Metric → Judged on the basis of FAR (False Alarm Rate), No. of bugs found + fixed, overhead

first large scale study for effectiveness of specs, instead of algorithms on reducing RV overhead.

How Good Are the Specs? A Study of the Bug-Finding Effectiveness of Existing Java API Specifications

Results 1. Specs do find bug, in tolerable overhead. 2. Developers are ready for fixes 3. High FAR

Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov
Department of Computer Science
University of Illinois at Urbana-Champaign, USA
{legunse2,whassan3,xxu52,rosu,marinov}@illinois.edu

ABSTRACT

RV / Problem - effectiveness

Runtime verification can be used to find bugs early, during software development, by monitoring test executions against formal specifications (specs). The quality of runtime verification depends on the quality of the specs. While previous research has produced many specs for the Java API, manually or through automatic mining, there has been no large-scale study of their bug-finding *effectiveness*.

USP

We present the first in-depth study of the bug-finding effectiveness of previously proposed specs. We used JavaMOP to monitor 182 manually written and 17 automatically mined specs against more than 18K manually written and 2.1M automatically generated tests in 200 open-source projects. The average runtime overhead was under 4.3×. We inspected 652 violations of manually written specs and (randomly sampled) 200 violations of automatically mined specs. We reported 95 bugs, out of which developers already fixed 74. However, most violations, 82.81% of 652 and 97.89% of 200, were false alarms.

Numbers

Results

Our empirical results show that (1) runtime verification technology has matured enough to incur tolerable runtime overhead during testing, and (2) the existing API specifications can find many bugs that developers are willing to fix; however, (3) the false alarm rates are worrisome and suggest that substantial effort needs to be spent on engineering better specs and properly evaluating their effectiveness.

CCS Concepts

•**Software and its engineering** → **Software testing and debugging;**

Keywords

runtime verification, specification quality, empirical study

1. INTRODUCTION

In runtime verification, the execution of a software system is dynamically checked against formal specifications (specs for short) [6, 8, 10, 15, 19, 35–37]. At a high level, the program being monitored is instrumented to capture, as *events*, method calls and field updates that are related to the specs being checked. Then, at runtime, the instrumented program creates listener objects, commonly referred to as *monitors*, which check that the events conform to the specs and report *violations* when some spec is violated. In this paper, a "spec" refers to a behavioral specification, defined by Robillard et al. [55] as *"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"*. A spec violation indicates that some API is used in a way that is not consistent with its usage guideline, but such violation may or may not be a real bug in the code.

RV + Monitors

spec violation

The potential for using runtime verification during software testing was previously recognized [24, 27, 31, 35], but combining testing with runtime verification of multi-object parametric specs, required by object-oriented API specs, only recently became practically feasible, thanks to research and development progress on (i) making parametric spec runtime verification more efficient [8,16,23,35,38,61], (ii) being able to monitor many specs simultaneously [2, 52], and (iii) better-engineered runtime verification tools [5, 24]. We recently proposed to combine runtime verification with regression testing, where test executions are monitored against formal specs to find bugs during software evolution [32].

Look into Major Problem this paper works on

The quality of specs has generally been taken for granted in the runtime verification research community, where the major research direction over the last decade has been to improve the efficiency and scalability of algorithms, techniques, and tools. The specs used in previous research were manually written [1,6,23,35] or automatically mined [3,9,13, 17,28,30,33,34,39,40,45–49,54,58,62,63]. These specs were monitored to measure their runtime overhead. However, for finding bugs by combining runtime verification with software testing, the *effectiveness* of these specs becomes critical.

→ old papers

In this paper, we present the first in-depth investigation of the effectiveness of existing specs for finding bugs. We consider a spec *effective for bug finding* if it can catch true bugs but does not generate too many false alarms. We consider specs of the standard Java API because such specs can potentially find bugs in many projects across various domains, require no domain knowledge, and the runtime verification tool that we evaluate, JavaMOP [21, 24, 35], works for Java. We evaluate 199 existing manually written and automatically mined specs. Specifically, we use 182 manually written specs that were formalized directly from the Java API documentation [31] and used in previous studies on the effi-

→ How do you categorise effectiveness?

Uses automata theory to find bugs on runtime from a black box through stack trace.

Usually used in U with testing (static / dynamic) + model checking

Read docs

ASE'16, September 3–7, 2016, Singapore, Singapore

http://dx.doi.org/10.1145/2970276.2970356

Basically, study how effective the existing (mined + manual) specs are in finding bugs, instead of improving the RV algorithms. Study is done on large scale, another USP.

ciency and scalability of runtime verification [32,35,52]. We also use 17 specs that were mined automatically from large traces [47] and were used in spec mining studies [48,49].

Our work differs from previous evaluations of specs in the runtime verification and spec mining literature in three major ways. First, previous runtime verification studies mostly focused on the *efficiency* of monitoring, but we focus on the *effectiveness* question: "How good are the specs?" Second, most previous evaluations were conducted on the DaCapo benchmarks [4] (with at most 14 projects) or with a smaller number of open-source projects; in contrast, we use 200 open-source projects. Our results thus provide fresh insights to researchers in both runtime verification and spec mining communities, because our evaluation is based on a substantially larger set of more diverse projects. We believe that evaluating specs on (current) open-source projects instead of (old) benchmarks can be more representative for assessing the effectiveness of specs from developers' point of view and should be strongly considered in future evaluations of specs. Third, in many previous studies, researchers assumed that any spec violations were bugs, or decided themselves what was a bug or not, but we submit bug reports and fixes (i.e., *pull requests*) to let the developers be the judges of the bugs we discovered by inspecting spec violations.

In our experiments, we monitored the 182 manually written and 17 automatically mined specs while running 18065 manually written and 2135081 automatically generated tests in 200 open-source projects, manually inspected a subset of the spec violations, and sent pull requests for the violations that we believed to be bugs. On the positive, the average runtime overhead of monitoring was under 4.3×, and developers already fixed 74 of 95 bugs that we reported. On the negative, we found a large rate of false alarms among the inspected violations. We inspected 652 of 5263 violations of manually written and 200 of 1141 violations of automatically mined specs, and observed overall false alarm rates of 82.81% and 97.89%, respectively. Further, only a small fraction of the specs led to the discovery of bugs—11 of 182 manually written and 3 of 17 automatically mined specs—and even among these, the average false alarm rates were high, 45.51% and 96.69%, respectively. We reported several issues about the existing specs, and JavaMOP maintainers already corrected some, but in many cases, the specs appear completely ineffective and should not be used at all.

