Noname manuscript No.
(will be inserted by the editor)

How Effective are Existing Java API Specifications for Finding Bugs during Runtime Verification?

Owolabi Legunsen[1] · Nader Al Awar[2] · Xinyue Xu[1] · Wajih Ul Hassan[1] · Grigore Roşu[1] · Darko Marinov[1]

Received: date / Accepted: date

Abstract 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*.

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 test methods 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 or accepted 76. However, most violations, 82.81% of 652 and 97.89% of 200, were false alarms.

Based on our empirical results, we conclude 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

Owolabi Legunsen
legunse2@illinois.edu

Nader Al Awar
nma85@mail.aub.edu

Xinyue Xu
silviaxxy@gmail.com

Wajih Ul Hassan
whassan3@illinois.edu

Grigore Roşu
grosu@illinois.edu

Darko Marinov
marinov@illinois.edu

[1]University of Illinois at Urbana-Champaign, Urbana, IL, USA
[2]American University of Beirut, Lebanon

alarm rates are worrisome and suggest that substantial effort needs to be spent on engineering better specs and properly evaluating their effectiveness. We repeated our experiments on a different set of 18 projects and inspected all resulting 742 violations. The results are similar, and our conclusions are the same.

Keywords runtime verification, monitoring-oriented programming, specification quality, software testing, empirical study

1 Introduction

In runtime verification, the execution of a software system is dynamically checked against formal specifications (specs for short) (Bodden et al., 2007a, 2008; Chen and Roşu, 2003; Dwyer et al., 2010; Hussein et al., 2012; Luo et al., 2014; Meredith and Roşu, 2013; Meredith et al., 2008). 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. (Robillard et al., 2013) 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.

The potential for using runtime verification during software testing was previously recognized (Jin et al., 2012a; Karaorman and Freeman, 2004; Lee et al., 2012; Luo et al., 2014), 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 (Bodden et al., 2008; Forejt et al., 2012; Jin et al., 2011; Luo et al., 2014; Navabpour et al., 2011; Wu et al., 2013), (ii) being able to monitor many specs simultaneously (Arnold et al., 2008; Purandare et al., 2013), and (iii) better-engineered runtime verification tools (Bodden, 2011; Jin et al., 2012a). We recently proposed to combine runtime verification with regression testing, where test executions are monitored against formal specs of API usage to find bugs during software evolution (Legunsen et al., 2015).

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 runtime verification algorithms, techniques, and tools. The specs used in previous research were manually written (Allan et al., 2005; Bodden et al., 2007a; Jin et al., 2011; Luo et al., 2014) or automatically mined (Beckman and Nori, 2011; Chen et al., 2015; Dallmeier et al., 2010; Gabel and Su, 2010; Krka et al., 2014; Lee et al., 2011; Lemieux, 2015; Lemieux et al., 2015; Nguyen and Khoo, 2011; Nguyen et al., 2014; Pradel, 2009; Pradel and Gross, 2009, 2012; Pradel et al., 2010, 2012; Reger et al., 2013; Sun et al., 2015; Wu et al., 2011; Zhong et al., 2009). 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.

In this paper, we present the first in-depth investigation of the effectiveness of existing specs for finding bugs when runtime verification is combined with software testing. We consider a spec *effective for bug finding* if it can catch true bugs but does not generate too many false alarms. To be effective for bug finding, a spec must encode not only API-usage correctness conditions, but also capture mistakes that developers commonly make. 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 (Formal Systems Laboratory, 2014; Jin et al., 2012a; Luo et al., 2014), 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 (Lee et al., 2012) and used in previous studies on the efficiency and scalability of runtime verification (Legunsen et al., 2015; Luo et al., 2014; Purandare et al., 2013). We also use 17 specs that were mined automatically from large traces (Pradel and Gross, 2009) and were used in spec mining studies (Pradel and Gross, 2012; Pradel et al., 2012).

