CS 5154
Integrating Runtime Verification with Software Testing

Spring 2021

Owolabi Legunsen
Software has become more critical to most aspects of our daily lives.
The risk posed by software failure has also grown.

Report: Software failure caused $1.7 trillion in financial losses in 2017

Software testing company Tricentis found that retail and consumer technology were the areas most affected, while software failures in public service and healthcare were down from the previous year.

By Scott Matteson | January 26, 2018, 7:54 AM PST
Continuous Integration (CI): rapid test/release cycles

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

Developers

Version Control

CI Server

Builds per day:
- Facebook: 60K*
- Google: 17K
- HERE: 100K
- Microsoft: 30K
- Single open-source projects: up to 80

Releases per day
- Etsy: 50

Several important problems exist in these cycles

**P1: Passing tests miss bugs**

S1: Find more bugs from tests that developers already have

**P2. Failed tests, no buggy changes**

S2: Find bugs more reliably by detecting such failures

**P3. Testing can be very slow**

S3: Find bugs faster by speeding up testing

**P4. How to test in new domains?**

S4: Find bugs in emerging application domains

Developers

The problem that we’ll talk about today is... 

Problem: Passing tests miss bugs

Our Solution: Use Runtime Verification to find more bugs from tests that developers already have

In this lecture

- Integrating a lightweight formal method called runtime verification with everyday software testing
- Benefits (find more bugs earlier)
- Challenges (high overheads)
- Progress on resolving some of the challenges
Introduction to 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

• Many RV tools:
One reason why RV is appealing

Can RV help bring some of the mathematical rigor of formal verification to everyday software development?
No study of RV during testing of real-world software

Contribution: the first study of RV during testing of real-world software

All prior RV techniques targeted post-release runs
Contribution: the first techniques that adapt RV to evolving systems

No previous RV techniques for evolving systems
JavaMOP: a representative RV tool
Example property: Collection_SynchronizedCollection (CSC)

```
synchronizedCollection

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.
```
CSC property in JavaMOP

1. Collections_SynchronizedCollection (Collection c, Iterator i) {
2.   Collection c;
3.   creation event sync after() returning (Collection c):
4.   call (Collections.synchronizedList(Collection)) ... 
5.   event syncMk after (Collection c) returning (Iterator i):
6.   call (Collection+.iterator()) && target (c) && condition (Thread.holdsLock(c)) {} 
7.   event asyncMk after (Collection c) returning (Iterator i):
8.   call ( Collection+.iterator() && target(c) && condition (!Thread.holdsLock(c)) {} 
9.   event access before (Iterator i):
10.  call ( Iterator.*(..)) && target (i) && condition (!Thread.holdsLock(this.c)) {} 
11.  ere : ( sync asyncMk) | (sync syncMk access) 
12.  @match { RVMLogging.out.println ( Level.CRITICAL, __DEFAULT_MSG); ... } 
13.}
Other example properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Nature of bug found</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringTokenizer_HasMoreElements</td>
<td><strong>Crash:</strong> don’t fetch elements from an empty collection</td>
</tr>
<tr>
<td>ByteArrayOutputStream_FlushBeforeRetrieve</td>
<td><strong>Correctness:</strong> don’t read streams with incomplete data</td>
</tr>
<tr>
<td>InetSocketAddress_Port</td>
<td><strong>Performance:</strong> don’t use too many ephemeral ports</td>
</tr>
</tbody>
</table>
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

... CSC
... SuiteHTMLReporter
... 65:  im = Collections.synchronizedList(...);
... 66:  for (IInvokedMethod iim : im) { ... }
...
How JavaMOP works

Collections.synchronizedList()
Collection+.iterator()
Example: finding bugs from RV of test executions

1. CSC (Collection c, Iterator i) {
2.   Collection c;
3.   creation event sync after() returning (Collection c):
4.      call (Collections.synchronizedList(Collection)) || ... { this. c = c; }
5.   event asyncMk after (Collection c) returning (Iterator i):
6.      call (Collection+.iterator() && target(c) && !Thread.holdsLock(c) {})
... 

11. ere : (sync asyncMk) | ....
12. @match { RVMLogging.out.println ( Level.CRITICAL, __DEFAULT_MSG); ... }}

65. im = Collections.synchronizedList(...);
66. for (IInvokedMethod iim : im) { ... }

Spec Violations
CSC was violated on... (SuiteHTMLReporter.java:65)... synchronized collection accessed in thread-unsafe manner
How is RV different from testing?

Specified independent of code+tests
Specified in mathematical logic

Automatically generated to check the properties
Contribution: large-scale study of RV during testing

We conducted our study to answer the following questions:

• How many additional bugs does RV help find during testing?

• How high is RV overhead during testing?

• How often do property violations not indicate true bugs?
Properties used in our study

- Formal specifications of correct standard Java library API usage
- Manually written\textsuperscript{[1]} or automatically mined \textsuperscript{[2]} by other researchers
- 161 manually written properties from 4 packages: java.lang, java.io, java.util, and java.net
- JavaMOP supports different formalism: LTL, CFG, FSM, ERE, SRS, etc.


