Techniques for Evolution-Aware Runtime Verification

Owolabi Legunsen, Yi Zhang, Milica Hadzi-Tanovic, Grigore Roşu, Darko Marinov
Department of Computer Science, University of Illinois at Urbana-Champaign, IL 61801, USA
{legunse2,yzhng173,milicah2,grosu,marinov}@illinois.edu

***Abstract*—Runtime Verification (RV) can help find bugs by monitoring program executions against formal properties. Developers should ideally use RV whenever they run tests, to find more bugs earlier. Despite tremendous research progress, RV still incurs high overhead in (1) machine time to monitor properties and (2) developer time to wait for and inspect violations from test executions that do not satisfy the properties. Moreover, all prior RV techniques consider only one program version and wastefully re-monitor unaffected properties and code as software evolves.**

We present the first evolution-aware RV techniques that reduce RV overhead across multiple program versions. Regression Property Selection (RPS) re-monitors only properties that can be violated in parts of code affected by changes, reducing machine time and developer time. Violation Message Suppression (VMS) simply shows only new violations to reduce developer time; it does not reduce machine time. Regression Property Prioritization (RPP) splits RV in two phases: properties more likely to find bugs are monitored in a critical phase to provide faster feedback to the developers; the rest are monitored in a background phase.

We compare our techniques with the evolution-*unaware* (*base*) RV when monitoring test executions in 200 versions of 10 open-source projects. RPS and the RPP critical phase reduce the average RV overhead from 9.4× (for base RV) to 1.8×, without missing any new violations. VMS reduces the average number of violations 540×, from 54 violations per version (for base RV) to one violation per 10 versions.

***Index Terms*—runtime verification, regression testing, software evolution, specifications, software testing.**

I. Introduction

Runtime Verification (RV) [4], [10], [11], [16], [21], [33], [34], [37], [43], [58] is a technique for monitoring program executions against formal properties. A property is a logical formula over a set of *events*, e.g., method calls; intuitively, it captures developers' intent on correct API usage [76]. An RV tool takes a program, program inputs (e.g., tests), and properties. The tool instruments the program based on the properties so that executing the instrumented program generates events and creates *monitors* to listen to events and check properties. The outputs are *violation messages* (*violations* for short) which report that the execution violated some property at a code location. RV helped find many bugs but induces high runtime overhead in executing the instrumented program instead of the uninstrumented program, and some violations do not indicate true bugs but are false alarms [52], [76].

All prior RV techniques considered only a *single program version*, but software evolves over multiple versions. Developers should ideally use RV whenever they run tests, to find more bugs earlier in the development process. However, as software evolves, rerunning traditional, evolution-unaware RV (*base RV*) has unnecessarily high overhead: machine time can be wasted on repeatedly checking unchanged code, and developers can repeatedly see the same violations (even if they want to handle some violations later, they have no way to suppress those violations). It is therefore important to develop techniques that can reduce RV overhead—in both machine and developer time—during software evolution. This paper presents compelling evidence that taking software evolution into account can significantly reduce RV overhead across *multiple program versions*.

A. Techniques

We present three evolution-aware RV techniques: *regression property selection* (*RPS*), *violation message suppression* (*VMS*), and *regression property prioritization* (*RPP*). RPS, VMS, and RPP focus RV (and its users) on changed parts of code and new violations that are generated. RPS can reduce RV overhead in machine time and developer time; VMS can reduce the overhead in developer time but not machine time; and RPP can reduce time to see results for most critical properties, e.g., those historically more likely to find bugs.

RPS re-monitors only properties that can be violated in parts of code *affected* by changes, i.e., either directly changed or indirectly affected; these code parts may generate new events due to changes. Our current implementation of RPS re-monitors only properties whose events can come from affected *classes*. We focused on class-level RPS following recent evolution-aware techniques which showed greater overall benefits of performing analysis at class level than at finer granularity levels like methods or statements [9], [26], [51], [88].

VMS by itself re-monitors all properties in a new code version, but shows only *new violations* that were not in the old version. VMS collects violations from both versions and computes a mapping of code between new and old versions. VMS then filters out violations of the same property that occurred on the likely equivalent locations in both versions. VMS makes it easier to focus on new violations, and developers can decide whether to inspect only new or also old violations.

RPP partitions RV into two phases: it monitors some properties in the critical phase—so called because it is on the developer's critical path from the moment of submitting code changes to getting the results—and monitors the remaining properties in the background phase. RPP reduces time to get feedback on critical properties but still monitors all properties. Developers select critical properties, e.g., those that helped find bugs or those for heavily-used APIs, etc. In our evaluation, critical properties become those that were previously violated.

We define safety and precision for evolution-aware RV techniques (Section III-A): an evolution-aware RV technique is *safe* if it does not miss a new violation, and *precise* if it shows only new violations. We develop two strong RPS variants that are safe under certain assumptions. We also develop 10 weak RPS variants that can trade some safety for more efficiency, i.e., reduced overhead. RPS variants differ in what properties they select and where they instrument the selected properties.

B. Results

We compared RPS, VMS, and RPP with base RV using 161 properties on 200 versions of 10 open-source projects (20 versions per project). The results showed that our evolution-aware RV techniques can substantially reduce the runtime overhead and number of violations shown, compared to base RV. We compute the runtime overhead and the number of shown violations per version, then average across versions of a project and then across all projects.

Base RV has average runtime overhead of 9.4×, showing 54 violations per version. The two strong RPS variants have runtime overhead of 7.5× and 7.9×, showing 37 and 42 violations. The 10 weak RPS variants have runtime overhead of 2.5×–7.5×, showing 21–37 violations. Surprisingly, *all* weak RPS variants were safe in our experiments although they can be unsafe in theory. Our manual inspection showed why: all new violations happened due to changes whose effects were in the classes considered affected by all weak RPS variants.

VMS has negligible extra runtime overhead and reduces the number of violations shown by two orders of magnitude relative to base RV; VMS shows, on average, 0.1 new violation per version, while base RV shows 54 violations per version.

RPP's critical phase overhead is 1.8× (when combined with RPS), and our analysis of RPP showed that about 76% of base RV overhead goes into monitoring unviolated properties.

C. Contributions

This paper makes the following contributions:

- **Evolution-Aware RV Techniques.** We are the first to realize the RPS idea [53] and additionally propose VMS and RPP.
- **RPS Variants.** We develop two strong RPS variants, and 10 weak RPS variants that trade some safety for efficiency.
- **Results.** RPS and RPP reduced base RV overhead from 9.4× to as low as 1.8×, and VMS showed two orders of magnitude fewer violations than base RV.

II. Running Example

Our running example is necessarily detailed to show the specifics of RV that evolution-aware RV techniques exploit, and to highlight differences between the RPS variants in later sections. We illustrate properties, how evolution-unaware base RV works in the JavaMOP [40], [43], [58] tool used in our experiments, and violations.