Inspecting spec violations and submitting pull requests to developers took an estimated 1,200 hours and was challenging for three reasons. First, understanding the root cause of a violation is non-trivial. Although JavaMOP reports the line number for each spec violation, reasoning about a change that could correct the violation often requires deeper understanding of the code (and we were not developers on any of the 200 open-source projects); moreover, some of the violations were in third-party libraries, so we needed to comprehend parts of those libraries as well. Second, it is challenging to decide what constitutes an actual bug that should be submitted to the developers. At one extreme, we could only submit violations that can lead to program crashes. At the other extreme, we could simply submit every violation to the developers and see what they say, but this could unnecessarily burden the developers (who may then blacklist us or start to "desk-reject" our pull requests if they feel those are mostly useless to them). Even between these two extremes, it is debatable how to classify so-called "code smells" [17,39,49] which may indicate API misunderstanding by developers but are harmless in the current version of the code, e.g., calling `close()` on an `OutputStream` instance for which `close()` is a no op. Third, preparing a pull request in a way that developers would find useful requires substantial effort (another reason to not even attempt to submit every violation), and sometimes involved multiple internal iterations before submission. For all these reasons, we chose to report to the developers those cases where at least one of the authors believed that a violation indicated some problem in the current version of the code.

The results from our study show that the effort spent by the runtime verification community over the last decade on improving the performance of simultaneous monitoring of parametric specs has paid off. Indeed, the technology has matured enough to incur acceptable runtime overhead when monitoring test executions in open-source projects against dozens of specs. Also, the existing API specs from prior runtime verification and spec mining research can find many bugs that developers are willing to fix. However, the false alarm rates are worrisome and suggest that there is a need for the research community to fundamentally re-think spec finding and "spec engineering" approaches, towards making runtime verification a more effective early-stage, bug-finding aid that developers can use.

This paper makes the following contributions:

- ⋆ **Large-Scale Evaluation.** We present the first large-scale evaluation of runtime verification during software testing, with 199 specs and 200 open-source projects. The results show that monitoring has acceptable overhead during testing and can find important bugs, but the specs are largely ineffective and generate way too many false alarms.
- ⋆ **Analysis of Effectiveness.** We analyze reasons for bug-finding effectiveness of existing specs, in particular, the high rates of false alarms, and discuss developers' feedback on our pull requests.
- ⋆ **Recommendations and Data.** We provide a set of recommendations that can help the research community engineer more effective specs and better evaluate these specs. We also make data from our study publicly available [57].

2. BACKGROUND

We briefly describe runtime verification of specs in JavaMOP [10,21,24,35,36]. `Collection_SynchronizedCollection` (`CSC`), shown in Figure 1, is one of the specs in our study. `CSC` was proposed by Bodden et al. [7] (called `ASyncIteration`) to check for cases where a synchronized `Collection`'s `Iterator` is accessed from non-synchronized code. Figure 1 shows the three parts of a JavaMOP spec: lines 3–10 define *events* relevant at runtime, line 11 is the formal *property* to monitor over the events, and line 12 shows user-defined *handler* code that JavaMOP invokes when the monitored program reaches a certain state, i.e., when the spec is violated.

Each spec is parameterized by the types of objects whose instances may generate the events. Specifically, `CSC` is parameterized (line 1) by `Collection c` and `Iterator i`, which means that one monitor object will be created at runtime for every pair of related `c` and `i`. The `creation` keyword indicates that a monitor will be created after the `sync` event occurs (i.e., when one of the `synchronized*` methods on line 4 is invoked on a `Collection`). The monitor subsequently listens for the events `syncMk` (line 5), `asyncMk` (line 7), and `access`

```
1 Collections_SynchronizedCollection(Collection c, Iterator i) {
2  Collection c;
3  creation event sync after() returning(Collection c):
4   call(* Collections.synchronizedCollection(Collection)) || ... /* more calls */ ) { 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_MESSAGE); ... /* more printing */ }
13 }
```

Figure 1: Example spec, Collections_SynchronizedCollection (CSC), with its events and property

```
1    im = Collections.synchronizedList(...);
2  + synchronized(im) {
3    for (IInvokedMethod iim : im) {
4      ITestNGMethod tm = iim.getTestMethod();
5      ... }
6  + }
```

Figure 2: Buggy code in `TestNG`

```
Specification Collections_SynchronizedCollection has been violated on line
org.testng.reporters.SuiteHTMLReporter.generateMethodsChronologically
(SuiteHTMLReporter.java:365). 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.
```

Figure 3: A sample violation

(line 9). The `syncMk` events occur after `iterator()` is invoked on a `Collection` instance, `c`, to create an `Iterator`, `i`, and the thread did synchronize on `c` (lines 5–6). The `asyncMk` events occur after `iterator()` is invoked on `c`, but the thread did not synchronize on `c` (lines 7–8). Finally, the `access` events occur before any invocation of `Iterator` methods on `i` from any thread that did not synchronize on `c` (lines 9–10).

If the monitored program reaches a state where the extended regular expression (`ere`) property on line 11 is matched, then the handler code on line 12 is invoked. The `ere` matches when non-synchronized code creates an `Iterator` from a synchronized `Collection` (`sync asyncMk`) or when accessing a synchronized `Collection`'s `Iterator` from non-synchronized code (`sync syncMk access`). In our experiments, we used the default handler in JavaMOP: print a violation containing the spec name, the program line number where the spec violation occurred, a URL for the spec, and an explanation.

As an example, consider the buggy code in Figure 2, simplified from one of the six bugs we found in `TestNG`. The lines not starting with "+" (1 and 3–5) represent part of the original code that iterates over the synchronized `Collection` `im`. Note that the `for` loop is not synchronized, leading to a violation of the `CSC` spec. The violation that JavaMOP reports is shown in Figure 3; our inspection starting from this reported line of code led us to find the bug. The developers accepted our pull request that added the synchronization code, in the lines starting with "+" (2 and 6).

3. EXPERIMENTAL SETUP

We describe the open-source projects used in our study, the specs that were monitored while running tests in the projects, and how we automatically generated tests using Randoop [42,44,53]. We also explain our procedure for running JavaMOP and for inspecting resulting violations.

Table 1: Statistics of 200 projects used in our study

PID	Project	SHA	LOC	ManTests	AutoTests
P1	Altoros.VCSB	b6cfe23a	7290	1	–
P2	LogBlock.LogBlock-2	40548aad	875	1	–
P3	edanuff.CassandraCompositeType	6d09fcceb	1234	1	5427
P4	jriecken.gae-java-mini-profiler	80f3a5fe	908	8	92058
P5	mqtt	f4384253	11478	18	–
P6	plista.kornakapi	178061e3	3088	2	21594
P7	threerings.playn	c969166c	38388	139	–
P8	tbuktu.ntru	8126929e	7715	70	–
P9	OpenGamma.ElSql	db6c6d07	2581	160	11034
P10	sematext.ActionGenerator	10f4a3e8	1864	7	–
P11	vivin.GenericTree	15c59c99	677	49	7787
P12	hoverruan.weiboclient4j	6ca0c73f	3748	34	1229
P13	joda-time	cc35fb2e	85000	4157	124123
P14	IvanTrendafilov.Confucius	2c302878	1203	84	23196
P15	mikebrock.jboss-websockets	6d03a4ef	1736	1	6668
P16	b3log.b3log-latke	afb48c40	24399	76	–
P17	Thomas-S-B.visualee	410a80f0	3574	76	8164
P18	asterisk-java	b076175e	39498	220	33632
P19	Cue.lucene-interval-fields	805df6d	736	9	13102
P20	JSqlParser	001d665d	10517	341	14837
P21	Ovea.jetty-session-redis	afb2b25b	6358	7	15414
P22	bcel	24014e5e	35827	87	–
P23	zookeeper-utils	a2b80474	455	4	633
P24	bucchi.OAuth2.0ProviderFor.Java	db5e1d06	2654	47	–
P25	htrace	c32ec0b1	2521	11	–
P26	ptgoetz.storm-jms	d152d72f	1085	2	–
P27	UrbanCode.terraform	d67ac40c	12108	4	3089
P28	pignlproc	1a009980	2296	10	53693
P29	jmxtrans.embedded-jmxtrans	4f1ce2cc	5806	56	–
P30	apache.gora	bb09d891	24185	56	–
F69	69 projects with 100% FAR	various	349029	3834	561031
N101	101 projects without violations	various	520472	8484	1250330
		TOTAL	1214305	18065	2135081
		AVG	6071.52	90.33	17500.66
		MIN	24	1	1
		MAX	93260	4157	219404

