eMOP: An Evolution-Aware Runtime Verification Tool

Ayaka Yorihiro[1], Pengyue Jiang[2], Valeria Marqués[1], Benjamin Carleton[1], Owolabi Legunsen[1]
[1] Cornell University, Ithaca, NY 14850, USA
[2] University of Illinois Urbana-Champaign, Urbana, IL 61801, USA

***Abstract*—We present eMOP, the first tool for performing incremental runtime verification (RV) as software evolves. RV monitors program runs against formal specifications. We showed that RV amplifies the bug-finding ability of tests. We also proposed three evolution-aware techniques that reduce RV's runtime overhead to monitor specifications and human time to inspect specification violations. But, there is no tool today that developers and researchers can use to obtain all these benefits. eMOP fills that gap. We describe eMOP's design and implementation, and how to use it. We implement eMOP as a plugin that can be used to perform evolution-aware RV on Java projects that use the popular Maven build system. Our evaluation on 1,719 versions of 48 open-source projects shows that eMOP is on average 4× (max: 12.8×) faster and shows 43.5× fewer violations, compared to performing RV from scratch after each code change. eMOP is open-sourced; a video demo is at https://www.youtube.com/watch?v=wivOHrL0Xow.**

I. Introduction

Testing is widely used to check code quality, but it misses bugs. So, techniques for finding more bugs during testing are needed. Runtime Verification (RV) [5] is such a technique; it monitors program runs against formal specifications and produces violations if a specification is not satisfied.

We showed that RV amplifies the bug-finding ability of tests—it helped find hundreds of additional bugs from passing tests in many open-source projects [8]. But, RV incurs high costs in machine time to monitor specifications and human time to inspect specification violations. To reduce RV costs, we proposed three evolution-aware techniques that focus RV and its users on code that is affected by changes [9], [11]:

(1) **Regression Property Selection (RPS)** only re-checks, in a new code version, a subset of all specifications that may be violated in code that is affected by changes.
(2) **Violation Message Suppression (VMS)** displays only new violations to avoid overwhelming users with violations that are not related to their changes.
(3) **Regression Property Prioritization (RPP)** first monitors specifications that the user deems to be more important and then monitors the rest in the background.

Our evolution-aware techniques can lower RV costs, but there is no tool today that developers and researchers can easily use to obtain the benefits of RPS, VMS, and RPP on their projects.

We present eMOP, the first tool for evolution-aware RV. eMOP makes it easier to use RV during regression testing, reduces RV costs, and works in a modern development environment. We implement eMOP as a plugin that can be used on Java projects that use the popular Maven build system. eMOP adds evolution-awareness to JavaMOP, a widely-used and widely-cited RV framework [7].

The architecture of eMOP has several components; it (1) extends Maven's surefire plugin [13] and uses it to control testing; (2) uses STARTS [10] to reason about code changes and find classes that are affected by those changes; (3) reconfigures a JavaMOP Java agent on the fly to select specifications to monitor and where to instrument them; and (4) gets fine-grained information about changed lines of code from Git and uses those to compute which violations may be new.

After installation, a user only has to change some lines in their Maven configuration file and they can immediately start using the RPS, VMS, and RPP commands. Also, eMOP is customizable: the main commands have several options for configuring how RV is performed.

Our evaluation shows that eMOP is effective for reducing RV costs. We run eMOP on 1,719 versions of 48 open-source projects. On average, eMOP is 4× faster (max: 12.8×) and shows 43.5× fewer violations than using JavaMOP to perform RV from scratch after each change.

Comparison with our early prototype. Our earlier prototype [11] cannot be seamlessly integrated into build systems that developers use. Our prototype only works on single-module Maven projects, but eMOP also handles multi-module projects. Lastly, we expanded our evaluation; we only ran our prototype on 20 versions each in 10 projects.

eMOP provides a basis that we and other researchers can build on, towards bringing RV closer to routine usage for finding more bugs during testing. We make eMOP publicly available on GitHub: https://github.com/SoftEngResearch/emop.

