolution. Maven is a popular build system, so eMOP can bring the benefits of RV to a wider audience of developers. eMOP improves RPS and RPP, re-implements VMS, and allows users to easily combine RPS, VMS, or RPP.

Components in eMOP's architecture (1) extend Maven's surefire plugin [48] to perform analysis before and after monitoring tests; (2) use STARTS [36] to reason about code changes and find classes affected by changes; (3) re-configure a JavaMOP [29] Java agent *on the fly* to select which specifications to monitor and where to instrument them; and (4) get fine-grained change information from Git for computing new violations. Once installed, users only need to change few Maven configuration lines to start using RPS, VMS, RPP, or their combination.

We evaluate eMOP on 676 versions of 21 projects. On the subset of projects and their versions that we previously used to evaluate our prototypes, eMOP produces similar results. Overall, on average, eMOP is up to 8.4× faster (average: 4.0×) and shows up to 31.3× fewer violations (average: 11.8×), compared to using JavaMOP to perform RV from scratch after each code change.

We defined an evolution-aware RV technique as *safe* if it finds all new violations after a change, and *precise* if it finds only new violations [37]. Also, we proposed two sets of RPS variants: two variants that are *theoretically safe* and ten variants that are not, under the assumptions that we make.

Our prior evaluation [37] showed that all RPS variants (including theoretically unsafe ones) were *empirically safe*. But, on projects that we did not previously evaluate, we initially find that our theoretically unsafe RPS variants are not empirically safe if 3rd-party libraries change. So, to improve RPS safety, we rerun RV from scratch when libraries change. We find that RPS with eMOP is empirically safe in all projects and versions that we evaluate in this paper.

Table 1: eMOP vs. our early prototypes [37].

Feature	Prototype	eMOP
Maven integration	✗	✓
Single-module projects	✓	✓
Multi-module projects	✗	✓/✗
RPS	✓	✓
VMS	✓	✓
RPP	✓	✓
RPS + VMS	✗	✓
RPP + VMS	✗	✓
RPS + RPP	6 variants	12 variants
RPS + RPP + VMS	✗	✓
Safe w.r.t. CUT	✓	✓
Safe w.r.t. 3rd-party lib	✗	✓
Version comparison	jDiff	jGit
Ease of configuration	low	high

Comparison with our previous prototypes. Table 1 compares eMOP with our original prototypes [37]. ✓ means "supported"; ✗ means "not supported", and ✓/✗ means "partially supported". Unlike eMOP, our prototypes (1) do not integrate with Maven or work on multi-module projects; (2) partially support combining RPS and RPP, but do not combine RPS or RPP with VMS; (3) are less safe when 3rd-party libraries change; (4) use an external tool to re-obtain change information that is already in Git; and (5) are hard to configure. eMOP will aid future evolution-aware RV research; it is on GitHub [12], as are our artifacts [11].

2 eMOP

We summarize evolution-aware RV, and our eMOP implementation. Our original paper [37] has theoretical background, examples, definitions, diagrams, etc.

2.1 Evolution-Aware RV Techniques

Regression Property Selection (RPS). The inputs to RPS are the old and new program versions, and the set of Java API specifications. The outputs are *affected* specifications that may be violated in code affected by changes. RPS uses class-level static change-impact analysis [36] to find *impacted* classes that transitively depend on changed code. Then, RPS analyzes impacted classes together with all available specifications, and outputs specifications involving API methods that are called in impacted classes.

There are 12 RPS variants that differ in how impacted classes are computed (three options), and where affected specifications are instrumented (four options) [37]. Let Δ be the set of changed classes. Impacted classes are computed in three ways—(a) ps_3: Δ and its *dependents*—classes that use or extend those in Δ; (b) ps_2: classes in (a) plus *dependees*—classes that those in Δ use or extend; and (c) ps_1: classes in (b) plus *dependees of Δ's dependents*. Impacted classes are always instrumented, but there are two Boolean options (and four ways to combine them) for whether to instrument unimpacted classes or 3rd-party libraries. Theoretically, these variants differ in how much safety they trade off for efficiency. But, all variants were empirically safe in our original evaluation [37].