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 in this paper, we focus on the *effectiveness* question: "How good are the specs for finding bugs?" Second, most previous evaluations were conducted on the DaCapo benchmarks (Blackburn et al., 2006) (with at most 14 projects) or with a smaller number of open-source projects; in contrast, we use open-source projects—200 in our main experiment, plus another 18 projects in our validation study. 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 original developers be the judges of the bugs that we discovered from manually inspecting spec violations.

In our main experiment, we monitored 182 manually written and 17 automatically mined specs while running 18065 manually written and 2135081 automatically generated test methods in 200 open-source projects. We manually inspected a subset of the spec violations, and sent pull requests for violations that we believed to be bugs. On the positive, the average runtime overhead of monitoring was under 4.3×, and developers already fixed 76 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 specs. All the violations of manually written specs that we did not inspect were from 21 specs which, as discussed more in Section 5.2.1, we found from manual inspection of the specs to be broken, not able to find TrueBugs, or to always generate false alarms. We also inspected a sample of 200 from 1141 violations of automatically mined specs. We observed overall false alarm rates of 82.81% and 97.89%, respectively among the inspected violations of manually written and automatically mined specs. 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. However, in many cases, the specs appear completely ineffective and should not be used at all. Among the 200 projects in our main experiment, only 99 had at least one violation that we inspected. Of the 99, only 30 had less than 100% false alarm rate; all 333 violations that we inspected in the remaining 69 projects were not TrueBugs.

Inspecting spec violations and submitting pull requests to developers took an estimated 1,200 hours in our main experiment 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 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" (Gabel and Su, 2010; Nguyen and Khoo, 2011; Pradel et al., 2012) 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 these reasons, we chose to report to the developers those cases where at least one of the authors believed (and could convince the others) that a violation indicated some problem in the current version of the code.

Three years after we conducted our main experiment and reported the results (Legunsen et al., 2016b), we conducted a validation study to see whether the results of our main experiments hold even more broadly. We followed the same procedure as in our main experiments to monitor 161 manually written specs while running 1698 developer written test classes in 18 open-source projects. The 18 projects in our validation study are larger (in terms of lines of code), have longer running tests, have better code coverage, are widely used and are relatively more actively maintained than the 200 projects in our main experiment. The 161 specs that we used in our validation study are those that remain after removing the 21 specs that we previously reported to be broken (Legunsen et al., 2016b). We did not use automatically mined specs or automatically generated test methods in our validation study because they did not help find too many additional bugs in our main experiment.

The results from our validation study, especially in terms of false alarms, were quite similar to those in our main experiment. The average runtime overhead in our validation study was greater, at 7.4×, even though we monitored 21 fewer specs simultaneously during each test execution. 85.95% of violations that we inspected were false alarms (compared to 82.81% in our main experiment). Only 11 of 161 specs helped find a bug (same absolute number as in our main experiment). Among the specs that helped find a bug, the false alarm rate was 41.8%, compared to 45.51% in our main experiment. We have followed the same procedure as in our main experiment to manually inspect and classify all 742 spec violations that we found in our validation study. We report on our ongoing process of reporting violations in our validation study that we believe to be true bugs to the developers.

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 is paying off. Indeed, the technology has matured enough to incur potentially 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:

```
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 }
```

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

- ⋆ **Large-Scale Evaluation.** We present the first large-scale evaluation of runtime verification during software testing, with 199 specs and 218 open-source projects. The results show that runtime verification has potentially acceptable overhead during testing and can find important bugs that developers are willing to fix, but the specs are largely ineffective and generate way too many false alarms.
- ⋆ **Analysis of Effectiveness.** We analyze reasons for bug-finding ineffectiveness of existing specs. In particular, we analyze the high rates of false alarms along different dimensions of program and spec characteristics, 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. The data from our main experiment is publicly available (Legunsen et al., 2016c), and we plan to release the data from our validation study.