A. Examples of Monitored Properties

In our running example, we use the three properties in figures 1a–1c, written in JavaMOP syntax [38]; they helped find several confirmed bugs [52]. Properties have three parts: (1) *events*: relevant method calls or field accesses, (2) a *specification*: logical formula over the events, and (3) a *handler*: action to take when events match (or violate) the specification.

Collections_SynchronizedCollection (CSC): CSC checks that code synchronizes on a `synchronized Collection` before iterating over it [18]. Not synchronizing on such `Collection` before iterating "may result in non-deterministic behavior" [17]. CSC defines four *events* in lines 3–10 of Fig. 1a: (1) `sync` (lines 3–4) occurs when `Collections.synchronizedCollection` is called to create a `Collection c`, (2) `syncMk` (lines 5–6) occurs when `c.iterator` is called to obtain an `Iterator i` in a thread that holds the lock on `c`, (3) `asyncMk` (lines 7–8) occurs when `c.iterator` is called without first locking on `c`, and (4) `access` (lines 9–10) occurs when accessing `i` from a thread that does not hold `c`'s lock. When `sync` occurs, JavaMOP creates a *monitor* object to listen for CSC events (hence the `creation event` keywords).

CSC's specification (line 11 of Fig. 1a) is an Extended Regular Expression which matches if the code either (1) creates `c` (`sync` event) and obtains `i` without first locking on `c` (`asyncMk` event), or (2) creates `c` (`sync` event) and obtains `i` from a thread that locks on `c` (`syncMk` event) but accesses `i` from a thread that does not lock on `c` (`access` event). When a CSC monitor receives an event that causes its specification to match, its handler (line 12) is invoked. The handler can be any code, but most properties, including CSC, just print a violation to warn developers of a potential bug.

StringTokenizer_HasMoreElements (STHME): STHME checks that getting tokens from `StringTokenizer, st,` is only done after checking that `st` has more elements [82]. STHME (Fig. 1b) defines two events: (1) `hasnexttrue` (lines 2–4) occurs when `st.hasMoreElements` or `st.hasMoreTokens` is invoked and returns `true`, and (2) `next` (lines 5–7) occurs when `st.nextElement` or `st.nextToken` is invoked. The STHME specification (line 8) is a past-time LTL formula [57] stating that a `next` event on `st` must be preceded by a `hasnexttrue` event on `st`. When the STHME specification does not hold, line 9 prints a violation.

URLDecoder_DecodeUTF8 (URLD): URLD checks that URLs are decoded from UTF-8, to avoid producing incompatible URLs [83], [84]. URLD's only event, `decode` (lines 2–5 in Fig. 1c), occurs if URL is decoded with non-UTF-8 encoding. The handler on line 6 prints a violation on each `decode` event.

B. Base RV, Causes of Overhead, and Property Violations

We describe the example Java code in Fig. 1d and violations that occur when JavaMOP is used to monitor its execution against the CSC, STHME, and URLD properties. The example code is hypothetical, created to illustrate our techniques.

Example Code: Fig. 1d shows five classes—`A, B, C, D,` and `E`—and two versions—line 11 in the old version is replaced with line 12 in the new version. `A.a()` concatenates the string

```
Collections_SynchronizedCollection(Collection c, Iterator i) {
 Collection c;
 creation event sync after() returning(Collection c):
  call(* Collections.synchronizedCollection(Collection)){ this.c = c;}
 event syncMk after(Collection c) returning(Iterator i) :
  call(* Collection+.iterator()) && target(c) && Thread.holdsLock(c){}
 event asyncMk after(Collection c) returning(Iterator i):
  call(* Collection+.iterator()) && target(c) && !Thread.holdsLock(c){}
 event access before(Iterator i) :
  call(* Iterator.*(..)) && target(i) && !Thread.holdsLock(this.c){}
 ere: (sync asyncMk) | (sync syncMk access)
 @match{ RVMLogging.out.println(/*violation message*/); }}
```

(a) Collections_SynchronizedCollection (CSC) property

```
StringTokenizer_HasMoreElements(StringTokenizer s) {
 event hasnexttrue after(StringTokenizer s) returning(boolean b):
  (call(boolean StringTokenizer.hasMoreTokens()) ||
   call(boolean StringTokenizer.hasMoreElements())) && target(s) && b{}
 event next before(StringTokenizer s):
  (call(* StringTokenizer.nextToken()) ||
   call(* StringTokenizer.nextElement())) && target(s){}
 ltl: [](next => (*) hasnexttrue)
 @violation { RVMLogging.out.println(/*violation message*/); }}
```

(b) StringTokenizer_HasMoreElements (STHME) property

```
URLDecoder_DecodeUTF8() {
 event decode before(String enc) :
  call(* URLDecoder.decode(String, String)) && args(*, enc) {
   if (enc.equalsIgnoreCase("utf-8") || enc.equalsIgnoreCase("utf8"))
    return;
   RVMLogging.out.println(/*violation message*/); }}
```

(c) URLDecoder_DecodeUTF8 (URLD) property

```
1  class A {
2   String a(List i, String sep) {
3    String o = "";
4    for (Object a : i) {
5     o += a.toString() + sep;
6    } return o; }}
7
8  class B extends A {
9   String b(List l) {
10   String i;
11 - i = a(l, " ");
12 + i = a(Collections.synchronizedCollection(l));
13   return i.trim(); }
14  Boolean flag() { return true; }}
15
16 class C {
17  String c(List<String> l) {
18   B b = new B(); D d = new D();
19   String s = b.b(l);
20   return d.d(s, b.flag()) + ": " + s; }}
21
22 class D {
23  String d(String s, boolean flag) {
24  StringTokenizer t = new StringTokenizer(s);
25  String out = "";
26  if (flag) {
27   if(t.hasMoreTokens()){out = t.nextToken();
28  } else { out =  t.nextToken(); }
29  return out; }}
30
31 class E {
32  void e(String u, String e) throws Exception {
33   D d = new D(); assert(!u.isEmpty());
34   String url = d.d(u, false);
35   if (url.startsWith("https")) {
36    String s = URLDecoder.decode(url, e);
37    System.out.print(s); }}}
```

(d) Example evolving code

Fig. 1: Example properties and evolving code that we use to illustrate base RV and evolution-aware RV techniques

```
public class TC {
 @Test public void testC() {
  B b = new B(); C c  = new C(); D d = new D();
  List<String> ll = Arrays.asList("1", "2");
  assert(b.b(ll).equals("1 2"));
  assert(c.c(ll).equals("1: 1 2"));
  assert(d.d("1 2", false).equals("1")); } }

public class TE {
 @Test public void testE() throws Exception {
  E e = new E(); String u = "https://bing.com";
  assert(e.e(u + " b", "ISO-8859-1").equals(u)); } }
```

Fig. 2: Tests for code in Fig. 1d

representation of all elements in its input `List`. `B` extends `A` and `B.b()` invokes `A.a()` to get a string representation of the input `List`, which it then trims to remove leading or trailing white space. `C.c()` first invokes `B.b()` to obtain a string representation of its input `List`, which it prints after prefixing with the first sub-string, obtained from `D.d()`. `D.d()` tokenizes the input string and returns the first token; for performance reasons, it only checks that the input string has more than one token if its caller sets `flag` (e.g., the caller may already ensure non-emptiness). `E.e()` decodes an encoded HTTPS URL from a string after ensuring the string is not empty and invoking `D.d()` to get the first sub-string.