Violation Message Suppression (VMS). To reduce human time for inspecting specification violations, VMS aims to show only new violations after code changes. VMS does not reduce RV runtime overhead. The rationale behind VMS is that developers are more likely to look at and debug new violations, compared to looking at all old and new violations at the same time [41]. VMS takes the set of all violations in the new version, filters out those for which there is evidence that they are old violations, and presents the rest to the user as new violations.

Regression Property Prioritization (RPP). The goal is to reduce the time to see important violations, so users may react faster. RPP splits RV into two phases. Important specifications, defined by the user, are monitored in the *critical phase* and any violations of those specifications are reported immediately. The other specifications are monitored in a *background phase* that users do not have to wait for. Users can decide when and how violations from the background phase are presented or when specifications should be automatically promoted (demoted) from (to) the background phase.

2.2 Implementation

We implement 12 RPS variants, VMS, and RPP in our eMOP Maven plugin. We choose Maven (1) so users can more easily integrate evolution-aware RV, (2) to

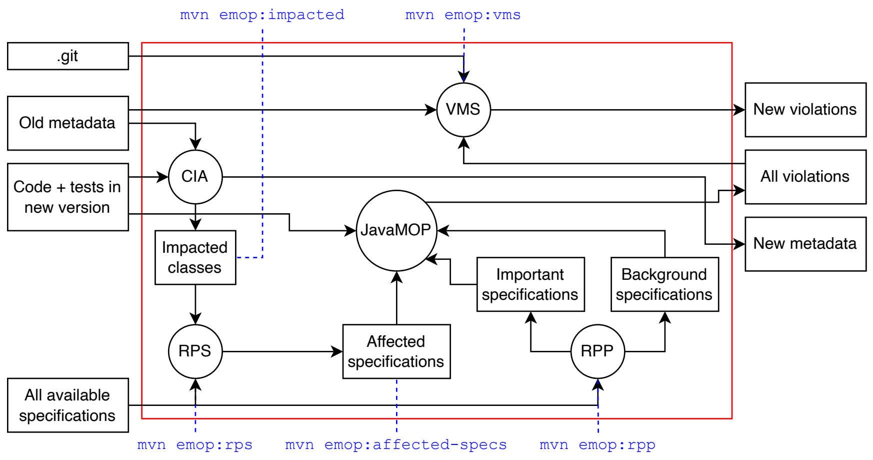

Fig. 1: eMOP's architecture.

ease evolution-aware RV usage during testing, and (3) because we built Maven plugins before [21, 36]. Future work can add eMOP to other build systems.

Figure 1 shows eMOP's architecture, and how components map to some available user commands (see §3). There, ovals show processes and rectangles show data. "Old Metadata" and "New Metadata" contain a per-class mapping from non-debug related bytecode to checksums computed from the old and new code versions, respectively, the classpath, and checksums for the jars on the classpath. eMOP uses STARTS [36] to compute these mappings and classpath information is used to detect changes in 3rd-party libraries. Also ".git" is Git's internal database of changes, which eMOP uses to find which violations are likely new. "CIA" represents our modified change impact analysis in STARTS; eMOP uses "CIA" to compute impacted classes in three ways (§2.1). Lastly, eMOP invokes "JavaMOP" to monitor test executions.

RPS. We invoke the AspectJ compiler, `ajc` [1], to statically analyze which specifications are related to impacted classes. JavaMOP specifications are written in an AspectJ dialect, so our static analysis outputs, as affected, specifications that `ajc` compiles into any impacted class. To reduce analysis cost, eMOP invokes `ajc` on stripped-down specifications that contain only method-related information. Finally, based on `ajc`'s output, eMOP modifies a JavaMOP agent on the fly to only monitor affected specifications and instrument them in locations required by the RPS variant.

VMS. We re-implement VMS on top of JGit [28] (instead of jDiff). JGit provides an API for extracting fine-grained information from ".git". By default, eMOP takes the most recent Git commit as the old version and the current working tree as the new version. So, VMS users can check if code changes introduce new violations before making a commit. Users can also specify what commit to compare the working tree against, or the ID of any two commits.