II. eMOP: Techniques and Implementation

We summarize the techniques implemented in eMOP. Our original paper [11] has examples, definitions, diagrams, etc.

A. Techniques

Regression Property Selection (RPS). The inputs to RPS are the old and new version of code, and the set of all specifications; it outputs *affected* specifications that may be violated in code whose behavior may be altered by the code changes. RPS works at the class level; it first performs a conservative static change impact analysis on the old and new code versions to find *impacted* classes. Then, it analyzes the impacted classes together with all specifications and outputs those specifications that are related to the impacted classes. RPS is *safe* if it finds all new violations that are introduced by a change, and *precise* if it only finds new violations.

There are 12 variants of RPS, based on how impacted classes are computed (three options), and where affected specifications are instrumented (four options). Let Δ be the

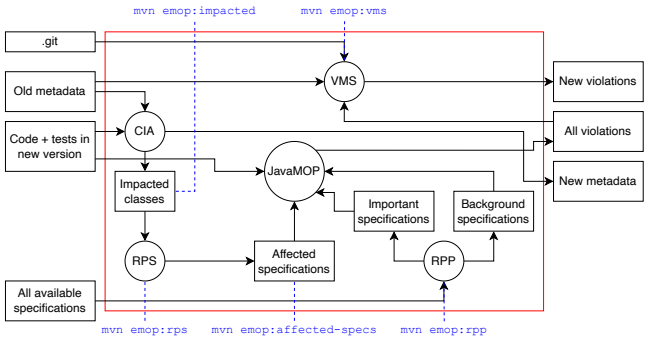

Fig. 1: eMOP architecture.

set of classes that changed. The three sets computed as impacted classes are (a) ps_1: Δ and its *dependents*—classes that use or inherit from those in Δ; (b) ps_2: classes in (a) and *dependees*—classes that those in Δ use or inherit from; and (c) ps_3: classes in (b) and *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 classes that are not impacted by changes or classes in third-party libraries that the program depends on. Theoretically, these variants differ in how much safety they trade off for efficiency. But, all variants were safe in our original evaluation [11].

Violation Message Suppression (VMS). To reduce the time that users spend to inspect specification violations, this technique aims to show only new violations that were not in the old version. 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 [12]. VMS takes the set of all violations in the new version, filters out old violations, and presents the rest to the user as new violations.

Regression Property Prioritization (RPP). The goal here is to reduce the time to see important violations, so that users may react more quickly. 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 immediately reported to the user. The remaining specifications are run in a *background phase* that the user does not have to wait for. Users can decide when and how violations from the background phase are presented. The criteria for selecting important properties can also be used to automatically promote (demote) specifications from (to) the background phase.

B. Implementation

We implement all 12 RPS variants, VMS, and RPP in an eMOP plugin for Maven—a popular build system. We choose Maven (1) so that its users can more easily integrate evolution-aware RV into their environment, (2) to make it easier to use evolution-aware RV during regression testing, and (3) because we have experience building Maven plugins.

Figure 1 shows the architecture of eMOP, and how they map to commands that users can invoke (see Section III). There, ovals represent processes and rectangles represent data. "Old Metadata" and "New Metadata" contain a per-class mapping from non-debug related bytecode to a checksum that was computed from the old and new versions of code, respectively. eMOP uses STARTS to compute these mappings. Also ".git" contains Git's internal database of historical changes; eMOP uses it to find which subset of all violations are new. The "CIA" oval represents changes that we make to STARTS' change impact analysis for eMOP to obtain the three options for computing impacted classes (Section II-A). Lastly, the "JavaMOP" oval represents invocations of JavaMOP agents that are used to monitor the test executions.

