Evolution-Aware Monitoring-Oriented Programming

Owolabi Legunsen, Darko Marinov, and Grigore Roşu
University of Illinois at Urbana-Champaign
{legunse2, marinov, grosu}@illinois.edu

***Abstract*—Monitoring-Oriented Programming (MOP) helps develop more reliable software by means of monitoring against formal specifications. While MOP showed promising results, all prior research has focused on checking a *single version* of software. We propose to extend MOP to support *multiple software versions* and thus be more relevant in the context of rapid software evolution. Our approach, called *eMOP*, is inspired by regression test selection—a well studied, evolution-centered technique. The key idea in eMOP is to monitor only the parts of code that changed between versions. We illustrate eMOP by means of a running example, and show the results of preliminary experiments. eMOP opens up a new line of research on MOP—it can significantly improve usability and performance when applied across *multiple versions* of software and is complementary to algorithmic MOP advances on a *single version*.**

I. Introduction

In *Monitoring-Oriented Programming (MOP)* [5], a potentially unreliable software is monitored against one or more correctness/safety properties formally specified as in the example in Fig. 1. There, lines 2-5 define two events that are of interest at runtime and when the events should be fired (after `hasNext()` returns `true`, and before calling `next()`), line 6 defines the formal property using linear temporal logic (each call to `next()` must be preceded by a successful call to `hasNext()`), and line 7 allows the user to give code to be executed when the property is violated at runtime. A tool, such as JavaMOP [10], automatically adds instrumentation code in the software where the events take place, which executes monitoring code generated from the formal properties. MOP can thus be used as a lightweight formal method for early bug detection during software development, by providing additional properties/oracles to complement software tests.

Significant advances have been made to improve the practicality of software runtime monitoring and of MOP [1]–[4], [11], [14], [15], targeted at solving two major hindrances: (i) high runtime and/or memory monitoring overhead, and (ii) scarcity of formal properties to monitor. Also, MOP and associated techniques, like property mining, have been useful for finding bugs in well-tested and well-used software [6], [8], [14], [19]–[21]. All these advances considered only *one software version* and did not address the important fact that software continuously evolves. Yet, as shown with other software analysis and testing techniques [12], [23], the accumulated benefits along *multiple versions* of evolving software can make such techniques more practical.

We propose *Evolution-Aware Monitoring-Oriented Programming* (eMOP), the first approach for MOP in evolving systems. eMOP is inspired by *Regression Test Selection (RTS)*,

```
1 Iterator_HasNext(Iterator i) {
2  event hasnexttrue after(Iterator i) returning(boolean b):
3    call(*Iterator+.hasNext()) && target(i)&&condition(b){}
4  event next before(Iterator i):
5    call(*Iterator+.next()) && target(i){}
6  ltl: [](next => (*) hasnexttrue)
7  @violation {...} }
```

Fig. 1: The HasNext property in JavaMOP

which aims to improve the efficiency of regression testing by selecting to re-run only a subset of tests that may be *affected* [22] by code changes between two versions. The idea of eMOP is to make MOP evolution-aware and improve *both* its efficiency and usability by monitoring only the parts of code that changed between versions. The goal is to reduce runtime overhead and show developers only the violations they care about, corresponding to the software changes.

The need for eMOP was shaped during a formative study in which we evaluated JavaMOP on 40 open-source projects to monitor their test execution against 181 MOP properties [14]. Test-running time per project increased from 16.4sec to 232.7sec with 18,810 violations generated, on average, when JavaMOP was used to monitor one version of each project. The overhead and number of violations are similar across multiple versions. It would be beneficial if the runtime overhead, and the violations generated, on a subsequent version of each project are due only to the differences between the two versions. This paper makes the following contributions:

- ⋆ **MOP for evolving software.** This is the first proposal for MOP in the context of software evolution, as far as we know.
- ⋆ **Synergy.** We are the first to combine MOP and RTS, and show synergy between them.
- ⋆ **Experiments.** We conducted the largest scalability study of MOP on several versions of open-source projects.
- ⋆ **Techniques.** We propose three new techniques for improving the efficiency and usability of MOP as software evolves.