In the first VMS run, all violations are new. Subsequently, VMS analyzes all violations from the old version against the code change. If a specification is violated in the same class and on a line that is mapped to the same location in both versions, VMS filters it out as old; the rest are presented as new violations. VMS users can choose to write new or old violations to the console, a file on disk, or both.

RPP. Users can provide a file containing important specifications for RPP to monitor in the critical phase; the rest are monitored in the background. Users can also provide a file containing important specifications and a file containing a disjoint set of specifications to be monitored in the background phase. If users do not provide a file, RPP uses the following default scheme. The first time, RPP monitors all specifications in the critical phase. Only specifications that are violated in the first run are monitored in the critical phase in the second run; the rest are monitored in the background. Subsequently, a specification that is violated in the background phase is promoted to the critical phase in the next run. Users can manually demote specifications from the critical phase. Future work can add options to let users specify when a specification should be automatically promoted to or demoted from the critical phase. RPP currently does not support multi-module projects.

Combinations. eMOP users can combine RPS, VMS, and RPP. When using all three together, RPS first finds affected specifications, then RPP splits the monitoring of affected specifications into critical and background phases before VMS shows new violations from RPP's critical phase. When using VMS with RPS or RPP, the other techniques are run first, then VMS shows new violations. Running RPS with RPP works in the same order as when combining all three.

3 Installation and Usage

Installing eMOP. eMOP can be installed by following the directions in the "Installation" section of the `README.md` file on eMOP's GitHub page [12]. Note that installing eMOP from sources requires satisfying all prerequisites that are listed on that GitHub page.

Integrating eMOP. To use eMOP in a Maven project, modify that project's configuration file—typically called `pom.xml`—to add the latest version of eMOP and the JavaMOP agent argument to the configuration of the Maven surefire plugin (which runs tests) file. The current way to add the eMOP to a project is to modify the `pom.xml` file like so:

```
<build><plugins>
  ...
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.20-or-greater</version>
    <configuration> <argLine>-javaagent:${JavaMOPAgent.jar}</argLine>
  </configuration>
  </plugin>
  <plugin>
    <groupId>edu.cornell</groupId>
    <artifactId>emop-maven-plugin</artifactId>
    <version>${latest_eMOP_version}</version>
    </plugin>
  ...
</plugins></build>
```

Using eMOP. These commands allow users to: (1) list impacted classes or affected specifications, or (2) run RPS, VMS, RPP, and their combinations:

```
$ mvn emop:help # list all goals (commands)
$ mvn emop:impacted # list impacted classes
$ mvn emop:affected-specs # list affected specifications
$ mvn emop:rps # run RPS
$ mvn emop:rpp # run RPP
$ mvn emop:vms # run VMS
$ mvn emop:rps-rpp # run RPS+RPP
$ mvn emop:rpp-vms # run RPP+VMS
$ mvn emop:rps-vms # run RPS+VMS
$ mvn emop:rps-rpp-vms # run RPS+RPP+VMS
$ mvn emop:clean # delete all metadata
```

The `emop:help` command lists all eMOP commands and what they do; the others are related to evolution-aware RV. Next, we describe some configuration options.

Running RPS. Three flags choose among RPS variants: (1) `closureOption` specifies how to compute impacted classes: `PS1`, `PS2`, or `PS3` (the default: `PS3`); (2) `includeLibraries` controls whether to instrument 3rd-party libraries (default: `true`); and (3) `includeNonAffected` controls whether non-impacted classes are instrumented (default: `true`). For example, this command runs RPS and instruments neither classes that are not impacted nor 3rd-party libraries:

```
$ mvn emop:rps -DincludeLibraries=false -DincludeNonAffected=false
```

Running VMS. Users can see violations on the console or in a `violation-counts` file. By default, only new violations are shown. But, users can view all violations in the console or in the file via Boolean `showAllInConsole` and `showAllInFile` options, respectively (default: `false`). Users can specify commit IDs using `lastSha` and `newSha`. For example, this command shows new violations relative to commit ID `abc123` in the console but it still outputs all violations to file:

```
$ mvn emop:vms -DlastSha=abc123 -DshowAllInFile=true
```