2 Background

We briefly describe runtime verification of specs in JavaMOP (Chen and Roşu, 2003; Formal Systems Laboratory, 2014; Jin et al., 2012a; Luo et al., 2014; Meredith et al., 2008). The spec, `Collection_SynchronizedCollection` (`CSC`), shown in Figure 1, is one of the manually written specs in our study. `CSC` was earlier proposed by Bodden et al. (2007b) (they called it `ASyncIteration`) to check for cases where a synchronized `Collection`'s `Iterator` is accessed from some non-synchronized code. Figure 1 shows the three parts of a JavaMOP spec: lines 3–10 define the *events* that are 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

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

Fig. 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.
```

Fig. 3: A sample violation

`Collection`). The monitor subsequently listens for the events `syncMk` (line 5), `asyncMk` (line 7), and `access` (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 ever 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, which is simplified from one of the six bugs that we found in `TestNG`, a widely used unit-testing framework. 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 on `im`, 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

In this section we describe how we conducted our experiments. Specifically, we describe the open-source projects used in our main experiment, the manually written and automatically mined specs that were monitored while running tests in the projects, and how we automatically generated tests using Randoop (Pacheco and Ernst, 2007, 2016; Pacheco et al., 2007). We also explain our procedure for running JavaMOP on the projects and for inspecting the resulting violations. Additional details are provided for the setup of the validation study in Section 4.5

Table 1: Statistics of 200 projects used in our study

PID	Project	SHA	LOC	ManTests	AutoTests
P1	Altoros.YCSB	bfcfe23a	7290	1	–
P2	LogBlock.LogBlock-2	40548aad	875	1	–
P3	edanuff.CassandraCompositeType	6d09cceb	1234	1	5427
P4	jriecken.gae-java-mini-profiler	80f3a59e	908	8	92058
P5	mqtt	f4384253	11478	18	–
P6	plista.kornakapi	178061c3	3088	2	21594
P7	threerings.playn	c969160c	38388	139	–
P8	tbuktu.ntru	8126929e	7715	70	–
P9	OpenGamma.ElSql	db6c6d07	2581	160	11034
P10	sematext.ActionGenerator	10f4a3e6	1864	7	–
P11	vivin.GenericTree	15c59c99	677	49	7787
P12	hoverruan.weiboclient4j	6ca0c73f	8748	34	1229
P13	joda-time	cc35fb2e	85000	4157	12123
P14	IvanTrendafilov.Confucius	2c302878	1203	84	23196
P15	mikebrock.jboss-websockets	fd03a4ef	1736	1	6668
P16	b3log.b3log-latke	afb48c40	24399	76	–
P17	Thomas-S-B.visualee	410a80f0	3574	76	8164
P18	asterisk-java	b07617fe	39498	220	33632
P19	Cue.lucene-interval-fields	8f8bff6d	736	9	13162
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.OAuth2.0ProviderForJava	db5e1d06	2654	47	–
P25	htrace	c32ec0b1	2521	11	–
P26	ptgoetz.storm-jms	d152d72f	1085	2	–
P27	UrbanCode.terraform	d67ac40c	12108	4	3069
P28	pignlproc	1a609980	2296	19	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
		SUM	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 main experiment 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. For example, in a multi-module Maven project, a failing test in one module may completely prevent running of the tests in other modules. 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 time or memory 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 on GitHub (Legunsen et al., 2016c).

Table 1 lists some basic statistics about the 200 projects used in our main experiment. PID either starts with "P" to provide the short ID of a project in which we found some real bug(s), 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 test methods, and AutoTests is the number of automatically generated test methods. "–" marks that we did not have Randoop test methods, 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 method within the time limit. For F69 and N101, ManTests and AutoTests show the sums for all respective projects. The rows SUM, AVG, MIN, and MAX are the sum, average, minimum, and maximum across all projects in each column, respectively.

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 (Lee et al., 2012; Luo et al., 2014) and 17 automatically mined specs (Pradel, 2015; Pradel et al., 2012). We next describe our rationale and procedure for selecting each set of specs.