3.1 Experimental Subjects

We selected the projects for our study from GitHub, starting from a list of the most popular Java projects. From these, we selected 200 projects that (i) used Maven (for ease of automation), (ii) had at least one test (so we can monitor test runs), (iii) had all tests pass without monitoring, and (iv) had all tests pass when monitoring with JavaMOP. Requirements (iii) and (iv) are important to have a fair measurement of runtime overhead of JavaMOP—if tests were to fail between the two runs, with and without monitoring, they may fail at different points in the execution, leading to rather different time measurements. Furthermore, tests could fail due to problems in the project or due to integration of JavaMOP. For example, we observed some failures of time-sensitive tests that have some timeouts resulting from the overhead of JavaMOP. We also observed test failures that happened because JavaMOP instrumentation interacted unexpectedly with some other instrumentation frameworks, e.g., test-mocking frameworks. We already reported some of these issues to the JavaMOP project [57].

Table 1 lists some basic statistics about the 200 projects used in our study. PID either starts with "P" to provide the

short ID of a project in which we found some real bug, or summarizes multiple projects with similar characteristics—"F69" summarizes 69 projects in which all inspected violations were false alarms, and "N101" summarizes 101 projects in which no violations were generated for the specs that we inspected. Project is the project name, SHA is the project revision we used, LOC is the number of Java lines in the project, ManTests is the number of manually written tests, and AutoTests is the number of automatically generated tests. "–" marks that we did not have Randoop tests, which happened for 49 projects with multiple Maven modules, 16 projects where generated tests did not compile, and 13 projects where Randoop did not generate any test within the time limit. For F69 and N101, ManTests and AutoTests show the sums for all respective projects. The rows TOTAL, AVG, MIN, and MAX are the sum, average, minimum, and maximum across all projects in each column.

Table 2: Mini-Survey. Ref: references; Subjects: kind of subjects; OSS: open-source projects; Sel-Classes: selected classes; #Sub: number of subjects; FAR[%]: false alarm rate reported; #Bugs: number of bugs found; Rep?: bugs reported to developers?

Ref	Subjects	#Sub	FAR[%]	#Bugs	Rep?
[46]	DaCapo+OSS	12	n/a	n/a	n/a
[54]	n/a	n/a	n/a	n/a	n/a
[28]	n/a	8	n/a	n/a	n/a
[39]	DaCapo	7	43.00	20	no
[13]	OSS+JDK	7	n/a	n/a	n/a
[63]	OSS	5	73.90	100	yes
[62]	OSS	8	n/a	1	no
[48]	DaCapo	10	0.00	54	no
[33, 34]	Sel-Classes	3	n/a	n/a	n/a
[60]	OSS	6	58.00	9	yes
[9]	OSS	4	n/a	n/a	n/a
[40]	OSS	3559	n/a	n/a	n/a
[17]	DaCapo	11	70.00	11	no
[3]	DaCapo	1	n/a	n/a	n/a
[29]	OSS	7	5.00	265	no
[49]	DaCapo	12	49.00	26	no
[58]	Sel-Classes	15	n/a	n/a	n/a

3.2 Specs Used in this Study

All Java API specs that we used in our study were obtained from the literature, 182 manually written specs [31, 35] and 17 automatically mined specs [49, 50]. We describe our rationale and procedure for selecting each set of specs.

3.2.1 *Manually Written Specs*

We used 182 manually written JavaMOP specs [32, 35], which are publicly available [51]. The specs were written by Lee et al. [31], who read Javadoc comments in four widely-used packages (`java.lang`, `java.net`, `java.io`, and `java.util`) and formalized sentences describing "must", "should" or "is better to" conditions. The specs are formalized using finite-state machines, extended regular expressions, linear temporal logic, and context-free grammars. JavaMOP can monitor specs in any formalism for which a suitable plugin exists.

To illustrate manual formalization of specs, consider again the CSC spec [12] from Section 2. It was formalized from text in `Collections.synchronizedCollection()` method's Javadoc: *"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"* [22]. Section 2 explained CSC in detail. As mentioned, this spec had been also used earlier [7]; by analyzing Javadoc comments, Lee et al. [31] ended up with some of the same specs that others had formalized before. Monitoring CSC in our experiments revealed bugs in several widely used projects, including `TestNG`, `ActiveMQ`, and `XStream`. However, our experiments also revealed issues and opportunities for improving the manually written specs, discussed in Section 5.2.

3.2.2 *Automatically Mined Specs*

To compare the effectiveness of manually written specs and automatically mined specs, we monitored 17 of the 223 specs automatically mined by Pradel et al. [45, 47, 49, 50]. Before settling on these specs, we performed a mini-survey of the spec mining literature to search for specs and to identify how spec mining was evaluated.

Paper Search: We searched for spec mining papers on DBLP [14] using this query: `specification|propert|contract|invariant|precondition mining|monitor|enforce|infer|mine venue:ICSE|venue:ASE|venue:RV|venue:PLDI|venue:POPL|venue:ISSTA|venue:ieee_trans_software_eng_tse_|venue:sigsoft_fse|venue:autom_softw_eng_ase_|venue:esec_sigsoft_fse|venue:tacas|venue:icsm|venue:icsme|venue:sas|venue:sac|venue:paste|venue:icfem|venue:issre|venue:compsac|venue:formats|venue:sttt|venue:ecoop|venue:fase|venue:oopsla_companion|venue:kdd|venue:vmcai|venue:seke|venue:cav|venue:oopsla|venue:electr_notes_theor_comput_sci_entcs_`

We obtained 163 potentially related papers, of which we considered only the 100 papers published in 2009–2015.

Paper Filtering: We split these 100 papers in half, and two of the authors read abstracts from each half independently to find papers that mined Java API specs that we could use. We omitted related papers, e.g., a survey [55], which did not report finding new specs. The result was 26 papers that we then read in more detail to answer these questions: (i) in what formalism are the mined specs (and can they be monitored with JavaMOP)? (ii) how many specs did they mine? (iii) did they find any bugs? (iv) do they report false alarms from evaluating the bug-finding effectiveness of the specs? (v) what is the reported false alarm rate, if any?