Running RPP. Users can provide specifications to RPP using two options named `criticalSpecsFile` and `backgroundSpecsFile`. If only `criticalSpecsFile` is provided, then all other specifications will be monitored in the background. By default, RPP tracks metadata for critical and background phase specifications as described in §2.1. But RPP also has a `demoteCritical` option (default: `false`) for demoting previously important specifications that are not violated in the critical phase of the current run to the background phase for the next run. For example, this command monitors specifications in `critical.txt` (respectively, `background.txt`) in the critical (respectively, background) phase:

```
$ mvn emop:rpp -DcriticalSpecsFile=critical.txt -DbackgroundSpecsFile=background.txt
```

Running Combinations. Options in the union of those from combined techniques can be used, e.g., this command runs RPS+RPP using the RPS variant that instruments neither classes that are not impacted nor 3rd-party libraries, while also demoting specifications during RPP:

```
$ mvn emop:rps-rpp -DincludeLibraries=false -DincludeNonAffected=false -DdemoteCritical=true
```

4 Evaluation

Setup. We evaluate eMOP on 21 Maven projects from our previous and ongoing work on RV and regression testing. They are all single-module Maven projects (RPP does not yet support multi-module projects). We use between 11 and 50 versions per project, for a total of 676 versions. Table 2 shows project names (click or hover to see GitHub URL), number of versions that we evaluate (sha#), sizes (KLOC), number of test classes in the first version (TC), average test time (test[s]), and average JavaMOP overhead for these versions (mop).

Table 2: Projects that we evaluate.

Project	sha#	KLOC	TC	test[s]	mop
jgroups-aws	11	0.3	1	3.3	12.5
Yank	17	0.6	18	3.1	7.2
java-configuration-impl	24	1.7	5	3.9	7.0
embedded-jmxtrans	50	3.1	15	4.1	12.3
jbehave-junit-runner	50	1.5	17	4.7	15.2
compile-testing	50	7.0	25	6.2	6.0
javapoet	20	8.1	18	9.1	6.6
exp4j	50	3.5	5	9.3	3.8
joda-time	50	93.3	158	10.8	3.9
jnr-posix	50	11.8	24	11.9	11.9
imglib2	20	33.0	81	13.0	2.5
HTTP-Proxy-Servlet	50	1.0	1	13.3	3.2
smartsheet-java-sdk	21	7.9	51	13.5	4.5
zt-exec	15	2.6	24	14.6	1.9
commons-imaging	20	44.5	107	18.4	4.9
jscep	50	3.2	50	18.9	3.7
commons-lang	20	80.7	172	21.5	3.1
datasketches-java	48	41.9	178	35.8	2.3
commons-dbcp	20	11.4	43	53.0	1.8
stream-lib	20	4.7	28	91.0	6.9
commons-io	20	32.7	114	102.1	4.1

We use the same versions for projects that we evaluated in our original paper [37]. To choose versions for the other projects, we iterate over the 500 most recent versions in each project (most recent first) and terminate when we have tried all 500 or found 50 versions that change at least one Java file, compile, tests pass, JavaMOP does not fail, and JavaMOP time is at least 20 seconds. The versions that we evaluate per project are in our artifact repository [11].

For RPS, we measure the time and the number of unique violations per variant. For VMS, we measure the number of new and total violations per version. For RPP, we measure the critical and background phase times. All overheads for RPS are computed from end-to-end times including time for compilation, analysis, running tests, and monitoring. RPP overhead is only for the critical phase. We run eMOP on 193 revisions (7 no longer compile) of 10 projects from our original paper [37], using the same experimental settings as before. We also run all eMOP variants on all 21 projects using Amazon EC2 C5.4xlarge instances.

Results: comparing with prior evaluation. Solid bars in Figure 2 show average overheads of JavaMOP and RPS variants for the 10 projects in our prior evaluation [37]. RV overhead is t_{rv}/t_{test}; t_{rv} and t_{test} are times with and without JavaMOP, respectively. §2 describes ps_1, ps_2, and ps_3; "ℓ" and "c" mean 3rd-party libraries and non-impacted classes, respectively, are not instrumented.