3.2.1 Manually Written Specs

We used 182 manually written JavaMOP specs (Legunsen et al., 2015; Luo et al., 2014), which are publicly available (Formal Systems Laboratory, 2016). The specs were originally written by Lee et al. (Lee et al., 2012), who read Javadoc comments in four widely-used packages (`java.lang`, `java.net`, `java.io`, and `java.util`), then annotated and formalized sentences describing "must", "should" or "it is better to" conditions. The specs are formalized using finite-state machines (FSM), extended regular expressions (ERE), linear temporal logic (LTL), and context-free grammars (CFG). JavaMOP can monitor specs written in any formalism for which a suitable logic plugin exists.

To illustrate manual formalization of specs, consider again the `CSC` spec (Formal Systems Laboratory, 2015a) from Section 2. It was formalized as an ERE 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"* (Oracle, 2015d). Section 2 explained `CSC` in detail, line-by-line. As mentioned there, this spec had been also used earlier (Bodden et al., 2007b). By analyzing Javadoc comments, Lee et al. (Lee et al., 2012) ended up with some of the same specs that others had formalized before. Monitoring `CSC` in our experiments using JavaMOP revealed bugs in several widely used projects, including `TestNG`, `ActiveMQ`, and `XStream`. However, our experiments also revealed a number of 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. (Pradel, 2009, 2015;

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?
Pradel et al. (2010)	DaCapo+OSS	12	n/a	n/a	n/a
Reger et al. (2013)	n/a	n/a	n/a	n/a	n/a
Krka et al. (2014)	n/a	8	n/a	n/a	n/a
Nguyen and Khoo (2011)	DaCapo	7	43.00	20	no
Dallmeier et al. (2010)	OSS+JDK	7	n/a	n/a	n/a
Zhong et al. (2009)	OSS	5	73.90	100	yes
Wu et al. (2011)	OSS	8	n/a	1	no
Pradel and Gross (2012)	DaCapo	10	0.00	54	no
Lemieux (2015); Lemieux et al. (2015)	Sel-Classes	3	n/a	n/a	n/a
Wasylkowski and Zeller (2009)	OSS	6	58.00	9	yes
Chen et al. (2015)	OSS	4	n/a	n/a	n/a
Nguyen et al. (2014)	OSS	3559	n/a	n/a	n/a
Gabel and Su (2010)	DaCapo	11	70.00	11	no
Beckman and Nori (2011)	DaCapo	1	n/a	n/a	n/a
Le Goues and Weimer (2009)	OSS	7	5.00	265	no
Pradel et al. (2012)	DaCapo	12	49.00	26	no
Sun et al. (2015)	Sel-Classes	15	n/a	n/a	n/a

Pradel and Gross, 2009; Pradel et al., 2012). We searched for mined specs for Java by conducting a mini-survey of the spec mining literature in which we also investigated how spec mining was previously evaluated.

Paper Search: We searched for spec mining papers on DBLP (Ley, 2015) 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 relevant papers that mined Java API specs that we could use. We omitted related papers, e.g., a survey by Robillard et al. (2013), 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. Pradel et al. (2012) 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 whose authors we emailed. Although 7 papers report finding bugs while evaluating mined specs, only 2 papers 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. (2014) who used thousands of projects but only to apply statistical techniques to mine specs from all these projects 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 that surveyed on spec mining. In fact, we find even higher false alarms rates. As was the case for the manually written specs, our experiments also revealed issues and opportunities for improvement 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 (Formal Systems Laboratory, 2015b). First, the JavaMOP compiler generates a Java agent (Oracle, 2015a) from the specs to be monitored, enabling dynamic instrumentation of code running in the Java Virtual Machine. Next, the Maven 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 fully automated JavaMOP agent creation, changing `pom.xml`, monitoring each project, and post-processing results. This allowed us to scale our experiments to 218 projects and 199 specs.