Email to Authors: After filtering, we settled on 17 papers and emailed authors who are not at our institution to ask for their mined specs. We received responses from authors of 7 papers, with 5 providing their specs. Of these 5, the specs from Pradel et al. [49] had the largest number that we could easily use—their specs were provided in the DOT format, which was straightforward to automatically translate to finite-state machines in the JavaMOP syntax.

Prior Evaluations: Table 2 lists the 17 papers. Although 7 papers report finding bugs while evaluating mined specs, only 2 report confirming the bugs with the developers. Further, evaluations were mostly performed on DaCapo, the benchmark initially curated to evaluate performance and not bug-finding effectiveness, and on a small number of open-source projects, with the exception of Nguyen et al. [40] who used thousands of projects but only to apply statistical techniques to mine specs and not to evaluate their bug-finding effectiveness. Finally, among the 7 papers that reported false alarm rates, the rates varied widely, from 0.0% to 73.9%. Our experiments are therefore complementary to those in the papers we surveyed. In fact, we find even higher false alarms rates. Our experiments also revealed issues in the automatically mined specs, as discussed in Section 5.

3.3 Runtime Verification with JavaMOP

Using JavaMOP to monitor test runs is quite simple: integrate JavaMOP in the project and invoke `mvn test`. JavaMOP integration in Maven-based projects is described online [20]. First, the JavaMOP compiler generates a Java agent [41] from the specs to be monitored, enabling dynamic instrumentation of code running in the Java Virtual Machine. Next, the build configuration file, `pom.xml`, is modified to make the Maven Surefire plugin (which runs the tests) aware of the JavaMOP agent. Subsequent invocations of `mvn test` attach the JavaMOP agent to the test-running process for monitoring the runs against all the specs simultaneously. We automated JavaMOP agent creation, changing `pom.xml`, monitoring each project, and post-processing results.

In each set of experiments, we ran the tests in each project twice. First, we ran without JavaMOP to measure the base test-running time and check that the tests pass by themselves. We then integrated JavaMOP and reran the tests to measure test-running time with monitoring and to record violations. We configured JavaMOP to log all output to a file. We excluded from monitoring standard Java libraries (that are less likely to have bugs) and some third-party libraries, such as Maven Surefire (to reduce overhead) and test-mocking frameworks (which we found to have unexpected interactions with JavaMOP, as mentioned in Section 3.1). All JavaMOP experiments were run on a 64-bit computer with Intel® Core™ i7-3770K CPU @ 3.50GHz processor and 32GB of RAM running Ubuntu 14.04.4 LTS and Java 7 or 8 (as required by the project).

3.4 Automatically Generating Tests

To evaluate whether the type of tests impacts the bug-finding effectiveness of the specs, we used Randoop [42, 44, 53] to automatically generate additional tests. We generated tests on a Core™ i7-4700MQ 2.40GHz Quad-Core processor PC with 8GB of RAM, Ubuntu 15.04, Java 7 or 8, and Randoop heap usage limited to 4GB. We ran Randoop on all 151 single-module Maven projects (out of total 200), which were easier to automate than multi-module Maven projects. We limited test-generation time to 1min and 5min. We had a separate run to monitor the generated tests (using JavaMOP) against the same set of manually written specs. The number of *new* violations, i.e., those which were not already reported while monitoring manually written tests, showed little difference between the tests automatically generated in 1min and 5min. Therefore, we decided to use the tests generated in 5min and did not increase the time limit further. Other researchers who used Randoop also found tests generated in different intervals to behave similarly [43, 56, 59].

3.5 Inspecting Violations

We describe our procedure for selecting and inspecting violations that JavaMOP reported while monitoring test runs. We refer to the source-code line number at which JavaMOP reports a spec violation as the violation *site*. JavaMOP reports a violation every time a spec is violated at runtime, so it can report many violations of the same spec at the same site (e.g., if the site is in a loop or invoked from multiple tests). We refer to all violations that are reported by JavaMOP during test execution as *dynamic violations (DV)* and we refer to unique violations—those that happen in the same project, for the same spec, and at the same site—as *static violations (SV)*.

We inspected some static violations from both manually written and automatically generated specs. For manually written specs, we inspected all violations from 42 specs and ignored all violations from 21 specs. For automatically mined specs, we sampled to inspect 200 out of 1141 violations of the 17 automatically mined specs that we monitored. To sample 200 violations, we used stratified sampling [11]: we divided all violations into strata based on the spec, and from each stratum randomly selected a number of violations, in proportion to the ratio of the stratum's size to the total number of violations. We excluded 21 manually written specs from inspection and did not monitor 206 automatically mined specs because of issues with these specs, discussed in Section 5.2.

Our inspection goal was to find as many bugs as possible while increasing the chance that the developers accept the resulting pull requests. Therefore, multiple authors inspected most violations. For manually written tests and manually written specs, first, two reviewers independently inspected each violation and classified it as one of:

TrueBug: A potential bug to be confirmed by reporting to the developers or by checking if it was already fixed;

FalseAlarm: The violation does not indicate a bug in the code but effectively a bug/imprecision in the spec; or

HardToInspect: The violation is hard to classify as a TrueBug or a FalseAlarm, because source code is missing or is particularly hard to reason about.

Next, the independent reviewers met to discuss and agree on the classifications they had independently assigned and to resolve cases in which one reviewer had classified a violation as a TrueBug but the other had given another classification. Cases where they still could not agree were classified as TrueBug if any one of the reviewers had classified as a TrueBug. A third reviewer then met with the two initial reviewers to confirm all violations that were classified as TrueBugs. For automatically mined specs, we followed a similar procedure: two reviewers inspected each violation reported from monitoring automatically mined specs while running manually written tests. For automatically generated tests, only one reviewer inspected each violation because we had built enough experience from inspecting the violations from manually written tests.

For each violation that we classified as a TrueBug, we submitted a bug report and/or a fix (pull request) to the developers of the respective project to check whether they agree that a code change can be beneficial. As discussed in Section 1, inspecting violations and submitting pull requests to developers is challenging. For inspections alone, each of the two initial reviewers spent between 4min and 54min per violation. Summing up all the time to meet for resolving disagreements, to prepare pull requests, to iterate over them internally, to communicate with developers, and to record and process the status of each pull request, we estimate that it took over 1,200 hours just for this process of inspecting and creating pull requests.

We carefully prepared pull requests, trying to obtain an "upper bound" on the effectiveness of the specs. That is, some violations that we classified as TrueBugs may have been ignored by developers running a tool on their own or in the absence of our pull requests. We did not simply submit bug reports indicating the violation of a spec in a codebase; we were concerned that developers may not understand the spec or care to change the code. Instead, we submitted pull requests that included a proposed code change.

