Techniques for Evolution-Aware Runtime Verification

Owolabi Legunsen, Yi Zhang, Milica Hadži-Tanović, Grigore Roșu, Darko Marinov
ICST 2019
4/26/2019
Runtime Verification (RV)

- RV dynamically checks program executions against formal properties, whose violations can help find bugs
  - a.k.a. runtime monitoring, runtime checking, monitoring-oriented programming, typestate checking, etc.

- RV has been around for decades, now has its own conference (RV)

- Many RV tools:
JavaMOP: a representative RV tool
Example property: Collection_SynchronizedCollection (CSC)

```java
public static <T> Collection<T> synchronizedCollection(Collection<T> c)

It is imperative that the user manually synchronize on the returned collection when iterating over it:

    Collection c = Collections.synchronizedCollection(myCollection);
    ...
    synchronized (c) {
        Iterator i = c.iterator(); // Must be in the synchronized block
        while (i.hasNext())
            foo(i.next());
    }
```

Failure to follow this advice may result in non-deterministic behavior.
TestNG example: from RV of test executions to bugs

Manual inspection: multiple threads can access “im”

CSC was violated on... SuiteHTMLReporter.java:66... a synchronized collection was accessed in thread-unsafe manner

JavaMOP

SuiteHTMLReporter

TestOnClassListener

65:  im = Collections.synchronizedList(...);
66:  for (IInvokedMethod iim : im) { ... }
RV during Continuous Integration (CI)?

1. Commit Changes
2. Fetch Changes
3. Build
4. Test
5. Pass/Fail
6. Release/Deploy

- Observation: All prior RV techniques are evolution-unaware (Base RV)
- Base RV would re-incur entire overhead if re-run after each code change

Developers

Version Control

CI Server

Build

Test

New Idea: Focus RV on code changes?

Contribution: the first techniques that adapt RV to evolving systems

0.97% of classes changed on average in our experiments
Contribution: Evolution-aware Runtime Verification

• Goal: leverage software evolution to scale RV better during testing

• Intended benefits:
  1. Reduce accumulated runtime overhead of RV across multiple program versions
  2. Show developers only new violations after code changes

• Complementary to techniques that improve RV on single program versions
  • Faster RV algorithms for single program versions
  • Running tests in parallel
  • Improve properties to have fewer false alarms
How JavaMOP works

- **CSC Properties**
- **Code + Tests**
- **Instrumentation**
- **Instrumented Code + Tests**
- **Execution**

- **Violations**
- **Monitors**
- **Events**

Code + Tests:
- Collections.synchronizedList()
- Collection+.iterator()
We proposed three evolution-aware RV techniques

1. **Regression Property Selection (RPS)**
   - Re-monitors only properties that can be violated in parts of code affected by changes

2. **Violation Message Suppression (VMS)**
   - Shows only new violations after code changes

3. **Regression Property Prioritization (RPP)**
   - Splits RV into two phases:
     - **critical phase**: check properties more likely to find bugs on developer’s critical path
     - **background phase**: monitor other properties outside developer’s critical path

The three techniques can be used together
Evolution-aware RV in JavaMOP

1. Regression Property Selection (RPS)
2. Violation Message Suppression (VMS)
3. Regression Property Prioritization (RPP)
Evolution-aware RV – Result Overview

- RPS and RPP significantly reduced accumulated runtime overhead of Base RV
  - Average: from 9.4x to 1.8x
  - Maximum: from 40.5x to 4.2x
- VMS showed 540x fewer violations than Base RV
- RPS did not miss any new violation after code changes
  - In theory can miss, but empirically it did not
- See paper for details on VMS and RPP
Base RV during software evolution

- Base RV re-monitors all properties after every code change
- No knowledge of dependencies in the code, or between code and properties

Old Version: monitor CSC, P1, P2
New Version: re-monitor CSC, P1, P2
\[ \Delta = \{B\} \]
Regression Property Selection (RPS) Overview

Selected subset of properties are those that may generate new violations.
Regression Property Selection (RPS) – step 1

Re-monitors only properties that can be violated in parts of code affected by changes

\[ \Delta = \{B\} \]

Step 1a: Build Class Dependency Graph (CDG) for new version

Step 1b: Map classes to properties for which the classes may generate events
Regression Property Selection (RPS) – step 2

Re-monitors only properties that can be violated in parts of code affected by changes

\[ \Delta = \{B\} \]

Step 2: Compute affected classes

Affected classes: those that generate events that can lead to new violations after code changes

Class X is affected if
1. X changed or is newly added
2. X transitively depends on a changed class, or
3. Class Y that satisfies (1) or (2) can transitively pass data to X
Regression Property Selection (RPS) – Steps 3 & 4

Re-monitors only properties that can be violated in parts of code affected by changes

\[ \Delta = \{B\} \]

Step 3: Select affected properties – those for which affected classes may generate events

Step 4: Re-monitor affected properties: \{CSC, P1\}

- P2 is NOT re-monitored in the new version
- Affected classes cannot generate P2 events
- Saves time to monitor P2; does not show old P2 violations

\[ \Delta = \{B\} \]
Total RPS time must be less than Base RV time

Analysis
- Step 1a: Build Class Dependency Graph (CDG) for new version
- Step 1b: Map classes to properties for which they may generate events
- Step 2: Compute affected classes
- Step 3: Select affected properties
- Step 4: Re-monitor only affected properties

Re-monitoring
- Static and Fast
- 4.3% of RPS time

Base RV (Re-monitor all properties)