II. Background and Example

Consider the buggy code snippet in Fig. 2, which mimics actual bugs discovered in production AspectJ code (bug IDs #218167 and #218171), caused by a typo on `Line 5`—it should be `iter2.hasNext()`. This causes a failure if `files` has more elements than `dirs` has. JavaMOP can catch this bug by monitoring the `HasNext` property shown in Fig. 1, as long as `findFiles()` is called with `files` having at least two elements. We use JavaMOP for pragmatic reasons—the ideas in this paper are general and implementable in any comparable

```
1 public boolean findFiles(List files, List dirs){
2   File file, dir; int count = 0;
3   for(Iterator iter = files.iterator(); iter.hasNext();){
4     file = (File) iter.next();
5     for(Iterator iter2=dirs.iterator(); iter.hasNext();){
6       dir = (File) iter2.next();
7       if (new File(dir, file.getName()).exists()){
8           count++; break;} //file is in dir
9     }}  return count == files.size(); }
```

Fig. 2: Java code demonstrating subtle bug

```
1 Specification Iterator_HasNext has been violated on line
      net.sf.jsqlparser.statement.Statements.findFiles(
      Statements.java:55). Documentation for this property
      can be found at http://runtimeverification.com/
      monitor/annotated-java/Iterator_HasNext.html
2 Iterator.hasNext() was not called before calling next().
```

Fig. 3: Sample JavaMOP violation message

MOP tool. The violation handler on line 7 in Fig. 1 can be any user-provided code; JavaMOP's default is to generate a *violation* containing the property name, the line number of the target program where the violation occurred, a URL of the formal property definition, and an explanation of the violation. Fig. 3 shows a sample violation of the `HasNext` property.

This example illustrates one benefit of, and one problem with, MOP. The *benefit* is that, as long as testing executes `findFiles()` (with `files` having at least two elements), JavaMOP will generate a violation due to the `HasNext` property, which can help discover the bug. Even if some test executes `findFiles()` with two lists of equal length, JavaMOP will generate the violation. In contrast, the checks that the standard Java library has for `next()` would not throw an exception unless `findFiles()` is called with `files` having more elements than `dirs` has.

The *problem* of monitoring the `HasNext` property is that it can lead to a lot of false alarms, e.g., if the developer implicitly or explicitly uses knowledge of a list's size to fetch the `next()` element without first calling `hasNext()`. In fact, violations of the `HasNext` property during our own formative study were dominated by false alarms because developers either explicitly called `size()` on a list before calling `next()`, or called `next()` `n` times, where `n` is the number of elements used to initialize the list. In these cases, the generated violations reduce developers' confidence in JavaMOP, and make it harder to inspect all violations (especially if there are a lot of them) and to find subtle but serious bugs, like the one in Fig. 2.

While one could attempt to devise more precise properties or employ some hybrid static and dynamic analyses to reduce false alarms, recent work on static analysis [12] shows that focusing on multiple software versions can bring additional benefits. We expect that developers will find MOP tools like JavaMOP more useful if all violations in one version are only generated from parts affected by the code changes made to the previous version. Specific proposals to achieve the envisioned improvements of eMOP over MOP are discussed next.

III. Approach

We propose three techniques to make MOP evolution-aware. Each proposed technique can reduce the monitoring overhead and the number of violations generated between versions. The first technique selects to *"re-monitor" only properties* that can be violated due to changes in the code. The second technique selects to *generate monitors only* for code that is affected by the changes, i.e., it does not generate monitors for code that is definitely not affected. The third technique *combines MOP with RTS*, which selects to rerun a subset of the tests—if fewer tests are re-run, we expect fewer properties to be re-monitored, and fewer monitors to be regenerated, compared to rerunning all the tests. These techniques are orthogonal to one another; they can be combined to further reduce the costs of MOP. We plan to implement these techniques as extensions to JavaMOP.