Monitoring and Causes of RV Overhead: We use the code in Fig. 1d to describe three RV concepts: instrumentation, monitor creation, and event/violation handling. Let us consider what happens when the tests in Fig. 2 are run on the old version of Fig. 1d. During class loading, JavaMOP instruments all statements in classes `A` through `E` that can generate events mentioned in the properties. The instrumentation causes events to be triggered during execution. Example instrumentation points in Fig. 1d include (1) before creating an `Iterator` on line 4 which may trigger CSC events, (2) after `hasMoreTokens` and before `nextToken` on line 27, and before line 28, which may all trigger STHME events, and (3) before line 36 which may trigger URLD events. At runtime, monitors are created to listen for and handle events. In the old version, only STHME and URLD monitors are created; `creation event` for CSC never occurs because `List l` on line 11 is not a synchronized `Collection`. One STHME monitor is created when the first relevant event occurs on each `StringTokenizer`; only one URLD monitor is created at the start of execution (unlike CSC and STHME, URLD has no parameters). Base RV induces high runtime overhead due to managing very many monitors and dispatching even more events to monitors [42], [58], e.g., with base RV, one project in our evaluation with 78 thousand lines of code created over 232 million monitors, which received almost 3 billion events.

Violations: When events occur that match or violate a monitor's specification, the violation handler prints a violation.

```
Specification Collections_SynchronizedCollection has been violated on line B.b(B
.java:11). Documentation for this property can be found at https://
runtimeverification.com/monitor/annotated-java/__properties/html/java/util/
Collections_SynchronizedCollection.html
A synchronized collection was accessed in a thread-unsafe manner.
```

Fig. 3: An example property violation

like in Fig. 3. A violation contains the violated property name, the location (i.e., fully qualified class name, method, source file name, and line number) of the last event that caused the violation, a URL for the property definition, and a sentence describing the violation. These help developers to reason whether a property violation is a true bug or false alarm.

We distinguish between *violation instances*, the *list* of violations, and the set of *violations*. Violation instances repeat, e.g., if property-violating code is in a loop or executed by multiple tests. We map violation instances of the same property that occur at the same location to the same violation. Developers may prefer to only see violations, but seeing all violation instances can help in debugging. Running tests in Fig. 2 on old version of Fig. 1d generates two violations from three violation instances. Lines 7 and 12 in Fig. 2 cause two instances of a STHME violation by executing `t.nextToken` on line 28 of Fig. 1d without calling `t.hasMoreTokens`. Line 12 in Fig. 2 causes one instance of a URLD violation by executing line 36 of Fig. 1d to decode a non-UTF-8 encoded URL. It can be time consuming to inspect/debug violations [52]. We next discuss evolution-aware RV techniques which aim to reduce runtime overhead of RV and show fewer violations as software evolves.

III. Evolution-Aware RV Techniques

We describe our evolution-aware RV techniques which leverage software evolution to reduce the runtime overhead of base RV across multiple program versions and to focus developers on new violations after a change. Base RV (illustrated through the example in Section II) is evolution-unaware. For example, running base RV on the new version of code in Fig. 1d would re-monitor all available properties and re-incur the entire overhead wastefully because the code change does not affect (i.e., alter the behavior of) all classes, e.g., `E` is unaffected. Further, properties whose events are only generated from unaffected classes cannot have any new violations after the code change. Finally, it may be desirable to monitor on the developer's critical path, from when they launch tests to when they see the test results, only properties that are more likely to find bugs than others, e.g., based on a project's history.

Section III-A defines *safety* and *precision*, two notions that we use in this paper to analyze and measure the quality of our evolution-aware RV techniques. Section III-B describes RPS, our technique to re-monitor only properties that can have new violations after a code change, and also includes our definition of affected classes and how RPS uses affected classes to select the subset of properties to re-monitor in a new program version. Section III-C describes various RPS variants. Sections III-D and III-E describe our other two evolution-aware RV techniques, VMS and RPP, respectively. RPS, VMS, and RPP can be used separately or together, and we illustrate them throughout this section using the example from Fig. 1.

A. Safety and Precision

Safety measures loss in violation-finding (and thus potential bug-finding) ability. Precision measures minimality. We define safety and precision relative to base RV and *relevant* violations. In this paper, relevant violations are *new violations*—violations that are in the new version, but not in the old version, after accounting for violations that merely changed line numbers in the code. Definition 1 allows developers to plug in other notions of relevant violations.

Definition 1. *Relevant Violation: Relevant violations for an evolution-aware RV technique are those due to the changes.*

Definition 2. *Safety: An evolution-aware RV technique is safe if it finds* all *relevant violations that base RV finds.*

Definition 3. *Precision: An evolution-aware RV technique is precise if it finds* only *relevant violations that base RV finds.*

B. Regression Property Selection (RPS)

RPS reduces accumulated base RV overhead by re-monitoring only properties that can be violated in parts of code affected by changes [53]. For RPS to be useful, its end-to-end time (i.e., time to select properties plus time to re-monitor selected properties) must be less than base RV time. Thus, we consider changes and affected parts of code at the class-level granularity, which was more effective than only finer-granularity levels (e.g., statements or methods) for other evolution-aware techniques [26], [51], [88]. The reason is that the analysis at the class level achieved a better balance of efficiency (class-level analysis is faster than analyses at finer granularity) and precision (class-level analysis may capture more than necessary because it is coarser grained).

The notion of *affected classes* is central to RPS, because it relates code changes with the properties. Intuitively, a property should be re-monitored only if its events can be generated from some class affected by the code change. That is, affected classes are those that can generate events that lead to new violations after code changes. Conversely, a class that is *unaffected* by a change cannot generate an event that leads to a new violation. Formally, RPS variants compute affected classes as those that satisfy some of the following conditions, which capture when a class may generate events that lead to new violations after a code change:

Definition 4. *Affected Class: For RPS, a class C is affected by a change if (1) C changed, (2) C transitively depends (via inheritance or use) on a class that changed, or (3) a class that satisfies (1) or (2) can pass objects to C.*

Condition 3 captures classes whose control flow may change (leading to new events and violations) if received objects change. For example, in Fig. 1d, `D` does not depend on the changed class (`B`) or its transitive dependents (`C` and `TC`); if only `B.flag()` changes to return `false` on line 14, then the

lse" branch on line 28, instead of the "then" branch on line 27 will execute, leading to a STHME violation.

Definition 5. *Regression Property Selection (RPS): A technique to select and re-monitor, in a new program version, only properties that may have new violations.*

RPS has four steps: (1) construct a class dependency graph (CDG) from the new program version, (2) find affected classes, (3) select properties, and (4) re-monitor selected properties.

