

# How Good Are the Specs?

A Study of the Bug-Finding Effectiveness of  
Existing Java API Specifications

**Owolabi Legunsen**, Wajih Ul Hassan, Xinyue Xu  
Grigore Rosu and Darko Marinov

ASE 2016

Singapore, Singapore

September 7, 2016



CCF-1421503, CCF-1421575, CCF-1438982, CCF-1439957



**ILLINOIS**  
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

TM

# What is a Specification (Spec)?

“A spec is **a way to use an API** as asserted by the developer or analyst, and **which encodes information about the behavior of a program** when an API is used”

--Robillard et al.<sup>[\*]</sup>

- Violating a spec **may or may not** be a bug

[\*] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. TSE, 39(5):613–637, 2013.

# An Example Spec in our Study - CSC

- CSC = Collections\_SynchronizedCollection
- CSC is specified in the Javadoc for java.util.Collections:

“It is **imperative** that the user manually synchronize on the returned collection when iterating over it ... Failure to follow this advice **may** result in non-deterministic behavior” [\*]

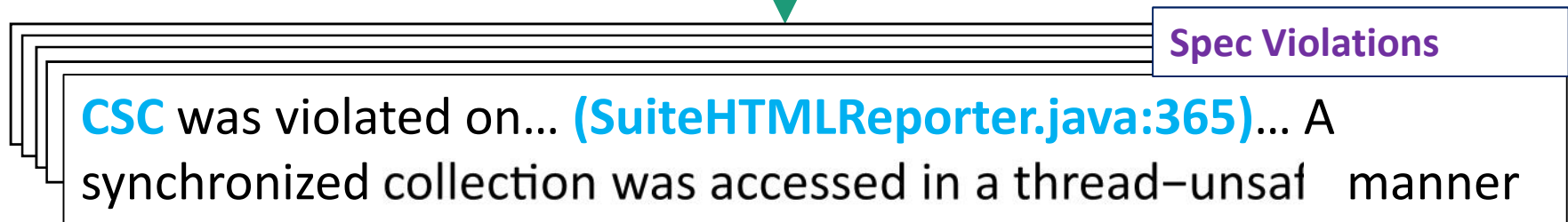
- CSC was formalized to enable checking this spec

[\*] [https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedCollection\(java.util.Collection\)](https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedCollection(java.util.Collection))

# CSC Formalized in JavaMOP

- JavaMOP is a runtime verification tool that can check program executions against formal specs
1. Collections\_SynchronizedCollection (Collection c, Iterator i) {
  2.   Collection c;
  3.   creation **event** sync after() **returning** (Collection c):
  4.     **call** (Collections.synchronizedCollection(Collection)) || ... { **this** . c = c ; }
  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. }

# Illustrative Example



```
364 im = Collections.synchronizedList(...);  
365 for (IInvokedMethod iim : im) { ... }
```



Line 365 invokes `im.iterator()` without first synchronizing on `im`

Pull Request

```
364 im = Collections.synchronizedList(...);  
365 + synchronized (im) {  
366     for (IInvokedMethod iim : im) { ... }  
367 + }
```