A. Regression Property Selection (RPS)

The goal of RPS is to re-monitor only properties that can be affected by the code changes. For example, if no call sites of any method in the `Iterator` interface are affected by code changes, there is no need to monitor the `HasNext` property (Fig. 1) in the new version. Luo et al. [14] recently proposed an approach for efficiently monitoring many properties at once. We plan to extend this approach to be aware of code changes. The key idea is to map each property to all the parts of the code from which events were sent to monitors initiated from that property. When code is changed, only the corresponding properties for the monitors that received events from the code affected by the change need to be re-monitored.

We will need to account for the different kinds of changes that can be made to code—modifications, additions, and deletions. To compute these changes, we will consider two options. First, if RPS is used alone, without RTS, we plan to use an efficient change impact analysis technique that accounts for object-oriented language features, like dynamic dispatch [16], [18]. Second, when RPS is used with RTS, we will reuse the changes that RTS already computes. Another concern is non-determinism of paths covered during execution, which may make some properties to be violated in some runs but not in others, for the same code version. We plan to use recent results in test non-determinism [13] to tackle non-determinism during property selection. Similar to RTS [7], [9], [16], the cost of analysis must be less than the cost of re-monitoring all properties for property selection to be practical. We plan to explore the different levels of granularity at which the impact of code changes is tracked (i.e., at the statement, block, method, or class level) to find a sweet spot in the trade-off among coarseness of granularity levels, speed of analysis, and precision of RPS.

B. Regression Monitor Selection (RMS)

RMS is another way to reduce the cost of re-monitoring. It prevents monitors from being regenerated for code that cannot be affected by the changes, even if the related property is affected by the change, and thus, RPS cannot omit the property for the entire code. For example, if two classes,

`Foo` and `Bar`, both use methods of the `Iterator` interface, but `Foo` cannot be affected by a code change, then, with RMS, monitors will not be initiated for call sites in `Foo`, although `HasNext` will be re-monitored in `Bar`. Currently in JavaMOP, monitors keep track of their context—one or more line numbers in the code from which events are fired—in order to generate meaningful violations, as shown in Fig. 3. We plan to extend this by making each monitor also keep track of the line of code on which it was initiated, and by persisting all the contextual information for use in subsequent runs. After a code change, when a monitor-initiation event is fired, a check is first made to see if the same event was fired from the same line on the old version. If this check is affirmative, a second check is performed to see whether the line on which the monitor-initiation event or the lines from which the monitor's associated events were fired in the previous version have been affected by the change. If none of these lines are affected by the change, then the monitor will not be initiated. We expect RMS to remove the overhead for initiating and running some monitors, and to suppress violations that would otherwise have been regenerated. There are known problems with using syntactic information, like line numbers, to prevent violations in the old version from being regenerated in the new version [12]. We expect these problems to be ameliorated by matching on multiple lines instead of a single line [17]. In addition, we plan to provide support for developers to manually prevent monitors from being generated for a given property or code fragment.

C. MOP plus Regression Test Selection

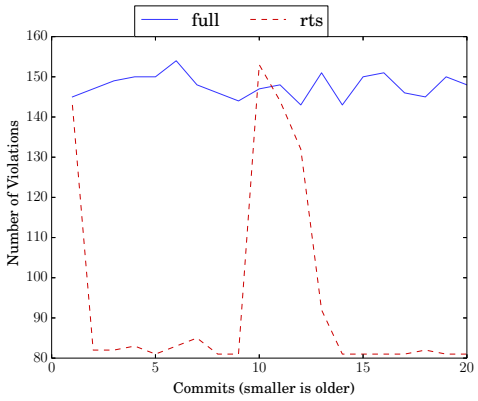
The techniques proposed in sections III-A and III-B, i.e., RPS and RMS, require knowledge of the changes between two versions of software. RTS, by definition, already computes changes. Thus, it is natural to use RTS together with RPS and RMS, especially as our approach is already test-driven. More so, running fewer tests is expected to incur less runtime overhead and trigger fewer violations, relative to running all the tests on every single version of the code. We started evaluating the combination of an RTS tool with JavaMOP.