Definition 6. *Class Dependency Graph (CDG): A graph that has a node for each class in the program, and an edge from class C to class C′ if C depends on C′ via inheritance or use.*

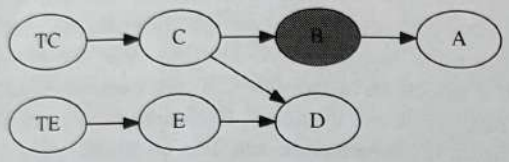

Fig. 4: Class dependency graph (CDG) for Figures 1d and 2. Edges mean "depends on"; the changed class is colored

Step 1: RPS constructs the CDG in Fig. 4 for the new version of the code in Fig. 1d and the tests in Fig. 2.

Step 2: Strong RPS computes affected classes from the CDG as $affected(\Delta) = \Delta \circ (E^{-1})^* \circ E^*$, where Δ is the set of changed and new classes, E is the set of edges in the CDG, * is the reflexive and transitive closure, $\circ$ is the relational image, and $^{-1}$ is the inverse relation. $affected(\Delta)$ captures the three conditions in Definition 4. In our example, $\Delta = \{\texttt{B}\}$; only `B` changed (Condition 1). $\Delta \circ (E^{-1})^* = \{\texttt{B}, \texttt{C}, \texttt{TC}\}$; `TC` and `C` transitively depend on Δ (Condition 2). Lastly, $affected(\Delta) = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{TC}\}$; `A` and `D` may generate new events due to changes to `B` or the interaction of `C` with `B` (Condition 3). `E`, `TE` $\notin affected(\Delta)$ since they cannot generate new events. Although elided in our example due to space limits, newly added classes are in Δ, so RPS re-monitors properties that may be violated in newly added classes.

Steps 3 and 4: RPS re-monitors only CSC and STHME in the new version. No (new) events for URLD are generated in $affected(\Delta)$. So, RPS saves the time to re-monitor URLD (if both tests are run), and developer time for (re-)inspecting URLD violations. Any URLD violations must be in `E` and cannot be new violations, because `E` $\notin affected(\Delta)$.

Discussion of RPS: If a property was not instrumented into the old version, but code changes can cause it to be violated in $affected(\Delta)$, RPS selects it, e.g., CSC is selected by strong RPS. Two CSC violation instances occur in the new version in Fig. 1d; lines 4–6 iterate over the `synchronized Collection` initialized on line 12 without locking on it, matching the left disjunct in CSC's specification (line 11, Fig. 1a), so the handler (line 12) prints the violation in Fig. 3.

Base RV does not consider changes, dependencies, or classes that generate events for each property. After each change (e.g., from line 11 to line 12 in Fig. 1d), base RV re-monitors all properties and shows old and new violations. In our example, base RV shows three violations: the two STHME and URLD violations from the old version, plus the new CSC violation. RPS shows only the old STHME violation, plus the new CSC violation. Note that RPS by itself is not precise: it does *not* show only new violations. Showing only new violations is the goal of VMS (Section III-D).

C. RPS Variants

RPS determines (1) *what* properties to select and (2) *where* in the program to instrument selected properties. The *strong RPS* described in Section III-B is safe under certain assumptions: it selects to re-monitor *all* properties for which events can be generated from *all* affected classes ("what"), and instruments them throughout the program ("where"), including third-party libraries and even unaffected classes. However, that strong RPS variant is imprecise (it may instrument and monitor selected properties in unaffected classes). We describe here a second, more precise strong RPS variant. *Weak RPS* variants trade some safety for further overhead reduction. Weak RPS variants differ in what affected classes they use for selecting properties and where they instrument selected properties.

Strong RPS Safety Assumptions: Strong RPS is safe under the following assumptions: (1) the CDG is complete, (2) there are no test order dependencies [8], [29], [89], and (3) dynamic language features, e.g., reflection and classloading, do not introduce additional CDG edges.

Notation: Subscripts distinguish how affected classes are computed. ps_1 computes $affected_1(\Delta) = \Delta \circ (E^{-1})^* \circ E^*$ (Definition 4). ps_2 computes $affected_2(\Delta) = \Delta \circ ((E^{-1})^* \cup E^*)$, which consists of only classes that either depend transitively on Δ (dependents) or Δ transitively depends on (dependees); $affected_2$ is more unsafe than $affected_1$ because it omits condition 3 from Definition 4 to not include classes, e.g., `D` in Fig. 4, that may generate new events because they receive objects from dependents of Δ. ps_3 relaxes Definition 4 even further by omitting condition 3; it computes $affected_3(\Delta) = \Delta \circ (E^{-1})^*$, i.e., only dependents of Δ.

Once the corresponding set of affected classes ($affected(\Delta)$) has been used to select the properties to re-monitor (namely properties whose events may be generated from $affected(\Delta)$), we obtain more variants by choosing "where" to instrument the selected properties. We can reduce where to instrument the selected properties, in order to obtain more reduction of base RV overhead, at two levels: (1) do not instrument the selected properties in unaffected classes in the program but still instrument all third-party library classes loaded into the JVM, and (2) do not instrument the selected properties in any third-party library class.

For the first level of instrumentation reduction, we use the superscript `c` to show that unaffected classes in the program (i.e., complement of $affected(\Delta)$) are not instrumented: $ps_1^{\texttt{c}}$ excludes $(affected_1)^{\texttt{c}}$, $ps_2^{\texttt{c}}$ excludes $(affected_2)^{\texttt{c}}$, and $ps_3^{\texttt{c}}$ excludes $(affected_3)^{\texttt{c}}$. To see the benefit of not instrumenting $affected(\Delta)^{\texttt{c}}$, consider ps_1 and $ps_1^{\texttt{c}}$, which are both safe. $ps_1^{\texttt{c}}$ is safe because unaffected classes cannot generate any new events or alter the sequence of events for the selected

TABLE I: "What" properties RPS variants select

What	ps_1	ps_2	ps_3
properties in Δ	✓	✓	✓
properties in dependents of Δ	✓	✓	✓
properties in dependees of Δ	✓	✓	✗
properties in dependees of dependents of Δ	✓	✗	✗

TABLE II: "Where" RPS variants instrument properties

Where ($i \in \{1,2,3\}$)	ps_i	ps_i^{c}	ps_i^{ℓ}	$ps_i^{c\ell}$
$affected(\Delta)$	✓	✓	✓	✓
$affected(\Delta)^{c}$	✓	✗	✓	✗
third-party library classes	✓	✓	✗	✗

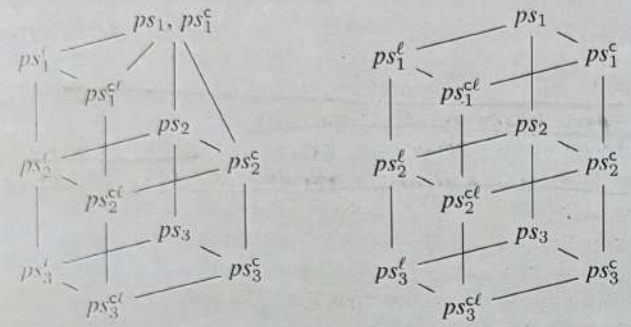

Fig. 5: Lattices of RPS variants. Left lattice ordered by "less safe than". Right lattice ordered by "more efficient than"

properties, so they cannot have new violations. However, ps_1^{c} can be more efficient and more precise (i.e., show fewer old violations) than ps_1 if selected properties can generate events from classes in $(affected_1)^{c}$. For example, in the CDG of Fig. 4, if a selected property p can generate events from $\in affected_1$ and E$\in(affected_1)^{c}$, and tests TC and TE are run, ps_1^{c} can save the time to monitor p in E. (Note that when safety assumptions of strong RPS do not hold, ps_1 is safer than ps_1^{c}; by instrumenting selected properties in unaffected classes, ps_1 can find some violations that ps_1^{c} miss.) On the other hand, not instrumenting $affected(\Delta)^{c}$ can make weak RPS variants more unsafe—an weak RPS variant that instruments all classes has a chance to find some violations from instrumented classes that are not in the computed $affected(\Delta)$.