RPS. This component programmatically calls the AspectJ compiler, `ajc` [1], to statically analyze which of all available specifications are related to the impacted classes from "CIA". JavaMOP specifications are written in a dialect of AspectJ, so our static analysis is simple: it checks which specifications are compiled into each class and outputs the union of all such specifications as the affected ones. To reduce costs, eMOP invokes `ajc` on stripped down versions of the specifications that only contain class-related information. Finally, after processing `ajc`'s output, eMOP dynamically modifies the JavaMOP agent to only monitor the affected specifications and instrument them in the code locations required by the RPS variant being run.

VMS. This component builds on JGit [6], which provides an API for working with ".git"; future work can extend eMOP to other version control systems. By default, eMOP takes the most recent Git commit to be the old version and the current working tree as the new version. This way, VMS users can check if code changes introduce new violations before making a commit. Users can also specify another commit other than the most recent one to compare the working tree against, or they can specify the ID of any two commits to compare.

When VMS is first run, all violations are presented as new. Subsequently, VMS analyzes all violations from the old version against the "diff". 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 how to display new violations: write to the console, write to a file on disk, or both.

RPP. Users can provide a file containing important specifications, and the RPP component will monitor them in the critical phase; the rest are monitored in the background. Users can also provide one file containing important specifications and another file containing a disjoint set of specifications to be monitored in the background phase. If users provide no file, RPP defaults to the following. The first time RPP is run, all specifications are monitored in the critical phase. Then, specifications that are violated during the first run are subsequently always monitored in the critical phase. If a specification in the background phase is violated, then it is promoted to the critical phase in the next version. Users have an option to demote specifications that are not violated in the critical phase. We plan to add options to let users specify when a specification should be promoted or demoted.

Combinations. Any combination of RPS, VMS, and RPP can be used together. eMOP currently lets users combine RPP with RPS. As we continue to engineer eMOP in the future, we will add all other combinations. We will also support using eMOP with regression test selection (RTS), since we are already invoking STARTS. We did not yet integrate with RTS because our original evaluation shows that it does not reduce RV costs as much as evolution-aware techniques do [11].

III. Installation and Usage

Installing eMOP from source. The latest eMOP sources can be cloned from GitHub [3] and installed by running this command from inside the cloned eMOP directory:

```
$ mvn install
```

Integrating eMOP. To use eMOP, add the latest version of the eMOP plugin along with adding the JavaMOP agent to surefire within that project's Maven configuration, i.e., `pom.xml` file:

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

Using eMOP. The main eMOP goals allow to: (1) list impacted classes, (2) list affected specifications, (3) run RPS, (4) run VMS, (5) run RPP, and (6) run RPS and RPP together:

```
$ mvn emop:help # list all goals
$ mvn emop:impacted # list impacted classes
$ mvn emop:affect # 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:rps-vms # run RPS+VMS
$ mvn emop:clean # delete all metadata
```

The first goal, `emop:help`, lists all other eMOP goals and what they are used for. The other goals are related to evolution-aware RV and we next describe the some of their options.

Running RPS. There are three options that control RPS variant choice: `closureOption`, `includeNonAffected`, and `includeLibraries`. The first option, `closureOption`, accepts three values that specify how to compute impacted classes: `TRANSITIVE_OF_INVERSE_TRANSITIVE`, `TRANSITIVE_AND_INVERSE_TRANSITIVE`, or `TRANSITIVE`. We call these ps_1, ps_2, and ps_3, respectively; the default is ps_3. The other two options are Boolean and they control whether to instrument non-impacted classes (superscript ^{c} means it is `false`), classes in third-party libraries (superscript ^{l} means it is `false`), or both. Both options are `true` by default. This command runs the ps_3^{cl} variant of RPS which does not instrument classes that are not impacted or classes in third-party libraries:

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

Running VMS. Violations can be shown to the user in in the console or the `violation-counts` file. By default, only new violations are shown in both locations. But, users can view all violations in the console or file via the Boolean `showAllInConsole` and `showAllInFile` options, respectively. Also, users can specify a commit ID to compare violations with using the `lastSha` and `newSha` options. This command presents all violations in the file and shows only new violations, relative to commit ID `abc123`, in the console:

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

Running RPS+RPP. When combining techniques, the set of available options is the union of those from each technique. So, for RPS+RPP, any of the 12 RPS variants can be chosen as previously described. Additionally, RPP provides a `demoteCritical` option, which demotes all important specifications that are not violated in the critical phase of the current run to the background phase in the next run; it is `false` by default. Two options allow users to provide specifications to RPP: `criticalSpecsFile` and `backgroundSpecsFile`. This command runs ps_3^{cl}+RPP with demotion:

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

IV. Evaluation

Setup. We evaluate eMOP on 48 Maven projects from our previous and ongoing work on RV and regression testing. They include 42 single-module Maven projects, and 6 multi-module Maven projects, and a full list is in the eMOP repository [2]. We use between 10 and 50 versions from each project, for a total of 1,719 versions. To choose versions, we iterate over the 500 most recent commits in each project and terminate when we have tried all 500 or when we have found 50 versions that change a Java file, compile, tests pass, and JavaMOP does not fail. 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. We run all experiments on an Intel® Xeon® Gold 6348 machine with 512GB of RAM running Ubuntu 20.04.4 LTS, using 96 cores.

Results. The solid bars in Figure 2 show the average overheads of JavaMOP and RPS variants across all versions of the projects that we evaluate. We present RV overhead in multiples ($\times$) as the ratio $t_{mop}\ /t_{test}$, where t_{mop} is the time with JavaMOP, and t_{test} is the time without JavaMOP. All overheads for RPS are computed from end-to-end times including time for analysis, running tests, and monitoring test executions. The overhead for RPP is computed from only the time taken to run the RPP critical phase. Also we only combine ps_3^{cl} with RPP; it is the variant with the lowest overhead.

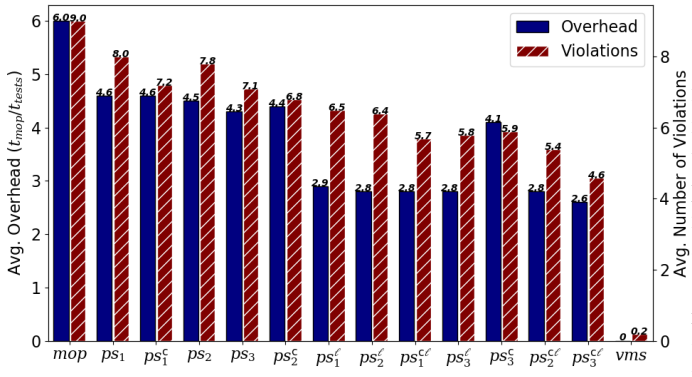

Fig. 2: Runtime overheads of, and violations from JavaMOP and violations from all RPS variants with 161 properties.

All RPS variants reduce the 6.0× average JavaMOP overhead. Variants ps_1 and ps_1^c, both at 4.6×, have the most RPS overhead, while ps_3^{cl} has the least overhead, at 2.6×. Excluding libraries significantly reduces RPS overhead, seen in the difference between ps_1 and ps_1^l, from 4.6× to 2.9×.

Striped bars in Figure 2 show average numbers of unique violations per version across all projects. The *vms* bar shows VMS average. RPS reduces the average number of violations from 9, but a user who inspects all violations must still inspect up to 4.6 violations per version. VMS only shows 0.2 new violations after every change, or one in every five versions. So, using VMS reduces users' manual inspection burden.

Figure 3 shows average overheads for JavaMOP and the critical and background phases, and for ps_3^{cl}+RPP. At 8.1×, RPP's background phase is 5× slower than its critical phase (1.5× overhead). Secondly, like with the prototype [11], the combined overheads of both phases is more than JavaMOP's, largely because tests are run twice in RPP but once with JavaMOP. Lastly, ps_3^{cl}+RPP's overheads in the critical (1.5×) and backgroud (0.8×) phases are lower than RPP's alone: only affected specifications are monitored.