IV. Preliminary Empirical Studies

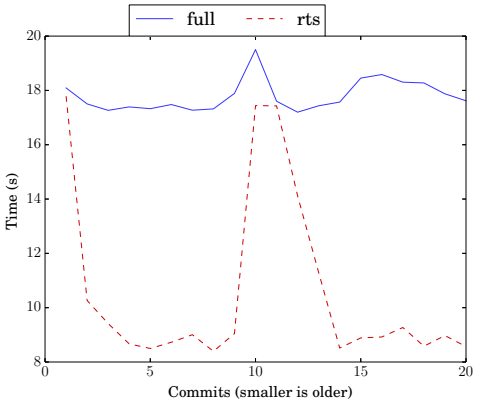
So far, we have investigated (i) how well JavaMOP works on open-source projects; (ii) the potential benefits of combining MOP with RTS; and (iii) if property selection can lead to performance and usability gains across multiple code versions.

Scalability: We evaluated JavaMOP on 852,021 lines of code in 40 open-source projects. We monitored 181 properties, formalized from the Java API [11] while running the existing tests in the projects. As far as we know, this is the first evaluation of MOP combined with large-scale unit testing. The average size of the evaluated projects was 21,300 lines of code—the smallest was 142 lines of code and the largest 186,796. The average number of tests across all 40 projects was 910.7, and the average time to run the tests was 16.4sec. This average time increased to 232.7sec after integrating with JavaMOP. These experiments show that, while JavaMOP scales to many open-source projects, the runtime overhead still requires further improvement.

The average number of violations generated while monitoring the tests across all 40 projects was 18,810. All violations were generated from 24 projects. Of the 16 projects that had no violations, 13 were large, mature, well-used and well-tested projects from the Apache Software Foundation. Taken together, these experiments motivated the idea behind eMOP—too many violations make tools less usable for developers [12] and, by being evolution-aware, MOP can become the approach of choice for incrementally reducing the number of violations in less mature software to zero.



(a) Violation counts for `AsteriskJava`



(b) Monitoring times for `AsteriskJava`

Fig. 4: Comparing performance with and without RTS

MOP + RTS: To assess MOP in the context of software evolution, we investigate the combination of JavaMOP and Ekstazi [7], [9], a recent, lightweight RTS tool. We perform the following experiments on `AsteriskJava`: (i) run JavaMOP with *all* the tests in 20 versions of `AsteriskJava` (`full`) and (ii) run MOP while executing *only* the test classes selected by Ekstazi in the same 20 versions (`rts`).

The number of violations and the monitoring overhead were collected on each run and those for `full` and `rts` are compared and plotted. Fig. 4 shows the result of comparing JavaMOP on `AsteriskJava` for `full` and `rts`. Even without RPS and RMS, combining MOP with RTS already yields

improvements across multiple versions. From the numbers in Fig. 4 the average number of violations and time to run all tests with JavaMOP on 20 versions of `AsteriskJava` were 147.75 and 17.8sec, respectively. These numbers dropped to 94.5 and 10.6sec, respectively, with RTS. We observed some test-nondeterminism (discussed in III-A) during manual inspection of anomalous `commit 1` and `commit 10` of Fig. 4. The number of test classes for `full` and `rts` are the same in `commit 1` but the number of violations are different. In `commit 10`, there are more violations in `rts` than in `full`, even though `rts` involved one fewer test class.