The second level of instrumentation reduction does not instrument any third-party library class. We denote weak RPS variants that exclude *all* third-party library classes with ℓ in the superscript. For example, $ps_3^{c\ell}$ means that $affected_3$ is used to select properties, classes in $(affected_3)^{c}$ are not instrumented, and third-party library classes are also not instrumented. ps_3^{ℓ} means that $affected_3$ is used to select properties and only third-party library classes are not instrumented. In sum, we evaluate strong RPS (ps_1, ps_1^{c}) and 10 weak RPS variants: ps_2, ps_3, ps_2^{c}, ps_3^{c}, ps_1^{ℓ}, ps_2^{ℓ}, ps_3^{ℓ}, $ps_1^{c\ell}$, $ps_2^{c\ell}$, and $ps_3^{c\ell}$. Tables I and II distinguish RPS variants in terms of what part of the CDG is used for selecting properties, and where the selected properties are instrumented; ✓ means inclusion, and ✗ means exclusion.

Efficiency/Safety Tradeoff: Weak RPS variants trade some safety for lower runtime overhead. Fig. 5 shows two lattices of RPS variants; lower variants can be less safe (left lattice) more efficient (right lattice) than higher ones. ps_2 computes $\{$A, B, C, TC$\}$ as $affected_2$. D is not in $affected_2$, so ps_2 can miss new STHME violations, e.g., when changing only `true` to `false` on line 14 in Fig. 1d—ps_2 does not even re-monitor STHME for this change. If such cases are rare, then ps_2 can be safe but have lower overhead than strong RPS. In general, ps_2 can be unsafe if there is data flow to classes that are not dependents or dependees of Δ. ps_3 computes $affected_3$ as $\{$B, C, TC$\}$—dependents of Δ—which includes neither Δ's dependees, e.g., A, nor dependees of Δ's dependents, e.g., D. Therefore, ps_3 can be more unsafe than ps_2, e.g., if A changes such that a new violation results from events that are local to A, ps_2 will find that new violation, but ps_3 will not find it. Technique ps_3^{c} could miss new violations that ps_2^{c} finds. In fact, ps_3^{c} misses the new CSC violation after the change in Fig. 1d; it selects to re-monitor CSC but does not instrument A, so `asyncMk` is not triggered. Excluding third-party library classes from instrumentation can also be unsafe, e.g., if A or D are third-party library classes. Weak RPS variants that do not instrument $affected(\Delta)^{c}$ can be faster but safe if changes only lead to new violations in Δ and its dependents. Lastly, for weak RPS, false alarms can result from excluding classes from instrumentation. For example, if the STHME property's `hasnexttrue` event is triggered from a third-party library class that is not instrumented, and the `next` event is triggered in $affected(\Delta)$, a violation will occur even though the program satisfies the STHME property

D. Violation Message Suppression (VMS)

VMS improves base RV by showing only new violations. Showing only new violations right after a change is more effective than showing old plus new violations to get developers to act—they are still in the mental context of the change and are the ones who can best address new violations [65].

Definition 7. *Violation Message Suppression (VMS): A technique to show, in a new program version, only new violations that did not occur in an old version.*

Base RV shows three violations in the new version of the example in Fig. 1d: line 4 (two instances), line 28 (two instances), and line 36 (one instance). The latter two were in the old version, and their line numbers did not change. (More generally, VMS does not simply check equality of line numbers but builds a likely mapping between old and new line numbers based on code context.) In the new version, VMS shows only the violation on line 4, instead of showing all three. VMS can be used with RPS to reduce the old violations shown by RPS. Running RPS on new version of Fig. 1d will show two violations (lines 4 and 28); VMS shows only one (line 4).

VMS' inputs are the violations from the old and new versions, plus the source files in both versions. Each violation, $v = \langle p, c, l \rangle$, contains a triple of the property name (p) that was violated, and the class (c) and line number (l) of the last event that violated the property. Let V_1 and V_2 be the set of violations from monitoring the old version (P_1) and the new

ersion (P_2), respectively. VMS computes V_{new}, the set of new violations that are in P_2 but not in P_1. VMS does not simply compute $V_2 \setminus V_1$ that may report many old violations for which only the line numbers changed. Using only line numbers to match statements in two code versions performs poorly [75].

For each class $C_\delta \in \Delta$, where Δ is the set of changed classes (including newly added and renamed classes), VMS first creates a mapping, M_{C_δ}, from line numbers in the source file of C_δ in P_2 to line numbers with the likely same statement in the corresponding source file in P_1. Each line number in P_2 maps to at most one line number in P_1; some line numbers in P_2 may not be in M_{C_δ}. Note that M_{C_δ} is likely (i.e., not exact) as it is based on simple syntactic and not semantic equivalence; the latter is rather challenging and does not scale currently [28], [56]. M_C is identity if C did not change. Then, $V^{\Delta}_{new} = \cup_{C_\delta \in \Delta}\ VMS(V_1, V_2, M_{C_\delta})$, where $VMS(V_1, V_2, M_{C_\delta}) = \{\langle p, C_\delta, l\rangle \in V_2 \mid \nexists l' \in M_{C_\delta}(l) \vee \langle p, C_\delta, M_{C_\delta}(l)\rangle \notin V_1\}$. Let Δ' be the set of unchanged classes. New violations in Δ' are $V^{\Delta'}_{new} = \cup_{C \in \Delta'}\ VMS(V_1, V_2, M_C)$. $V_{new} = V^{\Delta}_{new} \cup V^{\Delta'}_{new}$ is the output of VMS. $V^{\Delta'}_{new}$ is non-empty when interactions with changed classes cause new violations in Δ', or when test non-determinism i.e., "flakiness" [7], [9], [30], [58], [79] leads to non-determinism during monitoring.

Discussion of VMS: VMS can save developer time for inspecting violations but slightly increases machine time, e.g., VMS increases time by <1% in our experiments. As we showed with our example, VMS can further reduce violations shown by RPS.