In Figure 2, all RPS variants reduce the average JavaMOP overhead, which is 7.2× when our evolution-aware techniques are not applied. As expected, based on how we designed these variants, ps_1 incurs the most RPS overhead (5.7×), while $ps_3^{c\ell}$ incurs the least overhead (1.8×).

In general, excluding 3rd-party libraries has a significant effect on reducing RPS overhead, as seen for example in the difference between ps_1 (5.7×) and ps_1^{ℓ} (3.6×) in Figure 2. For libraries that do not depend on 3rd-party libraries, or

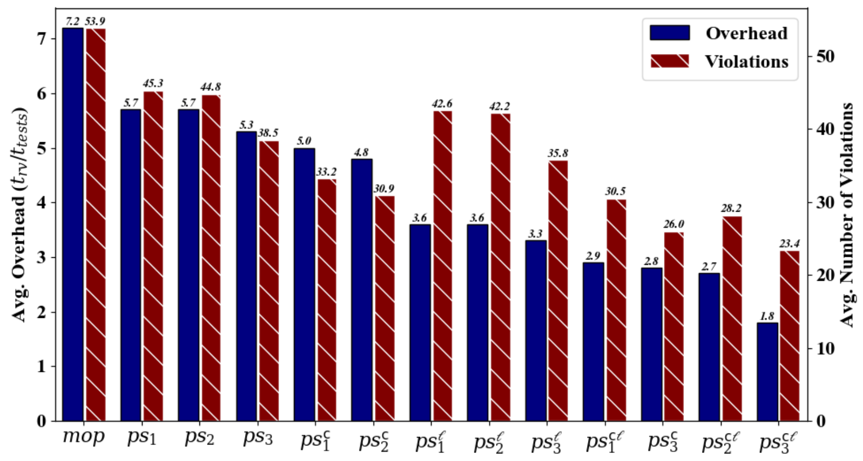

Fig. 2: Runtime overheads of, and violations from, JavaMOP and RPS variants in eMOP for projects and versions in our original evolution-aware RV paper [37].

those that depend on libraries that do not use API methods related to monitored specifications, library exclusion has negligible effect.

Striped bars in Figure 2 show average numbers of unique violations per version across these projects. RPS reduces the number of violations reported, but in the best case ($ps_3^{c\ell}$) it still shows an average of 23.4 violations—most of which are old—after every code change. So, a technique like VMS is needed that reports only the few new violations. The overheads and violations in Figures 2 follow the same trends across RPS variants as in our original paper [37]. So, we are more confident in how we implemented evolution-aware RV in eMOP. Tooling and Maven overheads likely explain any differences with our old results.

Results on more subjects and versions than prior work. We discuss details about how eMOP performs when all three evolution-aware RV techniques are combined, and then discuss the contribution of each technique.

Striped bars in Figure 3 show average overheads per project when all RPS variants are combined with RPP and VMS. The projects in Figure 3 include some that were not in Figure 2, and some in Figure 2 are not in the figure because of dependency issues when we moved experiments to the cloud. We show the RPS variants in Figure 3 in decreasing order of average overheads.

In terms of total time incurred (not shown in Figure 3), the best-performing variant in RPS+RPP+VMS is $ps_3^{c\ell}$ (it does not instrument 3rd-party libraries or classes that are not impacted by changes). Comparing the striped mop and $ps_3^{c\ell}$ bars shows that, on average across these projects and versions, $ps_3^{c\ell}$ reduces RV overhead by roughly 4.0× (a very rough estimate, since we take a mean of means). The project with the biggest speedup saw a 8.4× reduction in overhead, from 15.2× with JavaMOP to 1.8× with $ps_3^{c\ell}$.

