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
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.[*]

• Violating a spec may or may not be a bug

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)
CSC Formalized in JavaMOP

- JavaMOP is a runtime verification tool that can check program executions against formal specs

```java
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

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

**Spec Violations**

**CSC** was violated on... *(SuiteHTMLReporter.java:365)*... A synchronized collection was accessed in a thread-unsafe manner

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

**Pull Request**

```java
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

200 Code + Tests → JavaMOP → Spec Violations
6,404
Specs
852 Manual Inspection
852 Bug?
Yes
No
95 Submit Pull Request
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

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

\[
\text{Overhead} = \frac{\text{mop} - \text{base}}{\text{base}}
\]

- 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

<table>
<thead>
<tr>
<th></th>
<th>Count</th>
<th>Breakdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total TrueBugs</td>
<td>114</td>
<td>From manual specs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>From auto specs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Unique TrueBugs</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>Already fixed TrueBugs</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Reported TrueBugs</td>
<td>95</td>
<td>Accepted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>74</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rejected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pending</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18</td>
</tr>
</tbody>
</table>

- Bugs accepted in Joda-Time, TestNG, XStream, BCEL, etc.
RQ3: False Alarm Rates (FAR)

\[
FAR = \frac{False\text{Alarms}}{False\text{Alarms} + True\text{Bugs}} \times 100\%
\]

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

<table>
<thead>
<tr>
<th>FAR [%]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FAR = 100%</td>
<td>69</td>
</tr>
<tr>
<td>50% ≤ FAR &lt; 100%</td>
<td>20</td>
</tr>
<tr>
<td>0% ≤ FAR &lt; 50%</td>
<td>3</td>
</tr>
<tr>
<td>FAR = 0%</td>
<td>7</td>
</tr>
</tbody>
</table>
RQ3: FAR vs. Project Characteristics

<table>
<thead>
<tr>
<th>Type of specs</th>
<th>FAR [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manually written specs</td>
<td>82.81</td>
</tr>
<tr>
<td>Libraries</td>
<td>86.55</td>
</tr>
<tr>
<td>Project code</td>
<td>80.82</td>
</tr>
<tr>
<td>Single-module</td>
<td>81.87</td>
</tr>
<tr>
<td>Multi-module</td>
<td>86.23</td>
</tr>
<tr>
<td>Manually written tests</td>
<td>82.51</td>
</tr>
<tr>
<td>Automatically generated tests</td>
<td>84.21</td>
</tr>
<tr>
<td>Automatically mined specs</td>
<td>97.89</td>
</tr>
<tr>
<td>Libraries</td>
<td>100.00</td>
</tr>
<tr>
<td>Project code</td>
<td>94.87</td>
</tr>
<tr>
<td>Single-module</td>
<td>97.84</td>
</tr>
<tr>
<td>Multi-module</td>
<td>98.04</td>
</tr>
</tbody>
</table>

FAR was very high along all dimensions considered.

Slightly higher FAR in libraries than in project code.
RQ3: FAR among Inspected Specs

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

### Table

<table>
<thead>
<tr>
<th>Manually written specs</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>182</td>
</tr>
<tr>
<td>Number of specs not violated</td>
<td>119</td>
</tr>
<tr>
<td>Number of specs not inspected</td>
<td>21</td>
</tr>
<tr>
<td>Number of inspected specs</td>
<td>42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FAR</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAR = 100%</td>
<td>31</td>
</tr>
<tr>
<td>50% ≤ FAR &lt; 100%</td>
<td>6</td>
</tr>
<tr>
<td>0% ≤ FAR &lt; 50%</td>
<td>4</td>
</tr>
<tr>
<td>FAR = 0%</td>
<td>1</td>
</tr>
</tbody>
</table>
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