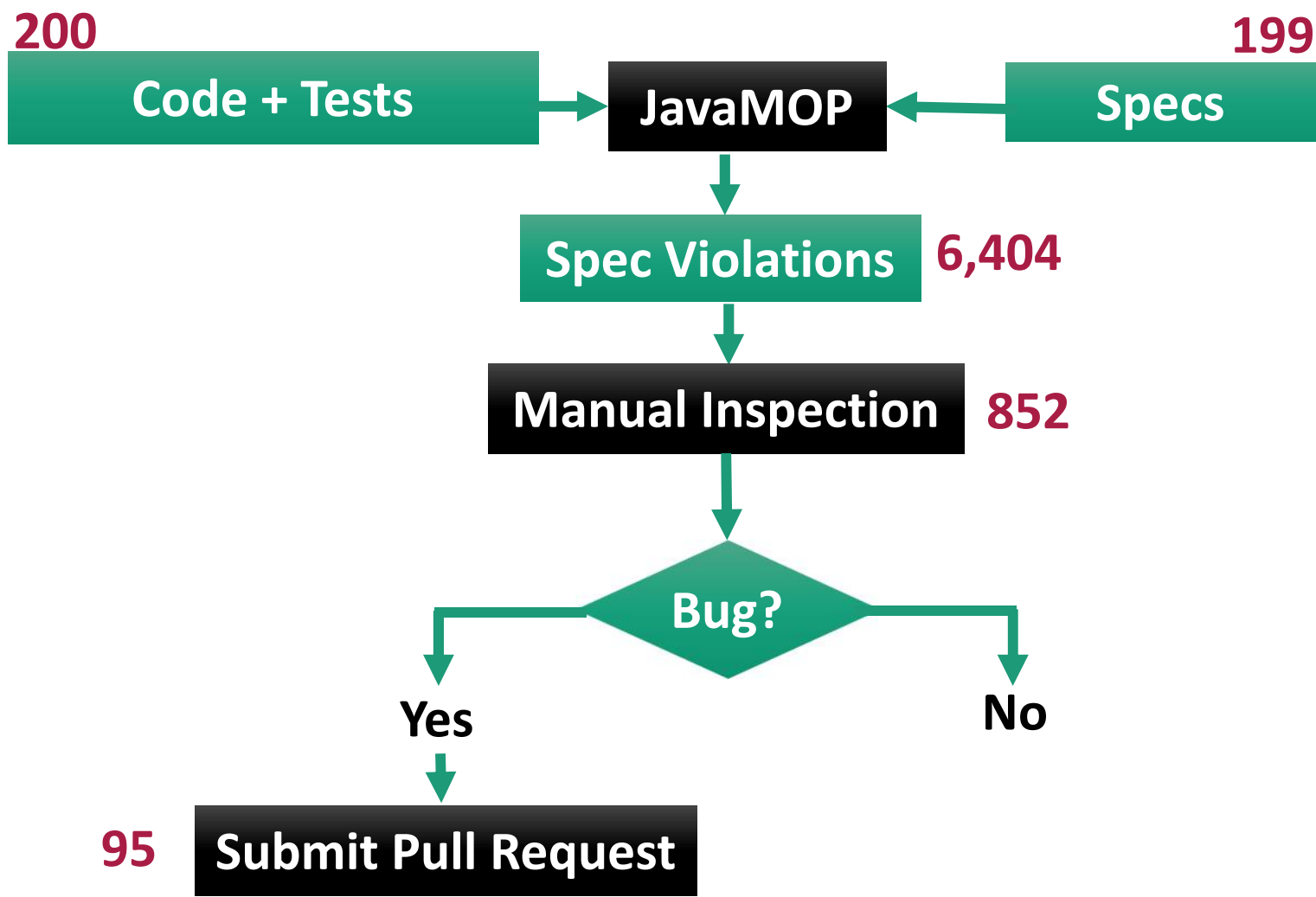
Accepted by TestNG developers

Rejected by XStream developers

# Specs in SE Research

- Researchers have proposed many specs by writing manually or mining automatically
- This is the first large-scale study of the effectiveness of these specs for finding bugs during testing
- **An effective spec catches true bugs without generating too many false alarms**

# Overview of Our Study



# Experimental Subjects



- 200 open-source projects were selected from GitHub
  - Average project size: 6 KLOC
  - Average number of tests: 90.3
- Each selected project satisfies four criteria:
  - ✓ Uses Maven (for ease of automation)
  - ✓ Contains at least one test
  - ✓ Tests pass when not monitored with JavaMOP
  - ✓ Tests pass when monitored with JavaMOP



# Specs Used in our Study

- 182 manually written specs formalized by Luo et al. [1]
- 17 automatically mined specs provided by Pradel et al. [2]
- All specs in our study are publicly available online

[1] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Serbanuta, and G. Rosu. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In RV, pages 285–300, 2014.

[2] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In ICSE, pages 925–935, 2012.

# Tools Used in our Study

- JavaMOP (runtime verification tool)
  - Easy to use: integrate into pom.xml and run “mvn test”
  - JavaMOP allows to monitor multiple specs simultaneously
- Randoop (automatic test generation tool)
  - **Does type of tests affect the bug-finding effectiveness of specs?**
  - We generated tests for 122 of 200 projects
  - Average number of generated tests = 17.5K
  - Total number of generated tests = 2.1M

# Inspecting & Classifying Violations

- We inspected 852 (of 6,404) unique spec violations
  - We did not inspect violations from 21 manually written specs
  - We sampled 200 violations of 1,141 automatically mined specs
- Multiple co-authors inspected most violations
- Classification
  - FalseAlarm (716)
  - TrueBug (114)
  - HardToInspect (22)

# Research Questions

- RQ1: What is the runtime overhead of monitoring?
  - ✓ Runtime overhead: 4.3x
- RQ2: How many bugs are found from spec violations?
  - ✓ We reported 95 bugs: 74 accepted, 3 rejected so far
- RQ3: What are the false alarm rates among violations?
  - ✗ 82.81% for manually written specs
  - ✗ 97.89% for automatically mined specs