RPP Contributions to the Combination. Solid bars in Figure 3 show how well RPS performs on its own. Comparing the mop and $ps_3^{c\ell}$ bars shows that RPS reduces RV overhead by 2.4× when used alone. We elide per-project details for

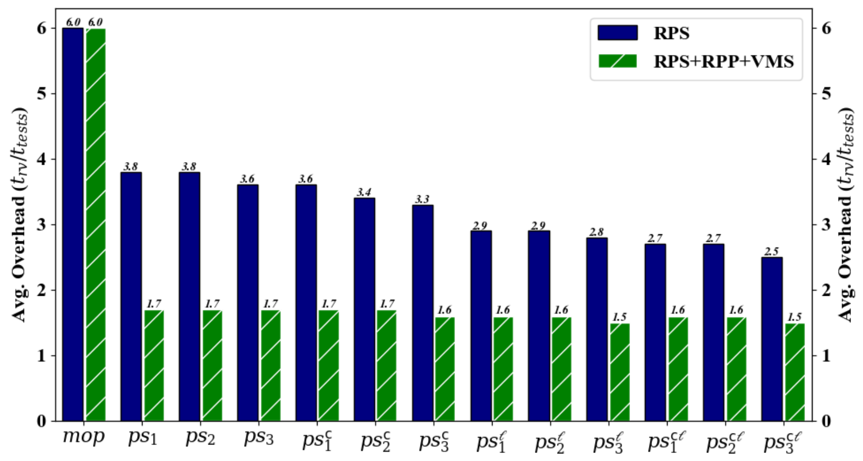

Fig. 3: Average runtime overheads of all RPS variants in eMOP when run alone, and when each RPS variant is combined with RPP and VMS.

lack of space, but we discuss two observations. First, the project that benefits the most from using only RPS had a 4.9× overhead reduction (from 6.9× to 1.4×). Second, the further reduction of average RV overhead resulting from combining RPP with RPS can be seen by comparing the solid and striped bars in Figure 3. Doing so shows that RPP's critical phase, plus RPS incurs less overhead than using RPS alone.

Table 3: Unique violations from JavaMOP (not evolution-aware), and VMS.

Project	sha#	mop (sum)	VMS (sum)	mop (avg)	VMS (avg)
jgroups-aws	11	12	12	1.1	1.1
Yank	17	22	8	1.3	0.5
java-configuration	24	68	8	2.8	0.3
embedded-jmxtrans	50	627	19	12.5	0.4
jbehave-junit-runner	50	248	9	5.0	0.1
compile-testing	50	541	48	10.8	1
javapoet	20	180	9	9	0.5
exp4j	50	0	0	0	0
joda-time	50	1100	43	22	0.9
jnr-posix	50	1659	72	33.2	1.4
imglib2	20	1540	78	77	3.9
HTTP-Proxy-Servlet	50	929	66	11.6	1.3
smartsheet-java-sdk	21	598	300	28.5	14.3
zt-exec	15	15	2	1	0.1
commons-imaging	20	1064	57	53.2	2.9
jscep	50	1706	213	34.1	4.3
commons-lang	20	1220	61	61.0	3.0
datasketches-java	48	96	34	2	0.7
commons-dbcp	20	40	2	2	0.1
stream-lib	20	300	16	15	0.8
commons-io	20	1795	94	89.8	4.7
Across projects	676	13760	1151	20.2	1.7

VMS Results. Table 3 shows VMS results; "sha#" is the number of versions that we evaluate per project, the "mop" and "VMS" columns show the sum and average of violations found per version. Recall that VMS does not reduce RV runtime overhead; rather, it aims to show only new violations. We find that using VMS shows much fewer violations than RPS or JavaMOP. Specifically, across all evaluated projects (see "Across projects" row), VMS shows only 1.7 violations per version, compared to 20.2 violations with JavaMOP—an average reduction of 11.8×. The project with the most average reduction—31.3×—is `embedded-jmxtrans`. Fractional values that are less than 1.0 in the per project average rows show the number of violations shown every ten versions, on average. For example, in `jbehave-junit-runner`, VMS shows an average of one violation in every ten versions, but JavaMOP shows five violations per version.

Our manual analysis shows that all RPS variants are safe—they do not miss any new violation that VMS reports. (Like in our original paper, we assume that VMS reports all new violations.) These results on VMS and safety are in line with findings from our original paper. So, users will likely feel less swamped by a deluge of violations that RV shows if run from scratch after every change.