E. Regression Property Prioritization (RPP)

Developers may be more interested in violations of critical properties than other violations, e.g., violations of properties that previously helped find bugs may be more critical. RPP partitions RV into two phases: a *critical* phase and a *background* phase. After a code change, the critical phase immediately re-monitors (manually or automatically selected) critical properties and provides results to developers. The background phase separately re-monitors other properties. Developers get delayed feedback if non-critical properties are violated. RPP allows (manually or automatically) moving properties between the phases as properties become more or less critical during software evolution. To evaluate RPP, we consider previously violated properties as critical. RPP is inspired by regression test prioritization [22], [36], [78], [81], [87], but we are first to propose RPP for reducing RV overhead as software evolves.

Discussion of RPP: The benefit of RPP is to remove the re-monitoring of non-critical properties from developers' critical path (from the moment of submitting code changes to the moment of getting feedback). RPP's disadvantage is that it delays the time for developers to get feedback if non-critical properties are violated. RPS and VMS can be used with RPP—RPP merely first runs some subset of selected properties.

IV. Implementation

We present our implementation of RPS, VMS, and RPP.

A. Regression Property Selection (RPS)

Building CDG, Computing Changes and Affected Classes: We used STARTS [51], [54] to build CDGs, compute Δ, find $affected(\Delta)$ in P_1, and persist checksums of classes in P_1 to disk. The checksums are used to compute the classes that changed between P_1 and P_2. STARTS is a publicly available regression test selection (RTS) tool that implements most of these steps. By default, STARTS computes $affected_3$, which suffices for RTS [47], [51], [54], [66], but is not sufficient for strong RPS. We extended STARTS to compute $affected_1$ and $affected_2$. We chose STARTS because it is static and fast—it requires neither test runs nor code instrumentation to find dependencies among classes, or compute $affected(\Delta)$. We monitor test executions, so using a dynamic technique to compute dependencies or $affected(\Delta)$ would incur additional overhead. Also, instrumentation performed by a dynamic technique could interfere with JavaMOP instrumentation.

Monitoring: We used JavaMOP [43], [58] to monitor test executions against formal properties. JavaMOP is publicly available [40], uses AspectJ for load-time instrumentation, and allows monitoring many properties in one execution. JavaMOP was used in several RV studies [10], [20], [37], [52], [53], [58], [72], [74]. In each version, we follow publicly available instructions [39] to build and attach a JavaMOP agent [64] with selected properties to the JVM that executes tests.

Selecting Properties to Re-monitor: The properties re-monitored are those for which $affected(\Delta)$ can generate events. To select properties, we first used the AspectJ compiler to very quickly and statically weave all available properties into $affected(\Delta)$, and record properties whose aspects get weaved. If aspects from a property do not get weaved into any class in $affected(\Delta)$, its events cannot be generated from $affected(\Delta)$ at runtime. Time to select properties is part of RPS end-to-end time, so we optimized static weaving to be as fast as possible—only 3.3s on average in our experiments.

B. Violation Message Suppression (VMS)

VMS implementation is straightforward: (1) take violations from P_1 and P_2, (2) remove violations generated in P_2 if line mapping can map the same violation to a likely corresponding line number in P_1 (after taking care of renames), and (3) report any remaining violations generated in P_2 as likely new violations. Our line mapping extends the jDiff utility of jEdit [41], a Java implementation of Myers' classic algorithm [61].

C. Regression Property Prioritization (RPP)

We considered critical properties to be those that were violated in the project's history. In the first version of each project, there is no history, so there is a choice to monitor all properties in either the critical or background phase. It is not clear which of these choices is better: monitoring all properties in the either phase for the first version unfairly increases its average overhead. Therefore, we split properties into critical and background phases after the first version, depending on whether they were violated in the first version. We do not include the first version when computing the

TABLE III: Projects in our study

Name	#Test	KLOC	Tests[s]	t_{mop}/t_{tests}
commons-dbcp	26	20.1	56.5	2.0
imglib2	74	44.2	11.3	3.7
commons-lang	130	69.5	22.4	3.9
jackson-core	79	31.7	11.2	5.6
commons-io	96	29.2	106.5	5.8
commons-math	432	180.4	93.6	6.4
imaging	63	37.6	18.4	6.4
javapoet	17	7.9	10.7	7.2
stream-lib	24	8.4	127.1	12.2
opentripplanner	126	78.7	55.1	40.5
X	106.7	50.7	51.3	9.4

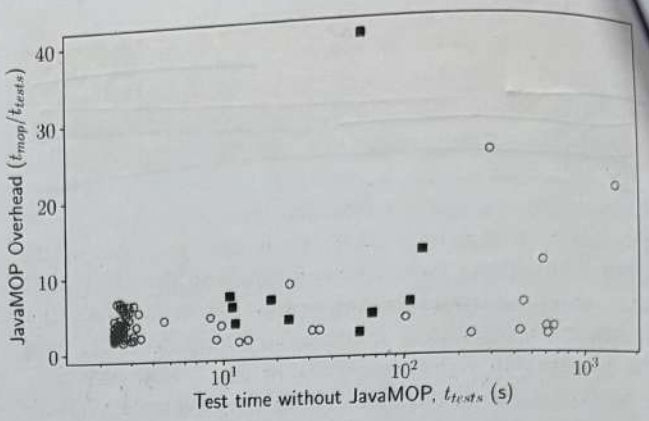

Fig. 6: Test time vs. base RV overhead for several projects

average overheads of each phase. From the second version onward, if a property gets violated in the background phase, our RPP implementation moves it to the critical phase in the next version. We leave it as future work to investigate criteria for moving properties which have not been violated after a while from the critical phase back to the background phase.

V. Evaluation

We list our research questions, describe our experimental setup, and answer the research questions.

A. Research Questions

We answer the following research questions: **RQ1:** How much does RPS reduce the machine time overhead of base RV? **RQ2:** How many violations does VMS show and how safe are RPS variants? **RQ3:** How much does RPP reduce time for developers to get feedback on critical properties?

B. Experimental Setup

Projects: Table III shows 10 open-source, Maven-based Java projects from GitHub used in our study, 9 of which we also used in prior work [51], [54], [80]. #Test is average number of test classes used (we skipped very few test classes from 6 projects due to problems with JavaMOP instrumentation), KLOC is average thousands of lines of code, Tests[s] is average test time, and t_{mop}/t_{tests} is average base RV overhead.
Properties: We used 161 manually written properties found to be good in our prior study [52]. The properties were written to formalize Java APIs [50], [58] and are publicly available [70].
Versions: We started from a recent commit in each project and went back into the history, to select 20 commits/versions where (1) at least one `.java` file changed, (2) all tests pass without JavaMOP, and (3) all tests pass with JavaMOP.
Running Experiments: We wrote scripts to automate running tests, collect violations and measure time for three configurations on each version: (1) without JavaMOP, (2) with base RV, and (3) with each evolution-aware RV technique. For RPS, the most common case is that `.java` file changes modify the bytecode, so properties may need to be re-monitored. If `.java` file changes do not modify bytecode, we skip tests (no re-monitoring); time is only spent to check for changes. If changes affect bytecode, but no properties are selected to be re-monitored, all tests are run without JavaMOP, and the end-to-end time is the time to compute changes, find *affected*(Δ), check if properties need re-monitoring, and run tests.