# RQ1: Time Overhead of Monitoring

$$\textit{Overhead} = \frac{\textit{mop} - \textit{base}}{\textit{base}}$$

mop: time to run tests with monitoring

base: time to run tests without monitoring

- Average overhead: 4.3x
- Average additional time: 12.5s
- Specs are monitored simultaneously

# RQ2: Bugs in Subject Programs

	<b>Count</b>	<b>Breakdown</b>	
<b>Total TrueBugs</b>	<b>114</b>	<b>From manual specs</b>	<b>110</b>
		<b>From auto specs</b>	<b>4</b>
<b>Unique TrueBugs</b>	<b>97</b>		
<b>Already fixed TrueBugs</b>	<b>2</b>		
<b>Reported TrueBugs</b>	<b>95</b>	<b>Accepted</b>	<b>74</b>
		<b>Rejected</b>	<b>3</b>
		<b>Pending</b>	<b>18</b>

- Bugs accepted in Joda-Time, TestNG, XStream, BCEL, etc.

## RQ3: False Alarm Rates (FAR)

$$FAR = \frac{FalseAlarms}{FalseAlarms+TrueBugs} * 100\%$$

- FAR = 82.81 % for manually written specs
- FAR = 97.89 % for automatically mined specs
- All inspected violations were in 99 projects:

FAR [%]	
FAR = 100%	69
50% ≤ FAR < 100%	20
0% ≤ FAR < 50%	3
FAR = 0%	7



# RQ3: FAR vs. Project Characteristics

Type of specs		FAR [%]
Manually written specs		82.81
Libraries		86.55
Project code		80.82
Single-module		81.87
Multi-module		86.23
Manually written tests		82.51
Automatically generated tests		84.21
Automatically mined specs		97.89
Libraries		100.00
Project code		94.87
Single-module		97.84
Multi-module		98.04

FAR was very high along all dimensions considered

Slightly higher FAR in libraries than in project code



# RQ3: FAR among Inspected Specs

Manually written specs	Count	FAR	Count
Total	182	FAR = 100%	31
Number of specs not violated	119	$50\% \leq \text{FAR} < 100\%$	6
Number of specs not inspected	21	$0\% \leq \text{FAR} < 50\%$	4
Number of inspected specs	42	FAR = 0%	1

- Only 11 of 182 manually written specs helped find a bug
- Only 3 of 17 automatically mined specs helped find a bug
  - FSM162, FSM33, and FSM373
  - 87.50%, 90.00% and 98.06% FAR, respectively

# Example False Alarm

- Consider the `Iterator_HasNext` spec: “`hasNext()` must return true before calling `next()` on an iterator”
  - 150 FalseAlarms, 97.40% FAR

**Highlighted** `Iterator_HasNext` violation is a false alarm

```
1 ArrayList<Integer> list = new ArrayList<>(); list.add(1);
2 Iterator<Integer> it = list.iterator();
3 if ( it.hasNext() ){ int a = it.next();}
4 if ( list.size() > 0 ){ int b = list.iterator().next();}
```

# Rejected Pull Requests



- XStream (a CSC violation)
  - “...there’s no need to synchronize it... As explicitly stated ..., XStream is not thread-safe ... this is documented ...”
- JSqlParser (no check for validity of s in parseLong(s, int) )
  - “...parser ... ensures that only long values are passed ... do you have a ... SQL, that produces a NumberFormatException?”
- threerings.playn (stream not flushed)
  - “[class] automatically flushes the target stream when done() is called ... an additional flush is unnecessary.”

# Positive Developer Responses



- Developers asked us for more fixes
  - “I found the following... Can you please check these out as well?”
- Developers accepted better exception messages
  - “Looks good, I’ll ... add that more helpful error message.”
- Developers liberally accepted some pull requests
  - “While I’m not convinced it is necessary, this will cause no harm.”

# Recommendations for the Future

- Open and community-driven spec repositories
  - We could have evaluated more specs if these existed
- More work on spec testing and filtering of false alarms
- Greater emphasis on bug-finding effectiveness
  
- Better categorization of specs
- Complementing benchmarks with OSS
- Confirming spec violations with developers

# Conclusions

- The first large-scale evaluation of existing Java API specs
  - ✓ 199 specs and 200 open-source projects
  - ✓ Average runtime overhead was 4.3x
  - ✓ Found many bugs that developers are willing to fix
  - ✗ False alarm rates are too high
- We made some recommendations for future research
- Study data is online: <http://fsl.cs.illinois.edu/spec-eval>

legunse2@illinois.edu