Limitations. eMOP only supports JUnit; it does not yet work for other testing frameworks, e.g., TestNG. eMOP's bytecode instrumentation sometimes clashes with the instrumentation that open-source projects already use for non-RV reasons. Non-trivial engineering is needed to make instrumentation compatible. We evaluated eMOP on 161 Java API specifications that are commonly used in RV research. As more specifications are added, more optimizations will likely be needed. eMOP uses JGit to map lines from old to new versions, so a few old violations can still be presented as new. More precise change-impact analyses, such as semantic differencing [20] can be investigated and added as an option in the future. eMOP's use of a static change-impact analysis leads to two limitations. First, eMOP may be unsafe if it does not find classes that are impacted by the changes due to the use of dynamic features like reflection. Second, it is possible that the set of impacted classes would be more precise if analysis is done at the method-level instead. eMOP may not work as-is for other kinds of specifications than the kinds of API-level specifications that we check. Lastly, eMOP does not yet control for test flakiness [5, 42, 45, 47] or non-determinism.

Related Work. Researchers proposed many other RV tools other than JavaMOP, e.g., [10, 22–25, 43]. eMOP is the first to integrate evolution-aware RV techniques into a popular build system. Evolution-awareness is not unique to JavaMOP; future work can make other tools evolution aware. Tools for offline RV exist, e.g., [4]. It is not yet clear how to make offline RV evolution aware. Plugins helped make non-RV techniques easier to use. For example, Evosuite [15, 16] is a test generation technique that seemed to gain more popularity after plugins for Maven, Eclipse, and IntelliJ were developed [2]. Also, after decades of research on regression test selection (RTS) [6, 18, 19, 33, 39, 46], RTS plugins that are integrated with Maven or Ant [17, 36] led to recent adoption of RTS tools among developers and renaissance in RTS research.

5 Conclusions and Future Work

eMOP brings the benefits of evolution-aware RV to Maven. We find that eMOP reduces RV costs and makes it easier to use RV during regression testing. We plan to evaluate eMOP on more projects, address some of its limitations, and implement more features. eMOP is open-sourced; we hope that it will provide a platform for advancing the research on integrating software testing and RV.

Acknowledgements. We thank the anonymous reviewers for their comments on an earlier draft of this paper. This work was partially supported by funds from the Google Cyber NYC Institutional Research Program and the US National Science Foundation under Grant Nos. 2019277 and 2045596.

References

1. ajc. https://www.eclipse.org/aspectj/doc/next/devguide/ajc-ref.html.
2. A. Arcuri, J. Campos, and G. Fraser. Unit test generation during software development: Evosuite plugins for Maven, IntelliJ, and Jenkins. In *ICST*, pages 401–408, 2016.
3. E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. Introduction to runtime verification. In *Lectures on Runtime Verification: Introductory and Advanced Topics*, pages 1–33. 2018.
4. D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu. MONPOLY: Monitoring usage-control policies. In *RV*, pages 360–364, 2012.
5. J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. DeFlaker: Automatically detecting flaky tests. In *ICSE*, pages 433–444, 2018.
6. S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran. Regression test selection techniques: A survey. *Informatica*, 35(3):289–321, 2011.
7. Collections_SynchronizedCollection Specification. https://github.com/owolabileg/property-db/blob/master/annotated-java-api/java/util/Collections_SynchronizedCollection.mop.
8. SuiteHTMLReporter does not synchronize iteration on a synchronized list. https://github.com/testng-team/testng/pull/931.
9. JUnitXMLReporter does not synchronize the two synchronized collections when iterating. https://github.com/testng-team/testng/pull/830.
10. J. Ellul and G. J. Pace. Runtime verification of ethereum smart contracts. In *EDCC*, pages 158–163, 2018.
11. eMOP Artifacts. https://github.com/SoftEngResearch/emop-artifacts.
12. eMOP GitHub Page. https://github.com/SoftEngResearch/emop.
13. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In *EDSS*, pages 141–175. 2013.
14. Y. Falcone, S. Krstić, G. Reger, and D. Traytel. A taxonomy for classifying runtime verification tools. In *Runtime Verification*, pages 241–262, 2018.
15. G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *FSE*, pages 416–419, 2011.
16. G. Fraser and A. Arcuri. A large-scale evaluation of automated unit test generation using Evosuite. *TOSEM*, 24(2):1–42, 2014.
17. M. Gligoric, L. Eloussi, and D. Marinov. Ekstazi: Lightweight test selection. In *ICSE Demo*, pages 713–716, 2015.
18. M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *ISSTA*, pages 211–222, 2015.
19. T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *TOSEM*, 10(2):184–208, 2001.
20. A. Gyori, S. K. Lahiri, and N. Partush. Refining interprocedural change-impact analysis using equivalence relations. In *ISSTA*, pages 318–328, 2017.
21. A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov. NonDex: a tool for detecting and debugging wrong assumptions on Java API specifications. In *FSE Demo*, pages 993–997, 2016.
22. S. Hallé and R. Khoury. Event stream processing with BeepBeep 3. In *RV-CuBES*, pages 81–88, 2017.
23. K. Havelund. Rule-based runtime verification revisited. *STTT*, 17:143–170, 2015.
24. K. Havelund and D. Peled. Efficient runtime verification of first-order temporal properties. In *SPIN*, pages 26–47, 2018.