Overview of our study

218 projects, 20K+ tests

Code + Tests ➔ JavaMOP ➔ Violations ➔ Manual Inspection ➔ Bugs ➔ Submit Pull Requests

Properties ➔ Violations ➔ Manual Inspection ➔ Not Bugs

6167 violations
1379 Manual Inspection
198 Bugs
1181 Not Bugs

95 Submit Pull Requests
Some of the projects where we found bugs

- Joda Time
- commons Math
- commons lang
- Apache Camel
- Hadoop
- ImgLib2
- GeoServer
- TestNG
- Gora
- Apache HBase
Summary of study results

• How many additional bugs does RV help find during testing?
  ✓ Total bugs found so far: 198
  ✓ So far: 95 bugs reported, 74 accepted, 3 rejected

• How high is RV overhead during testing?
  ✗ Up to 40x
  • e.g., 1min to 40min, 30mins to 10hours

• How often do property violations not indicate true bugs?
  ✗ 86% of ~1.4K violations were not bugs
Why are some violations (not) bugs?

```
65:  im = Collections.synchronizedList(...);
66:  for (IInvokedMethod iim : im) { ... }
```

TestNG accepted our pull requests for 13 CSC violations

XStream developers rejected our pull request for similar CSC bug

- “...there’s no need to synchronize it... as explicitly stated ..., XStream is not thread-safe ... this is documented ...”

Properties do not capture enough program context\(^1\)

---

\(^1\) S. Thummalapenta and T. Xie. Alattin: Mining Alternative Patterns for Detecting Neglected Conditions. ASE 2009
Logistics

• Homework 4 is released
  • Work in your project group
  • Due 5/10/2021

• Project Sprint 2 will be released soon
  • Focus: using testing JavaMOP and/or Randoop
  • Due 5/14/2021 (last day of classes)
Reflecting on the study results

• RV overhead is still high despite decades of tremendous research progress
  • Overhead in machine time (up to 40x)
  • Overhead in developer time to inspect violations (1200 hours / 1379 violations)
  • Yet, RV helped find many bugs from existing tests

• Do we need faster RV algorithms and better properties? Yes!

• But what if we also consider how developers are likely to use RV?
RV during Continuous Integration (CI)?

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

New Idea: Focus RV on code changes?

Code changes are typically very small relative to entire code base

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
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

Regression Property Selection (RPS)

Violation Message Suppression (VMS)

New?

Violations

Properties

Instrumentation

Instrumented Code + Tests

Execution

Monitors

Events

Regression Property Prioritization (RPP)

What?

Where?

Critical?

Code + Tests
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
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
Total RPS time must be less than Base RV time

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

Analysis

Re-monitoring

Static and Fast

4.3% of RPS time

Base RV (Re-monitor all properties)

Analysis

Re-monitoring

Time Savings

Total Time for RPS

Static and Fast
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 – $ps_1$

- 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>ps₁</th>
<th>ps₂</th>
<th>ps₃</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>
Using fewer affected classes can be (un)safe, e.g., $ps_2$

$$\Delta = \{B\}$$

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}
    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

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

- \( \Delta \) = \{B\}

<table>
<thead>
<tr>
<th>Where selected properties are instrumented (( i \in {1,2,3} ))</th>
<th>ps_1</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 – average runtime overhead

![Bar chart showing average JavaMOP overhead for various RPS variants and Base RV.](image-url)
Excluding third-party libraries does not miss many violations on average.

RPS Results – no. of violations reported

<table>
<thead>
<tr>
<th>Base RV</th>
<th>RPS Variants</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL</td>
<td>54</td>
</tr>
<tr>
<td>BL(\ell)</td>
<td>50</td>
</tr>
<tr>
<td>(ps_1)</td>
<td>42</td>
</tr>
<tr>
<td>(ps_1^\ell)</td>
<td>40</td>
</tr>
<tr>
<td>(ps_2)</td>
<td>40</td>
</tr>
<tr>
<td>(ps_2^\ell)</td>
<td>37</td>
</tr>
<tr>
<td>(ps_1^c)</td>
<td>37</td>
</tr>
<tr>
<td>(ps_1^{\ell c})</td>
<td>34</td>
</tr>
<tr>
<td>(ps_3)</td>
<td>34</td>
</tr>
<tr>
<td>(ps_3^\ell)</td>
<td>32</td>
</tr>
<tr>
<td>(ps_2^c)</td>
<td>27</td>
</tr>
<tr>
<td>(ps_2^{\ell c})</td>
<td>24</td>
</tr>
<tr>
<td>(ps_3^c)</td>
<td>23</td>
</tr>
<tr>
<td>(ps_3^{\ell c})</td>
<td>21</td>
</tr>
</tbody>
</table>

Avg. Number of Violations
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)
Where do we (want to) go from here?

Can we make RV scale like testing and have guarantees of verification?
Some steps that can get us closer...

• Obtain better properties to monitor
  • 85% false alarm rate is a very hard sell!

• Reduce the developer overhead of inspecting violations
  • Hint: We already tried Machine Learning (ICST’20)

• Scale RV to (ultra-)large software ecosystems
  • Most important software are being developed in monorepositories

• Improve the coverage of the tests (wrt to the properties)
  • Otherwise, we cannot have high guarantees
Summary

Overview of our study

218 projects, 20K+ tests

Code + Tests

GitHub

JavaMOP

Violations

Manual Inspection

Properties

6167

1379

198

Bugs

Not Bugs

1181

95

Submit Pull Requests

RPS Variants – Expected Efficiency/Safety Tradeoff

“more efficient than”

“less safe than”

2 Strong RPS variants are safe under certain assumptions: \( p_{s_1} \) and \( p_{s_1}^c \)

10 Weak RPS variants are unsafe; they trade safety for efficiency

RPS Results – average runtime overhead

Where do we (want to) go from here?

Formal Verification:
Prove mathematically that a program is correct

RV: Check that program executions are correct

Testing: Check if subset of program inputs gives correct output

Can we make RV scale like testing and have guarantees of verification?