Table 3: Dynamic (DV) and static (SV) violations, and overhead for 42 manually written Specs. ManTests: manually written tests; AutoTests: automatically generated tests; PID, TOTAL, AVG, MIN, MAX, "–": same as in Table 1

PID	ManTests			AutoTests	
	DV	SV	Overhead[%]	DV	SV
P1	13	4	187.93	–	–
P2	1	1	50.96	–	–
P3	20	2	110.72	0	0
P4	0	0	157.72	20	1
P5	412	2	-28.37	–	–
P6	24	2	155.36	0	0
P7	1	1	201.39	–	–
P8	27	7	27.88	–	–
P9	384	9	239.98	0	0
P10	961	3	128.17	–	–
P11	26	5	128.86	7	1
P12	0	0	248.97	558	16
P13	236	95	245.26	0	0
P14	74	1	123.09	75242	9
P15	0	0	2.30	287	3
P16	167	13	72.25	–	–
P17	37	13	126.38	18	1
P18	2	1	104.53	6717	6
P19	746	5	284.76	12520	3
P20	27977	1	105.98	1493	3
P21	21	4	324.51	7241	4
P22	181430	4	338.72	–	–
P23	1038	16	67.46	0	0
P24	88	5	228.58	–	–
P25	31	10	69.88	–	–
P26	7	5	51.50	–	–
P27	0	0	84.23	1322	8
P28	414	13	14.57	0	0
P29	29	13	4.30	–	–
P30	467	29	616.59	–	–
F69	85120	269	13297.49	96111	64
N101	0	0	8106.04	0	0
TOTAL	209753	533	25877.95	201536	119
AVG	1498.77	2.67	129.39	1651.93	0.98
MIN	0	0	-28.37	0	0
MAX	181430	95	1036.57	75242	16

multaneously monitoring all specs is under 4.3× on average. We believe this runtime overhead is acceptable during development time (not production time), considering the number of bugs we found and the fact that the tests in these projects run relatively fast—the average additional time incurred by JavaMOP was 4.08s for 42 specs and 12.48s for 182 specs. The relatively small average overhead reflects the tremendous progress made in the research community over the last decade to make runtime verification more efficient.

Table 3 also shows the number of dynamic (DV) and static (SV) violations from monitoring 42 manually written specs on both manually written tests (ManTests) for all 200 projects and automatically generated tests (AutoTests) for 122 projects (of the 200 projects, 151 were single-module Maven projects, but the tests generated by Randoop did not compile in 16 projects, and Randoop did not generate any test in 5min for 13 projects). Even when DV is relatively high, the overhead remains reasonable.

4. RESULTS

We aim to evaluate the effectiveness of existing specs for finding bugs when monitoring tests in open-source projects. We investigated the following research questions (RQs):

RQ1 What is the runtime overhead of monitoring?

RQ2 How many bugs are found from violations?

RQ3 What are the false alarm rates among violations?

4.1 RQ1: Runtime Overhead of Monitoring

Table 3 shows the runtime overhead (Overhead[%]) from monitoring 42 manually written specs. We measured overhead only for manually written (and not automatically generated) tests, because they pass in all 200 projects (while some automatically generated tests fail, making it hard to reliably measure overhead). Runtime overhead is computed as $(mop - base)/base * 100\%$, where *mop* is the time to run tests *with* monitoring, and *base* is time to run the tests *without* monitoring. As in previous JavaMOP studies, we observed some negative runtime overheads, e.g., in P5. These can be due to noise in the time measurements or due to the instrumentation changing the garbage-collection behavior of the program, causing it to run faster [23, 25, 36].

The average runtime overhead was 129.39% when monitoring only the 42 inspected specs (as shown in the table) and 330.14% when monitoring all 182 manually written specs (elided for lack of space). Therefore, the overhead of si-

NEXT page

4.2 RQ2: Bugs Found

We found a total of 114 SV that were TrueBugs, 110 for manually written specs and 4 for automatically mined specs. Recall that we map dynamic to static violations based on the project being monitored, the spec being violated, and the violation site. When multiple projects use the same library (even if not the exact same version), then multiple static violations can actually map to the same bug. Our 114 TrueBugs map to 97 unique bugs. Because most projects evolved since we started our experiments (with then latest revisions of the projects), 2 of the unique bugs we found were already fixed in the current latest revisions. For the remaining 95 bugs, we submitted pull requests, with 74 already accepted and only 3 rejected; the remaining 18 are still pending.

4.3 RQ3: False Alarm Rates

A key metric to evaluate the effectiveness of specs is the false alarm rate (FAR), i.e., the ratio $FA/(TB+FA)$, where FA and TB are the number of FalseAlarms and TrueBugs among inspected violations. For manually written specs, we inspected 652 violations—533 from manually written tests and 119 from automatically generated tests. Table 4 shows, for each project in which we inspected violations, the project ID (PID), number of inspected static violations (SV), number of violations in each classification (HTI, TB, and FA), and false alarm rate (FAR[%]). All 69 projects in F69 have 100% FAR (no TrueBugs) and had slightly more violations than all those with TrueBugs. 19 of 30 projects with some TrueBug had greater than 50% FAR. The TOTAL row shows the overall FAR: for manually written specs, it is 82.81% (110 TrueBugs and 530 FalseAlarms). For automatically mined specs, we inspected 200 violations. We elide the breakdown per project, but the overall FAR for automatically mined specs is 97.89% (4 TrueBugs and 186 FalseAlarms).

We further analyzed FAR along several dimensions, trying to identify where it may be lower. Table 5 (top part) shows the FAR breakdown for manually written specs. Violations in third-party libraries had 86.55% FAR, while violations in the project code had 80.82% FAR. Violations in single- vs. multi-module Maven projects had 81.87% vs. 86.23% FAR, and violations for manually written tests vs. automatically generated tests had 82.51% vs. 84.21% FAR. The similar FARs across all these dimensions suggests that the FARs are mostly due to inherent (in)effectiveness of the specs and less

Table 4: Per-project inspection summary for 42 manually written specs. SV: static violations; HTI: hard to inspect; TB: true bugs; FA: false alarms; FAR[%]: false alarm rate

PID	SV	HTI	TB	FA	FAR[%]
P1	4	0	4	0	0.00
P2	1	0	1	0	0.00
P3	2	0	2	0	0.00
P4	1	0	1	0	0.00
P5	2	0	2	0	0.00
P6	2	0	2	0	0.00
P7	1	0	1	0	0.00
P8	7	0	6	1	14.29
P9	9	0	6	3	33.33
P10	3	0	2	1	33.33
P11	6	0	3	3	50.00
P12	16	0	7	9	56.25
P13	95	0	40	55	57.89
P14	10	0	4	6	60.00
P15	3	0	1	2	66.67
P16	13	0	4	9	69.23
P17	14	0	4	10	71.43
P18	7	0	2	5	71.43
P19	8	0	2	6	75.00
P20	4	0	1	3	75.00
P21	8	0	2	6	75.00
P22	4	0	1	3	75.00
P23	16	0	4	12	75.00
P24	5	0	1	4	80.00
P25	10	0	2	8	80.00
P26	5	0	1	4	80.00
P27	8	0	1	7	87.50
P28	13	1	1	11	91.67
P29	13	0	1	12	92.31
P30	29	1	1	27	96.43
F69	333	10	0	323	100.00
TOTAL	652	12	110	530	82.81

Table 5: Split of inspection results along various dimensions. Column headers are same as in Table 4