25. K. Havelund, D. Peled, and D. Ulus. First-order temporal logic monitoring with BDDs. *FMSD*, 56(1-3):1–21, 2020.
26. K. Havelund and G. Roşu. Monitoring programs using rewriting. In *ASE*, pages 135–143, 2001.
27. java.util.Collections. `https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html`.
28. JGit. `http://www.eclipse.org/jgit`.
29. D. Jin, P. O. Meredith, C. Lee, and G. Roşu. JavaMOP: Efficient parametric runtime monitoring framework. In *ICSE Demo*, pages 1427–1430, 2012.
30. M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: A run-time assurance tool for Java programs. In *RV*, pages 218–235, 2001.
31. C. Lee, D. Jin, P. O. Meredith, and G. Roşu. Towards categorizing and formalizing the JDK API. Technical report, Computer Science Dept., UIUC, 2012.
32. O. Legunsen, N. Al Awar, X. Xu, W. U. Hassan, G. Roşu, and D. Marinov. How effective are existing Java API specifications for finding bugs during runtime verification? *ASEJ*, 26(4):795–837, 2019.
33. 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*, pages 583–594, 2016.
34. 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*, pages 602–613, 2016.
35. O. Legunsen, D. Marinov, and G. Rosu. Evolution-aware monitoring-oriented programming. In *ICSE NIER*, pages 615–618, 2015.
36. O. Legunsen, A. Shi, and D. Marinov. STARTS: STAtic Regression Test Selection. In *ASE Demo*, pages 949–954, 2017.
37. O. Legunsen, Y. Zhang, M. Hadzi-Tanovic, G. Rosu, and D. Marinov. Techniques for evolution-aware runtime verification. In *ICST*, pages 300–311, 2019.
38. M. Leucker and C. Schallhart. A brief account of runtime verification. In *Formal Languages and Analysis of Contract-Oriented Software*, pages 293–303, 2007.
39. Y. Liu, J. Zhang, P. Nie, M. Gligoric, and O. Legunsen. More precise regression test selection via reasoning about semantics-modifying changes. In *ISSTA*, pages 664–676, 2023.
40. B. Miranda, I. Lima, O. Legunsen, and M. d'Amorim. Prioritizing runtime verification violations. In *ICST*, pages 297–308, 2020.
41. P. W. O'Hearn. Continuous reasoning: Scaling the impact of formal methods. In *LICS*, pages 13–25, 2018.
42. F. Palomba and A. Zaidman. Does refactoring of test smells induce fixing flaky tests? In *ICSME*, pages 1–12, 2017.
43. G. Reger, H. C. Cruz, and D. Rydeheard. MarQ: monitoring at runtime with QEA. In *TACAS*, pages 596–610, 2015.
44. F. B. Schneider. Enforceable security policies. *TISSEC*, 3(1):30–50, 2000.
45. A. Shi, A. Gyori, O. Legunsen, and D. Marinov. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST*, pages 80–90, 2016.
46. A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen. Reflection-aware static regression test selection. *PACML*, 3(OOPSLA):1–29, 2019.
47. A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *FSE*, pages 545–555, 2019.
48. About surefire. `https://maven.apache.org/surefire`.