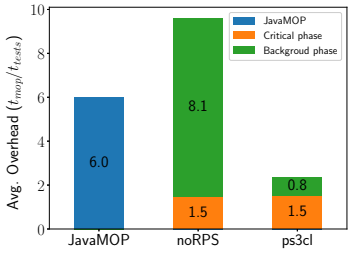

Fig. 3: Overheads for RPP's critical and background phases.

These overheads and violations follow the same trends as in our original paper [11]. But, the average numbers of violations are much lower here, as we evaluate many more projects and many of them have no violations. Also, we note that some projects benefit much more from evolution-awareness than others. For example, the project with the biggest speedup saw reduced overhead from 30.7× with JavaMOP, to 2.4× with ps_3^{cl}. Finally, as expected, not all projects benefit from evolution-awareness, 12 projects take more time on average with eMOP than with JavaMOP because their test times are very small or JavaMOP overheads are not high enough for eMOP analysis costs to be beneficial.

V. Limitations

Currently, eMOP only supports JUnit; it does not yet work for projects that use TestNG to run their tests. When eMOP is integrated with projects that need to instrument bytecode, there is often a clash with the instrumentation that eMOP performs for RV. Non-trivial engineering effort is needed to make instrumentation from multiple tools compatible with one another. We have only evaluated eMOP on the 161 specifications of the Java API that are commonly used in RV research. As more specifications are added, more optimizations will need to be investigated and added. eMOP uses JGit to map lines from old to new versions, so a few old violations can still be presented as new. More precise analyses, such as semantic differencing [4] can be investigated and added as an option in the future. eMOP's use of a static change impact analysis has two implications. 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. Lastly, eMOP does not control for test flakiness or non-determinism.

VI. Conclusions and Future Work

We presented eMOP, the first tool that brings the benefits of evolution-aware RV to Maven-based software development environments. Our evaluation shows that eMOP reduces the costs of RV and makes it easier to use during regression testing. Our future plans for eMOP include evaluating eMOP on more projects, addressing some of the limitations that we discovered, and implementing other features. Now that eMOP is open-source, we hope that it will provide a platform for others to join us in improving it, towards advancing the research goal integrate software testing and RV as a way to find more bugs early.

References

[1] ajc. https://www.eclipse.org/aspectj/doc/next/devguide/ajc-ref.html.

[2] eMOP List of Subject Projects. https://github.com/SoftEngResearch/emop/blob/master/projects.csv.

[3] eMOP GitHub Page. https://github.com/SoftEngResearch/emop.

[4] A. Gyori, S. K. Lahiri, and N. Partush. Refining interprocedural change-impact analysis using equivalence relations. In *ISSTA*, pages 318–328, 2017.

[5] K. Havelund and G. Roşu. Monitoring programs using rewriting. In *ASE*, pages 135–143, 2001.

[6] JGit. http://www.eclipse.org/jgit.

[7] D. Jin, P. O. Meredith, C. Lee, and G. Roşu. JavaMOP: Efficient parametric runtime monitoring framework. In *ICSE Demo*, pages 1427–1430, 2012.

[8] 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.

[9] O. Legunsen, D. Marinov, and G. Rosu. Evolution-aware monitoring-oriented programming. In *ICSE NIER*, pages 615–618, 2015.

[10] O. Legunsen, A. Shi, and D. Marinov. STARTS: STAtic Regression Test Selection. In *ASE Demo*, pages 949–954, 2017.

[11] O. Legunsen, Y. Zhang, M. Hadzi-Tanovic, G. Rosu, and D. Marinov. Techniques for evolution-aware runtime verification. In *ICST*, pages 300–311, 2019.

[12] P. W. O'Hearn. Continuous reasoning: Scaling the impact of formal methods. In *LICS*, pages 13–25, 2018.

[13] About surefire. https://maven.apache.org/surefire.