Type of specs	SV	HTI	TB	FA	FAR[%]
Manually written	652	12	110	530	82.81
Libraries	232	9	30	193	86.55
Project code	420	3	80	337	80.82
Single-module	513	11	91	411	81.87
Multi-module	139	1	19	119	86.23
ManTests	533	7	92	434	82.51
AutoTests	119	5	18	96	84.21
Automatically mined	200	10	4	186	97.89
Libraries	122	10	0	112	100.00
Project code	78	0	4	74	94.87
Single-module	148	9	3	136	97.84
Multi-module	52	1	1	50	98.04

Table 6: Per-spec inspection summary. Column headers are same as in Table 4

Spec	SV	HTI	TB	FA	FAR[%]
URLDecoder_DecodeUTF8	1	0	1	0	0.00
Collections_SynchronizedColl...	22	0	19	3	13.64
Collections_SynchronizedMap	5	0	4	1	20.00
Byte_BadParsingArgs	3	0	2	1	33.33
Long_BadParsingArgs	22	0	14	8	36.36
InetSocketAddress_Port	2	0	1	1	50.00
ByteArrayOutputStream_Flu...	123	0	55	68	55.28
StringTokenizer_HasMoreEle...	11	0	4	7	63.64
Math_ContendedRandom	14	0	5	9	64.29
Short_BadParsingArgs	3	0	1	2	66.67
Iterator_HasNext	157	3	4	150	97.40
31 Specs with 100% FAR	289	9	0	280	100.00
TOTAL	652	12	110	530	82.81

due to specific code-related factors. An interesting finding is that violations in libraries are somewhat more likely to be false alarms, as one would expect that libraries are indeed better tested and have fewer bugs than the project code.

Table 5 (bottom part) shows the breakdown for automatically mined specs. Compared to manually written specs, the FAR values are higher along all dimensions. The overall FAR was 97.89% (186 of 190 non-HTI violations). Compared within different dimensions, the FAR values were similar, e.g., 100.00% for violations in libraries vs. somewhat lower 94.87% for violations in the project code, showing a consistent relationship with violations of manually written specs. In brief, all these FARs appear rather high.

Table 6 shows the FAR values for the 42 manually written specs that we inspected (we did not inspect violations of 21 specs, as explained in Section 5.2.1). First, only 11 specs (i.e., 26.19% of 42 inspected specs and 6.04% of all 182 specs) helped find a TrueBug and could have provided some value to developers of some project(s). Second, 119 specs were never violated, so they only increased the runtime overhead. These specs may get violated if monitored on other projects. Third, all but one of the specs that we inspected caused at least one FalseAlarm, and the only spec without false alarms, `URLDecoder_DecodeUTF8`, was violated only once. Interestingly, the spec that was violated the most and is the least effective at bug finding, `Iterator_HasNext` with 97.40% FAR, is the *de facto* example spec in research papers on spec mining and runtime verification. Section 5.2.2 discusses why this spec and others generate so many FalseAlarms.

For automatically mined specs, only 3 (i.e., 17.65% of the 17) led to at least one TrueBug in the 200 inspected violations—`FSM162`, `FSM33`, and `FSM373`[1], with FARs of 87.50%, 90.00% and 98.06%, respectively. `FSM373` is very similar to the manually written `Iterator_HasNext` spec and has similar FAR as well. Based on the very high FARs among violations of both manually written and automatically mined specs, we conclude that the existing specs are rather ineffective for finding bugs, because they raise too many false alarms.

5. ANALYSIS OF RESULTS

We discuss some bugs we found, some issues with the specs (and opportunities to improve them), and some developers' responses to our pull requests (bug reports and fixes).

5.1 Analysis of Bugs Found

We describe some of our pull requests that the developers accepted and all three pull requests that the developers rejected so far.

5.1.1 *Accepted Pull Requests*

The project with the largest number of accepted pull requests in our study was `joda-time`, *"the* de facto *standard date and time library for Java prior to Java SE 8"* [26]. The `joda-time` developers accepted all our 40 pull requests, 37 of which based on the violations of the manually written spec `ByteArrayOutputStream_FlushBeforeRetrieve (BAOS)`. `BAOS` catches cases where an underlying `ByteArrayOutputStream` is not closed or flushed before retrieving the contents of the enclosing stream. The fix in our pull requests was

[1]These specs are publicly available [50].

simply to invoke `flush()` before `toByteArray()`, `toString()`, or `write*()` on a `ByteArrayOutputStream`. In all projects, 49 out of 55 `BAOS` pull requests that we submitted were accepted, 1 was rejected, and the others are pending.

Another big set of bugs was found from the violations of `CSC` (discussed in sections 2 and 3.2.1) and a closely related spec, `Collections_SynchronizedMap`. These specs are violated if the `Iterator` of a synchronized `Collection` is accessed from code that is not synchronized. Our fix was to put the calling code in a synchronized block. Our pull requests for these specs were mostly accepted, or were already fixed between the start of our experiments and when we wanted to report them in widely used applications—`Spring-Beans`, `TestNG`, and `XStream`. We also have a pending pull request in `ActiveMQ`.

All the 18 bugs that we found while monitoring automatically generated tests were related to missing checks for invalid input. 17 were of the form `Type_BadParsingArgs`, where `Type` is `Long`, `Short`, or `Byte`. These specs check that calls to the respective `Type.parseType(String s, int r)` methods do not have `s` empty or `null`. 12 pull requests were accepted, 1 has been rejected, and 4 are pending. The remaining (and still pending) invalid-input-related pull request was for a violation of `InetSocketAddress_Port` spec which checks that the `int` port number used to create new `java.net.InetSocketAddress` objects is between 0 and 65535, inclusive.

Finally, we found 4 bugs from monitoring the specs that Pradel et al. [49] mined automatically. Of these, 3 were duplicates of bugs found from monitoring manually written specs, so we did not report them again. The additional bug (with a pending pull request) was from a violation of `FSM33`, where `removeFirst()` was invoked on a `java.util.LinkedList` object without first checking that it was not empty.

5.1.2 Rejected Pull Requests

Three of our pull requests were rejected, mostly because we had limited domain knowledge. In `XStream`, we submitted a pull request for a `Collections_SynchronizedMap` violation, but the developer rejected it and responded: *"...there's no need to synchronize it... As explicitly stated in the documentation, XStream is not thread-safe during setup... this is documented behavior."* In `JSqlParser`, we reported a missing check for the validity of `s` in `Long.parseLong(String s, int i)`, and the developer responded: *"...The parser itself ensures that only long values are passed to LongValue. So do you have a problematic SQL, that produces a NumberFormatException?"* Indeed, the violation was from monitoring an automatically generated test, but since the violation is in a public class, it could lead to unhandled exceptions in applications that depend on `JSqlParser` but which do not thoroughly sanitize their own input `SQL` queries; we plan to revisit this in the future. In `threerings.playn`, we submitted a fix for a `BAOS` violation, and the developer responded: *"JsonAppendableWriter automatically flushes the target stream when `done()` is called, as is documented in the Javadoc for done. So an additional flush is unnecessary."* Indeed, `BAOS` did not detect the flush because the spec is buggy. The violation occurred in a method which casts a `java.io.OutputStream` to `java.io.Flushable` before invoking `flush()`. However, `BAOS` was written to only track calls of `flush()` on `java.io.OutputStream` and its subtypes, whereas `Flushable` is a supertype of `OutputStream`. JavaMOP, therefore, correctly finds a violation of the spec, but the spec is incorrect. We submitted a bug report for `BAOS` to the JavaMOP repository and confirmed that it did not affect any other `BAOS`-related pull request that we sent.