Time Savings
Total Time for RPS

Analysis Re-monitoring
RPS Safety and Precision - Definitions

- Evolution-aware RV is **safe** if it finds all new violations that Base RV finds.
- Evolution-aware RV is **precise** if it finds only new violations that Base RV finds.
- RPS discussed so far is safe but not precise.
  - Safe modulo CDG completeness, test-order dependencies, dynamic language features.
Results of Safe RPS – ps₁

• 20 versions each of 10 GitHub projects
  • Average project size: 50 KLOC
  • Average test running time without RV: 51 seconds

How can we improve these results?
RPS variants that use fewer affected classes

Goal: Reduce RV overhead by varying “what” set of affected classes is used to select properties

\[ \Delta = \{ B \} \]

<table>
<thead>
<tr>
<th>What classes are used to select properties?</th>
<th>ps_1</th>
<th>ps_2</th>
<th>ps_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changed classes (i.e., ( \Delta ))</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dependents of ( \Delta )</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dependees of ( \Delta )</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Dependees of ( \Delta )’s Dependents</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Inheritance or Use May Generate events for
Using fewer affected classes can be (un)safe, e.g., \( ps_2 \)

\[
\Delta = \{B\}
\]

```java
class B {
    - public static boolean b = false;
    + public static boolean b = true;
}

class C {
    void getF() {
        D.foo(B.b);
    }
}

class D {
    static void foo(boolean b) {
        if (b) { // P1 events
        D.foo(B.b);
        }
    else { // No P1 events
    }
}
```

\( ps_2 \) can be safe if C does not pass data to D
RPS variants that instrument fewer classes

Goal: Reduce RV overhead by varying “where” selected properties are instrumented

\[ \Delta = \{B\} \]

<table>
<thead>
<tr>
<th>Where selected properties are instrumented (i ∈ {1,2,3})</th>
<th>ps_i</th>
<th>ps_i^c</th>
<th>ps_i^l</th>
<th>ps_i^{cl}</th>
</tr>
</thead>
<tbody>
<tr>
<td>affected((\Delta))</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>affected((\Delta))^c</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>third-party libraries</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

- have fewer violations
- ~36% of RV overhead
- excluding them can be safe
RPS Variants – Expected Efficiency/Safety Tradeoff

“more efficient than”

“less safe than”

2 Strong RPS variants are safe under certain assumptions: $ps_1$ and $ps_1^c$

10 Weak RPS variants are unsafe; they trade safety for efficiency
RPS Results – Runtime Overhead

![Bar chart showing average JavaMOP overhead for different RPS variants and the base RV. The chart compares the overhead of BL, $ps_1$, $ps_1^c$, $ps_2$, $ps_3$, $ps_2^c$, $ps_3^c$, $BL^\ell$, $ps_1^\ell$, $ps_2^\ell$, $ps_3^\ell$, $ps_3^c$, and $ps_2^c$. The overhead values are indicated by the height of the bars, with the base RV having the highest overhead at 9.4, and the RPS variants showing a decreasing trend towards 2.5.](image-url)
RPS Results – Violations Reported

Base RV

RPS Variants

Avg. Number of Violations

<table>
<thead>
<tr>
<th>Base RV</th>
<th>RPS Variants</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL</td>
<td>BL</td>
</tr>
<tr>
<td>ps1</td>
<td>ps1^l</td>
</tr>
<tr>
<td>ps2</td>
<td>ps2^l</td>
</tr>
<tr>
<td>ps3</td>
<td>ps3^l</td>
</tr>
<tr>
<td>ps4</td>
<td>ps4^l</td>
</tr>
<tr>
<td>ps5</td>
<td>ps5^l</td>
</tr>
<tr>
<td>ps6</td>
<td>ps6^l</td>
</tr>
<tr>
<td>ps7</td>
<td>ps7^l</td>
</tr>
<tr>
<td>ps8</td>
<td>ps8^l</td>
</tr>
<tr>
<td>ps9</td>
<td>ps9^l</td>
</tr>
<tr>
<td>ps10</td>
<td>ps10^l</td>
</tr>
</tbody>
</table>

0.1 vms
RPS Results – precision and safety

• VMS is precise – it shows only new violations
  • RPS is not precise – it shows two orders of magnitude more violations than VMS

• We manually confirmed whether all RPS variants find all violations from VMS

• Surprisingly, all weak RPS variants were safe in our experiments
Why weak RPS variants were safe in our experiments

• 75% of event traces observed by monitors involved only one class

• 32 of 33 new violations were due to changes whose effects are in $ps_3$
  • Additional scenarios captured by $ps_1$ and $ps_2$ did not lead to new violations
  • We may have missed old violations when not tracking $ps_1$ or $ps_2$ scenarios

• 87% of old violations missed by excluding third-party libraries did not involve any event from the code
Regression Property Prioritization (RPP)

Combining RPS+RPP reduced RV overhead to 1.8x (from 9.4x)
Conclusion

• We proposed three evolution-aware RV techniques: RPS, VMS, RPP

• Our techniques reduced Base RV overhead from 9.4× to as low as 1.8×

• Taking evolution into account can significantly reduce Base RV overhead during software evolution

Owolabi Legunsen: legunse2@illinois.edu
Yi Zhang: yzhng173@illinois.edu
Milica Hadži-Tanović: milicah2@illinois.edu

Grigore Roșu: grosu@illinois.edu
Darko Marinov: marinov@illinois.edu