C. RQ1: Overhead reduction from RPS

We present RV overhead in multiples ($\times$), as the ratio t_{mop}/t_{tests}, where t_{mop} is time with JavaMOP and t_{tests} is time without JavaMOP. We first show on a sample of 89 projects whether the high overhead induced by base RV can be seen in open-source projects with short- (<10s), medium- (10s–300s), and long-running (>300s) tests. These 89 projects were sampled from our prior studies [51], [52], [54], [80] and from the Apache continuous integration server. Fig. 6 plots t_{tests} (x-axis, *log* scale, in seconds) vs. t_{mop}/t_{tests} (y-axis). Projects in all three categories exhibit high overhead, so high base RV overhead is not a fixed cost that is more pronounced in projects with shorter-running tests. Squares show projects in this study. We did not evaluate our techniques on the other projects because (1) $t_{mop} - t_{tests}$ is too small for RPS to be beneficial, (2) test-running times are high for long-running projects which requires more resources than we have to evaluate them, or (3) we could not get 20 versions that satisfy our criteria.

Solid bars in Fig. 7 show average runtime overhead of base RV (*BL*) and the RPS variants (ps) discussed in Section III-C. All overheads are computed from end-to-end time including time for analysis, running tests, and monitoring test executions. The results show several points. First, all RPS variants reduced the average base RV overhead, which is 9.4×. Strong RPS variants, ps_1 and ps_1^c, have 7.9× and 7.5× overhead, respectively. As expected, weak RPS variants with fewer classes in *affected*(Δ) achieve more reduction. $ps_3^{c\ell}$ is the most efficient weak RPS variant, with 2.5× overhead. Second, comparing *BL* and BL^ℓ shows that base RV spends about 36% of overhead on third-party library code: $(BL-BL^\ell)/BL$. Since ps_1^c is safe under certain assumptions, and, as we show in Section V-D, excluding unaffected and third-party library classes was safe in our experiments, $ps_1^{c\ell}$ may, in general, achieve the best efficiency/safety tradeoff among weak RPS variants.

We also evaluated how much regression test selection (RTS) [14], [23], [24], [26], [27], [31], [51], [77], [87], [88]

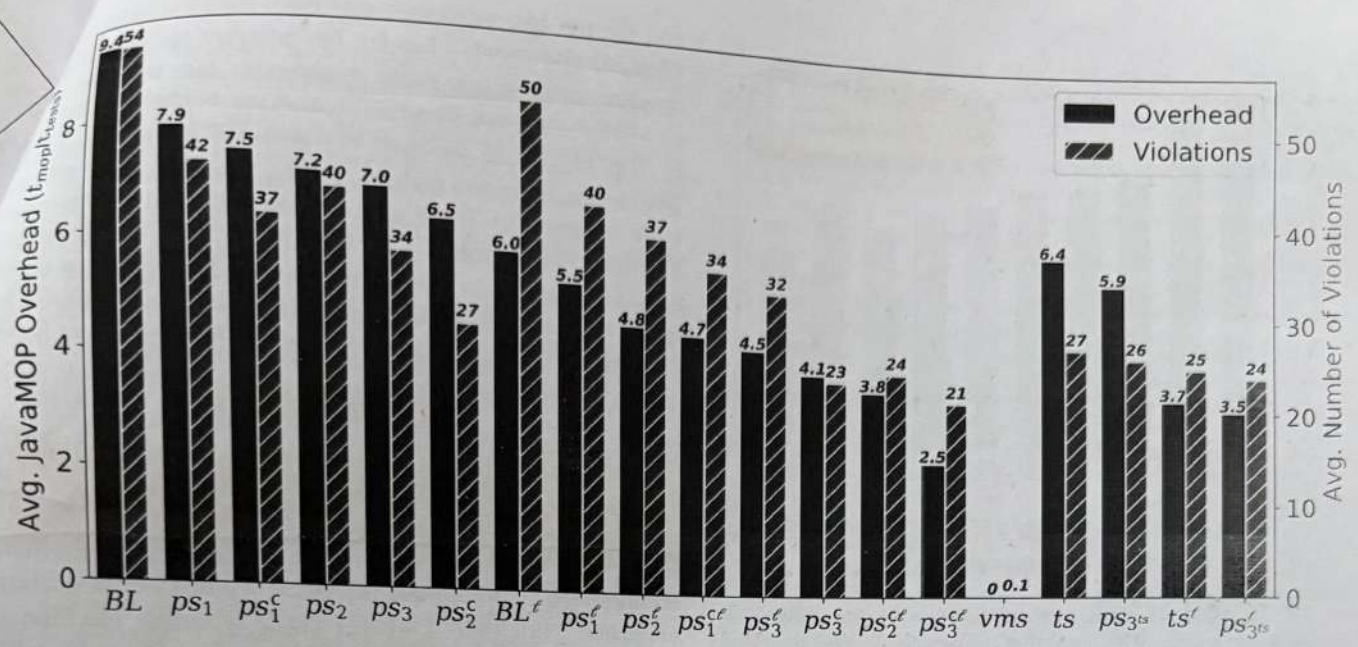

Fig. 7: Runtime overheads of, and violations from base RV (*BL*), RPS variants (*ps*), VMS, and RTS (*ts*) with 161 properties

can reduce base RV overheads during software evolution. RTS is a general approach (independent of RPS) for reducing the overhead of regression testing by re-running only a *subset of tests* whose behavior can differ after code changes. We previously noted that RTS can also reduce base RV overhead, since RTS already reduces testing overhead [53]. We evaluate a static class-level RTS technique, implemented in STARTS, which uses the same CDG as RPS. (Other RTS techniques compute dependencies dynamically [24], [26], [88], but it was challenging to evaluate them because their instrumentation often clashed with JavaMOP.) Since RTS can select more tests than those whose behavior differs after a code change, it may be imprecise as an evolution-aware RV technique (Section III-A). So, we also evaluated RPS plus RTS to see how these two can together further reduce base RV overhead.

The four rightmost solid bars in Fig. 7 show the overhead with RTS, with and without libraries. *ts* shows combination of RTS with base RV, i.e., rerun a *subset of tests* but re-monitor *all properties*, while $ps_{3^{ts}}$ shows RPS (using variant ps_3) plus RTS, i.e., rerun a *subset of tests* and re-monitor a *subset of properties*. When measuring the overhead, we used end-to-end RTS time, which includes the time to select the tests. Combining base RV with RTS has 6.4× overhead, compared with 9.4× for base RV. RPS plus RTS gives lower overhead (5.9×) than RTS alone, showing that RPS can provide value even where RTS is used. Since RTS can be unsound, it may incorrectly miss to select tests [23], [26], [51], [77], which makes RTS an unsafe evolution-aware RV technique. Finally, as we show in RQ2, RTS by itself is imprecise; it should be combined with VMS to show only new violations.