5.2 Issues with Monitored Specs

We next discuss why we did not monitor some specs or inspect some violations, and give examples to show why the specs reported a lot of false alarms.

5.2.1 Ignored Specs

Manually Written Specs: We inspected all (652) static violations (SV) from 42 manually written specs. 21 other manually written specs had violations, but we did not inspect them: (i) 8 `*StaticFactory` specs may, at best, find performance bugs not functional bugs (459 SV); (ii) 2 `*_Obsolete` specs get violated for every call to `Dictionary()` or `Enumeration()`, and were written as "suggestion" specs that should not lead to bugs (518 SV); (iii) 4 `*_StandardConstructors` specs were marked as potentially reporting false alarms (430 SV); (iv) 2 `Enum_*` specs were buggy and get violated on every invocation of `Enum` methods (874 SV); (v) 1 `Serializable_UID` spec gets violated when a `Serializable` class does not declare a `serialVersionUID`, which can be trivially checked statically (2348 SV); and (vi) 4 more specs were ignored because they did not report violation sites (93 SV). We reported 16 of these spec issues, together with 7 bugs that we found in other specs during our inspections—a total of 23 bug reports—to the JavaMOP repository, and the process of improving the specs is ongoing.

Automatically Mined Specs: Although we originally obtained 223 mined specs from Pradel et al., we monitored only 17, because a brief manual inspection of specs found that 206 had one or more of the following issues: (i) the spec (FSM) was very large, sometimes having tens of transitions and/or states, making it hard to understand and to inspect its violations; (ii) the spec relates only methods in the `javax.swing.*` or `java.awt.*` libraries; (iii) the spec imposes unnecessary temporal order on methods of multiple unrelated object types; and (iv) the spec imposes unnecessary temporal order on unrelated methods of the same object type. We did not report or attempt to improve the automatically mined specs. In fact, Pradel et al. [49] acknowledge that some of these specs are of low quality and develop a system that to prune some violations of mined specs. However, it would be better to additionally evaluate the spec mining techniques on larger, more diverse projects and confirm detected (potential) bugs with developers.

5.2.2 Analysis of False Alarms

The monitored specs reported many false alarms mainly because the specs (i) did not encode all correctness conditions, or encoded wrong conditions, and thus need to be improved; or (ii) captured harmless misuse of APIs which would rarely or never lead to actual bugs.

For example, consider the `Iterator_HasNext` spec which states that each invocation of `next()` on a `java.util.Iterator` object must be preceded by an invocation of `hasNext()` that returns `true` on the same object. `Iterator_HasNext` violations led us to discover 4 accepted bugs in the `Thomas-S-B.visualee` project, and other researchers had previously used `Iterator_HasNext` to find some real bugs in `AspectJ` (bug IDs #218167 and #218171 [60]). However, `Iterator_HasNext` also reports a huge number of false alarms—150 of 154 non-HardToInspect violations were false alarms—with FAR of

```
1 ArrayList<Integer> list = new ArrayList<>();
2 list.add(1);
3 Iterator<Integer> it = list.iterator();
4 if (it.hasNext()){ int a = it.next();}
5 if (list.size() > 0){
6  int b = list.iterator().next();
7 }
8 if (!list.isEmpty()){
9  int c = list.iterator().next();
10 }
11 HashMap<String, Integer> map = new HashMap<>();
12 map.put("one", 1);
13 if (map.containsKey("one")){
14  int d = map.values().iterator().next();
15 }
16 int e = list.iterator().next();
17 int f = map.values().iterator().next();
```

Figure 4: False alarms from `Iterator_HasNext` spec

```
1 Map<String, String> map = new HashMap<>();
2 map.put("1", "1"); map.put("2", "2");
3 for(String key : map.keySet()) {
4  String value = map.get(key);
5  map.put(key, value + "x");
6  //map.put(key + "x", value + "x");
7 }
```

Figure 5: False alarms from `Map_UnsafeIterator` spec

97.40%. Figure 4 illustrates several valid invocations of `Iterator.next()`—lines 4, 6, 9, 14, 16, and 17—with no bugs in the shown code. However, `Iterator_HasNext` will be violated for all those invocations except the one on line 4. The example `next()` invocations in Figure 4 illustrate only a few of the valid uses of the `next()` method that were violations of the `Iterator_HasNext` spec during our experiments. To make the `Iterator_HasNext` spec more precise, one would need to ensure that it encodes more valid ways of checking that an `Iterator` has enough elements before invoking `next()`, taking into consideration various possible `Collection` types.

`Map_UnsafeIterator` is another spec with false alarms; it checks whether code is modifying a `java.util.Map` instance while iterating over it. All 9 `Map_UnsafeIterator` violations that we inspected were false alarms. To illustrate, consider the code snippet in Figure 5. On each iteration, line 5 modifies the values in the `Map`—a valid operation. Nevertheless, `Map_UnsafeIterator` is violated, because it is too restrictive, and reports a violation for any modification to the `Map`. If line 5 is replaced with the commented-out statement on line 6, the standard Java library would throw a `ConcurrentModificationException`. We therefore asked other JavaMOP developers (not involved in this project) why anyone would want to monitor this spec. The response reflects one challenge in coming up with effective specs: *"Invoking the put method on a map object may or may not change its key set... there is a trade-off between accuracy and simplicity... it is up to the user; one can put more effort into writing more fine-grained specs... so that there will be fewer false alarms reported; or write a simple spec easily and [then] manually eliminate the false alarms."*

Roughly 20% of all the false alarms among manually written specs were from two `Closeable_*` specs, with 113 violations between them. Both had 100% FAR. One of them catches calls of `close()` on subtypes of `java.io.OutputStream` for which `close()` is a no op. The other catches situations where calling `close()` on an `OutputStream` object that is already closed has no effect. Although both of these specs can help find developers' likely misunderstanding of the API, we classified them as FalseAlarms because they are harmless in the current version of the code. It is debatable whether we should have classified these as "code smells" as done in some prior work [17,39,49], and whether these were serious enough to submit to the developers. We could not easily change the code to avoid these problems, and it is highly unlikely that the developers would have accepted our changes.

Automatically mined specs have similar reasons for false alarms as manually written specs. For example, `FSM373` is similar to `Iterator_HasNext`, so its false alarms were similar as well. However, one additional cause of false alarms among violations of `FSM373` was that it did not permit to call `hasNext()` multiple times successively (a self transition is missing from a state in the FSM). `FSM162` also contains transitions that are similar to `Iterator_HasNext`, but also adds in a single transition on the `Iterator.remove()` method such that the spec is violated if `remove()` is called multiple times successively.

5.3 Developers' Responses

We discuss some example responses and comments that developers made regarding our pull requests, which gives a valuable insight into developers' perception.