Property Selection: We performed a simple experiment to assess whether property selection benefits eMOP. The subject used was `JSQLParser`. The results are shown in Table I, where the second through fifth columns represent the total number of violations, violations of the `HasNext` property, unique number of properties violated, and the test run time, respectively. Row 1 is the initial run before integrating with JavaMOP. For row 2, JavaMOP is used to monitor the run of the tests in `JSQLParser`. Prior to running with JavaMOP again (row 3), we introduced the bug from Fig. 2 into a randomly selected class and inserted a call to `findFiles()` into a test in the corresponding test class. We then re-ran JavaMOP again with all the tests, while monitoring the same number of properties as in row 2. Finally, since only one property, `HasNext`, is affected by the change between the two versions, we manually forced JavaMOP to monitor only this property while running the tests again (row 4). The results show that RPS can save time, and show only violations that the user is interested in, as code evolves.

Run	Total	HasNext	Props	Time(s)
No MOP v1	N/A	N/A	N/A	8.4
Full MOP v1	27,895	0	6	164.1
Full MOP v2	27,904	9	7	231.8
eMOP (HasNext) v2	9	9	1	8.8

TABLE I: Preliminary Investigation of RPS

V. Related Work

While we are the first to propose techniques to make MOP evolution-aware, our work is enabled by a long line of research to make MOP more practical. Luo et al. [14] proposed algorithmic techniques to significantly improve the efficiency of MOP. These techniques were implemented in JavaMOP and enabled our evaluation on open-source projects. The original work on formalizing Java API properties [11] also enabled our experiments. eMOP is orthogonal to these approaches, having the aim of adapting MOP to software evolution.

Other techniques have been adapted to software evolution to make them more practical. Zhang et al. [23] proposed techniques for making mutation testing more practical by incrementally calculating results for a new version based on the results from the old version. Logozzo et al. [12] proposed a semantically sound technique for suppressing violations between two versions of code. eMOP is similar in spirit to these approaches but different in the specific techniques proposed and the application domain.

VI. Conclusion

We envision eMOP as an approach for making MOP evolution-aware and more practical. eMOP can reduce the runtime overhead for monitoring evolving software versions, and moreover, can show developers only the property violations based on the most recent code changes. Our preliminary experiments with JavaMOP showed that it is mature enough for use on open-source projects and can be used as a basis for eMOP. Other results are encouraging, and prove the concepts for two of the three techniques proposed. We believe eMOP opens up a new line of research towards more adoptable MOP.

Acknowledgments

We thank Lamyaa Eloussi, Milos Gligoric, Alex Gyori, Farah Hariri, Qingzhou Luo, Amarin Phaosawasdi, August Shi, and Zhang Yi for feedback on this work. This material is based upon work partially supported by NSF under Grant Nos. CCF-1012759, CCF-1421575, and CCF-1439957.

References

[1] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *OOPSLA*. 2007.
[2] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. In *RV*, 2007.
[3] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP*, 2007.
[4] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *FSE*, 2008.
[5] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *RV*, 2003.
[6] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining billions of AST nodes to study actual and potential usage of Java language features. In *ICSE*, 2014.
[7] Ekstazi. http://ekstazi.org/.
[8] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE*, 2008.
[9] M. Gligoric, L. Eloussi, and D. Marinov. Ekstazi: Lightweight test selection. In *ICSE Demo*, 2015.
[10] JavaMOP4. http://fsl.cs.illinois.edu/index.php/JavaMOP4.
[11] C. Lee, D. Jin, P. O. Meredith, and G. Roşu. Towards categorizing and formalizing the JDK API. Technical Report http://hdl.handle.net/2142/30006, Computer Science Dept., UIUC, 2012.
[12] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear. Verification modulo versions: Towards usable verification. In *PLDI*, 2014.
[13] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *FSE*, 2014.
[14] 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.
[15] P. Meredith, D. Jin, F. Chen, and G. Rosu. Efficient monitoring of parametric context-free patterns. In *ASE*, 2008.
[16] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *FSE*, 2004.
[17] S. P. Reiss. Tracking source locations. In *ICSE*, 2008.
[18] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *OOPSLA*. 2004.
[19] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. ASE, 2009.
[20] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *TACAS*, 2005.
[21] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *TSE*, 31(6), 2005.
[22] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 22(2), 2012.
[23] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *ISSTA*, 2012.