D. RQ2: VMS and RPS Safety

We discuss the results of VMS, and how we used these results to guide our manual checking of RPS safety. The striped bars in Fig. 7 show average number of violations from all techniques evaluated in Section V-C; the *vms* bar shows VMS average. The most significant result is that VMS is orders of magnitude more precise than RPS. For four projects, no new violation occurred in the range of versions.

Further, library exclusion results in very little difference in the average number of violations, which is good, because most violations can still be found when libraries are excluded. Very few violations in the libraries makes sense because libraries are widely used and tend to be better tested. We manually checked all violations that are not generated when libraries are excluded and found 87.5% of them to be in the third-party libraries themselves, and not in the project code. Among these, only one was a new violation and it was due to a library version change—all events leading to the violation were in the new version of the library, so library exclusion did not lead to missing any new violation in the projects. All violations that are missed in projects when excluding libraries were not new violations, providing some justification for excluding libraries when one does not care about violations in libraries, and partial explanation for why library exclusion did not lead to missing new violations.

RPS Safety: We manually confirmed safety of RPS variants by checking if all new violations from VMS were also reported by each variant. VMS reported a total of 33 new violations in 16 of the 200 versions across all projects. Of these, 5 were due to flakiness, which we confirmed by re-running several times (i.e., these 5 violations could also have happened in the old version). All RPS variants found 27 of the new violations, but all missed one new violation. The one new violation that all variants missed was the aforementioned library version change. It is surprising that weak RPS variants were safe in our experiments, since they are theoretically unsafe. Therefore, we carried out further manual inspection of the changes involved

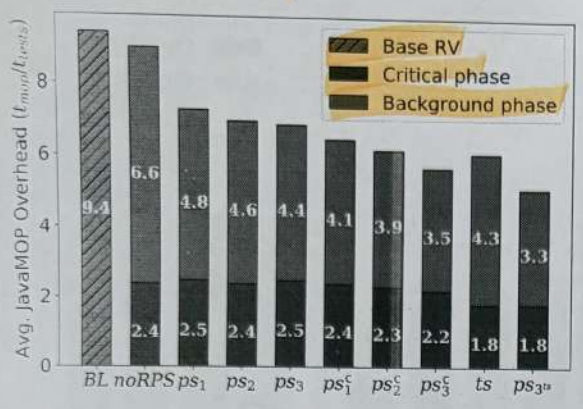

Fig. 8: Runtime overheads with Regression Property Prioritization's critical (*cr*) and background (*bg*) phases

in the 16 versions with new violations, to see why weak variants were safe. We found that all new violations happened due to events in $affected_3$, a subset of $affected_1$ and $affected_2$, so all the variants were able to catch the new violations. This further explains why weak RPS variants were safe in our experiments (Section V-C showed that excluding library classes did not lead to missing new violations in the projects).

E. RQ3: RPP Effectiveness

Fig. 8 shows RPP results, where the RPS variants do not exclude libraries. The first finding is that RPP alone (*noRPS*) overhead for monitoring background properties (*bg*) is roughly three times more than its overhead for monitoring critical properties (*cr*). The second finding is that overheads for *cr* and *bg* do not sum up to base RV overhead. For all projects, but one, *cr*+*bg noRPS* is greater than *BL*. Being greater is expected because time to run tests is repeated between *cr* and *bg*. The surprising project is opentripplanner, where the sum of *cr* and *bg* is less than *BL* for monitoring all properties together, likely due to reduced memory pressure when the properties are split. Finally, *cr* with $ps_{3^{ts}}$ has only 1.8× overhead, compared with 9.4× for base RV.

VI. Discussion

We highlight internal details of RPS and properties that contributed the most monitors and events in each project.

RPS Internals: Our analysis of data from running RPS shows that changes are small compared to the size of the program, and that our analysis is very fast compared to the time for monitoring. The data here is for ps_1. The average CDG in our experiments had 720 nodes and 3706 edges. On average, Δ contained 7 nodes each version, leading to an average of 233 nodes in $affected(\Delta)$. The total analysis time was 4.3% of the end-to-end time—this includes the time to find $affected(\Delta)$, repackage the Jar file with selected properties, and to find the classes from which new events may be generated after a code change. The rest of the time is spent on monitoring.

Different properties dominated base RV overhead: We measured the number of monitors created, events triggered, and the top two properties that contribute the most monitors (Top M) and events (Top E). No property always dominated monitor creation or event generation, but two properties, `Iterator_HasNext` and `StringBuilder_ThreadSafe` are in Top M for all projects. No property dominates Top E. `Iterator_HasNext` and `StringBuilder_ThreadSafe` are quite common in Top E and generate most events in opentripplanner, the project with the highest base RV overhead. `Iterator_HasNext` helped find several bugs [52], so developers may still want to monitor it. We are not aware that `StringBuilder_ThreadSafe` previously helped find bugs.

VII. Related Work

Many RV techniques and tools were proposed in almost two decades since the first papers on RV [33], [34], mostly concerned with speeding up RV on *one program version*. Example techniques (1) improve the efficiency of synthesizing monitors [35], (2) improve the efficiency of monitor garbage collection [42], [44], [58], (3) create a virtual machine to make RV more efficient in production [2], (4) reduce RV overhead by sharing information among monitors [20], [58], [72], (5) support efficient monitoring of properties written in different formalism [5], [32], [59], [60], (6) analyze observed executions in monitors to infer characteristics of unseen executions [12], [13], (7) allow RV to monitor multiple properties in one execution [44], [58], (8) reduce the time that RV wastes in loops [71], etc. Tools include Eagle [3], JavaMOP [40], [43], [58], jMonitor [45], JPaX [33], MarQ [74], MOPBox [10], Mufin [20], Ruler [5], and TraceMatches [1], [11]. We used JavaMOP because it is publicly available [40]. A complementary line of research is to automatically mine the properties that RV can monitor [6], [15], [19], [25], [46], [48], [49], [55], [62], [63], [67]–[69], [73], [76], [85], [86], [90].

Our recent large-scale study [52] showed that RV can find many new bugs during testing, but at high overhead of 4.3× for base RV, with the extra time incurred by JavaMOP (4.08s – 12.48s) being too small for RPS to be beneficial.

VIII. Conclusions

We presented three evolution-aware RV techniques: RPS, VMS, and RPP. Our techniques reduced base RV overhead from 9.4× to as low as 1.8×, were safe, and showed two orders of magnitude fewer violations than base RV. Our results provide strong evidence that taking evolution into account can significantly improve base RV.

Acknowledgments

We thank Milos Gligoric, Alex Gyori, Farah Hariri, Sasa Misailovic, August Shi, Andrei Stefanescu, Tianyin Xu, members of the Formal Systems Laboratory and CS591SE participants at UIUC for discussions. This work was partially supported by National Science Foundation grants CCF-1421503, CCF-1421575, CNS-1619275, CNS-1646305, CNS-1740916, and CCF-1763788. We gratefully acknowledge support from IOHK, Microsoft, Qualcomm, and Runtime Verification, Inc.