Developers Asked for More: After we submitted a pull request for a `BAOS` violation, the `apache.gora` developers asked us to help check other portions of their code: *"...Are there any other instances of this behavior throughout the codebase? ...I just undertook a quick scan of the codebase for ByteArrayOutputStream, I found the following instances. Can you please check these out as well?"* Even after we fixed these other instances that they pointed out, the developers asked whether we would be interested to help with similar problems in their other codebase. In another project, `hoverruan.weiboclient4j`, we sent a pull request that fixed one of seven `Long_BadParsingArgs` violations and simply reported the other six. The developers fixed the remaining six within a day of accepting our pull request.

Developers Viewed Pull Requests Liberally: The `joda-time` developers accepted one of our `BAOS` pull requests although they found it unnecessary: *"While I'm not convinced it is necessary, this will cause no harm."* We got similar comments for two pending pull requests. In `Apache Zookeeper`, for the `BAOS` spec, the developer wrote *"Makes sense. I don't see why we shouldn't do what you suggest (add the flush). You see why it's a no-op currently though, right? (and why we haven't seen issues with this code)"*. In `TestNG`, the developer tagged one of our synchronization-related pull requests as a `perf/enhancement` and said, *"I'm not sure if it is relevant here: the lists of results should be already computed...no one is supposed to add something new at the report phase."*

Developers Accepted Better Exception Messages: For pull requests pertaining to missing checks for invalid inputs, developers responded well to the better error messages that we provided. In `IvanTrendafilov.Confucius`, the developer responded *"Looks good, I'll be happy to add that more helpful error message to the lib. Yes, please also add this check for `parseShort` and `parseByte`..."*. Similarly, in `jriecken.gae-java-mini-profiler`, the developer commented on our suggested error message *"Not sure that this is much better than the previous behavior - the exception message is a little more helpful, but it still throws a NumberFormatException"*, and

requested that we further modify our pull request before they accepted it.

6. SUGGESTIONS FOR THE FUTURE

Based on the experience from this study, we give several suggestions to help the spec mining and runtime verification research communities with *spec engineering*, i.e., writing/discovering and evaluating more effective specs.

(1) **Increased Focus on Bug-Finding Effectiveness:** More focus should be on the bug-finding effectiveness of specs, which is more important to developers than the performance of monitoring. For example, the most widely used `Iterator_HasNext` spec was highly ineffective for finding bugs.

(2) **Better Spec Categorization:** It is crucial to find good ways to designate the severity levels of specs. All specs are not equal in their bug-finding effectiveness. Some specs, when violated, indicate a bug with a very high probability. Other specs indicate issues that may be bugs in some projects but not in others. Finally, some specs are less severe, indicating potentially poor coding practices and may not lead to the detection of actual bugs.

(3) **Complementing Benchmarks:** Continued use of benchmarks like DaCapo is good for comparison with older results and evaluating performance of new techniques, but benchmarks should be complemented with evaluations on a larger number of open-source projects, to assess the techniques and specs in more realistic scenarios.

(4) **Confirming Detected Bugs with Developers:** Evaluating on recent project versions and reporting detected bugs to developers of open-source projects should be encouraged more. Admittedly, the process is challenging and time consuming, requiring to understand the application domain and communicate with the developers. We have publicly released a list of all our pull requests, to serve as a starting point for collecting true bugs: [57]. We found interesting results from submitted pull requests, e.g., even "buggy" specs like `BAOS` can lead to accepted pull requests

(5) **Automated Filtering of Specs and False Alarms:** It is necessary to better automatically filter out likely false alarms to improve ineffective specs. We found that specs with too many violations were almost always ineffective. Pradel et al. [49] defined some heuristics-based automated techniques for filtering out violations while statically checking mined specs, while Gabel and Su [18] as well as Nguyen and Khoo [39] proposed techniques for checking that mined specs are indeed true specs. Much more work in this direction is needed, especially because manual inspection, which we did in this paper, is rather tedious.

(6) **Open Spec Repositories:** It would be beneficial to have community-driven spec repositories and standardized ways of representing specs to facilitate spec sharing—we could have evaluated more specs if it were easier to find and use them. We started such a repository using all the specs monitored in this paper [51], we plan to continue adding more specs to this repository, and invite the research community to contribute their specs there as well to facilitate research on engineering better specs.

7. THREATS TO VALIDITY

External: The results of our study may not generalize beyond the projects, tests, or specs that we evaluated. To mitigate this threat, we used a larger number of open-source projects than had been evaluated in previous runtime verification and spec mining studies. Further, the 200 projects that we used were quite diverse in size, number of tests, and GitHub activity. Concerning the bug-finding effectiveness of specs, we used the largest sets of manually written and automatically mined specs that we could find with our mini-survey of the spec mining and runtime verification literature, and that could easily work with JavaMOP. JavaMOP is representative of the performance of runtime verification tools and allows to simultaneously monitor specs written in different formalisms, making it well suited for our large-scale evaluation of existing specs. Our study is focused on Java, and the results may differ for other programming languages.

Internal: We wrote scripts to automate the monitoring of tests against the specs. Our scripts that run the tests, measure overhead, and post-process results were reviewed by at least two authors. During inspection and classification of violations into TrueBug, FalseAlarm, and HardToInspect, we initially had two reviewers inspect independently to prevent them from influencing each other. It is possible that some violations we labeled as FalseAlarms are actually TrueBugs. For violations that we labeled as TrueBugs, we submitted 95 pull requests, and developers make the final judgment whether to accept (74 so far) or reject (3 so far).

8. CONCLUSIONS

Runtime verification has been receiving increased attention in the research community, with substantial contributions to reducing the overhead of monitoring. However, insufficient work has been done on evaluating and improving specs. Our study shows that existing tools such as JavaMOP have an acceptable overhead for development-time monitoring of test runs, and the existing specs can find some true bugs. Unfortunately, a vast majority of violations from these specs are false alarms. We believe that this greatly hinders the adoption of these techniques by practitioners.

Based on the experience from our study, we provided a set of recommendations for future work. We also made publicly available the data from our study [57] to aid future research. The runtime verification and spec mining research communities need to put much more emphasis on better *spec engineering* to develop more effective specs. It is possible that improving existing specs or mining new effective specs will need more expressive formalisms, which may slow down monitoring and require further efficiency improvements to runtime verification. But only when effective specs are available, will it be truly worthwhile to consider how to further make monitoring faster and to have some chance of practical adoption. We hope this paper presents a call to action for the researchers to develop better specs and evaluate them more thoroughly.

9. ACKNOWLEDGMENTS

We thank Alex Gyori, Farah Hariri, Cosmin Radoi, and August Shi for feedback on early drafts of this paper, Rahul Gopinath for discussions and help with Randoop, and He Xiao and Yi Zhang for help with JavaMOP. We also thank all authors of papers who replied to our emails concerning their mined specs. This research was partially supported by the NSF Grants CCF-1421503, CCF-1421575, CCF-1438982, and CCF-1439957. Wajih Ul Hassan was partially supported by the Sohaib and Sara Abassi Fellowship.