In each set of experiments, we ran the tests in each project twice. First, we ran without integrating JavaMOP to measure the base test-running time and as a check that the tests in the project pass by themselves. We then integrated JavaMOP and reran the tests to measure test-running time with monitoring and to record violations. To compute overhead of monitoring without printing violations to the standard output, 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; Surefire is used to run tests in all Maven projects) and test-mocking frameworks (which we found to have unexpected interactions with JavaMOP, as mentioned in Section 3.1). In our main experiment we ran JavaMOP to monitor the execution of manually written tests on a 64-bit computer with 8 cores of 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 (Pacheco and Ernst, 2007, 2016; Pacheco et al., 2007) to automatically generate additional tests in our main experiment. We generated tests and monitored them on a Core™ i7-4700MQ 2.40GHz Quad-Core processor PC with 8GB of RAM, running Ubuntu 15.04, Java 7 or 8 (as required by the project), and Randoop heap usage limited to 4GB. We ran Randoop on all 151 single-module Maven projects (out of total 200 single- and multi-module projects), which were easier to automate than multi-module Maven projects.

We limited test-generation time to 1 minutes and 5 minutes. After generating tests, 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 1 minutes and 5 minutes. Therefore, we decided to use the tests generated in 5 minutes and did not increase the time limit for Randoop any further. Other researchers who used Randoop also found tests generated in different intervals to behave similarly (Pacheco et al., 2008; Shamshiri et al., 2015; Tan et al., 2012).

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)*.

In our main experiment, we manually 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 (Cochran, 1977): 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 developers accept the resulting pull requests. Therefore, multiple authors inspected most violations. For manually written tests and manually written specs, two reviewers first 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 a code smell; 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. HardToInspect cases were assigned to a third reviewer who inspected them, classified them as FalseAlarm or TrueBug and passed the results back to the original two reviewers for confirmation. We did not resolve a few of the HardToInspect violations, but rather chose to focus on writing and submitting pull requests for the 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 4 minutes and 54 minutes 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. Section 5 discusses in detail the inspection results and other results from our experiments.

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 carefully prepared pull requests. We did not simply submit bug reports indicating the violation of a spec in a code base; 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.

4 Results

Our goal is to evaluate the bug-finding effectiveness of existing specs when monitoring test runs in open-source projects. We answer these 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?
RQ4 How do false alarm rates vary by different program and spec characteristics?
RQ5 How repeatable are RQ1 to RQ4 results on a different set of projects?

RQ1 to RQ4 are about the results of our main experiment, while RQ5 is about the results of our validation study. We added RQ4 to start addressing questions that we commonly received while presenting RQ1 to RQ3 for our main experiment. Audiences often wanted to know whether there were program or spec characteristics that influenced false alarm rates.

4.1 RQ1: Runtime Overhead of Monitoring

Table 3 shows the runtime overhead (Overhead[%]) from monitoring all 182 specs (All 182 Specs) and 42 (Selected 42 Specs) 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

Table 3: Dynamic (DV) and static (SV) violations, and overhead (Overhead[%]) for 182 and 42 manually written Specs. ManTests: manually written tests; AutoTests: automatically generated tests; PID, SUM, AVG, MIN, MAX, "–": same headers as in Table 1

PID	ManTests						AutoTests			
	All 182 Specs			Selected 42 Specs			All 182 Specs		Selected 42 Specs	
	DV	SV	Overhead[%]	DV	SV	Overhead[%]	DV	SV	DV	SV
P1	144	23	558.01	13	4	187.93	–	–	–	–
P2	4	4	157.08	1	1	50.96	–	–	–	–
P3	22	4	214.28	20	2	110.72	11	1	0	0
P4	12	8	381.82	0	0	157.72	20	1	20	1
P5	58212	72	6.85	412	2	-28.37	–	–	–	–
P6	172	9	349.50	24	2	155.36	43	31	0	0
P7	19	11	469.43	1	1	201.39	–	–	–	–
P8	45	15	75.30	27	7	27.88	0	0	0	0
P9	6706	36	1983.61	384	9	239.98	58313	4	0	0
P10	1036	25	309.79	961	3	128.17	–	–	–	–
P11	1045	40	357.91	26	5	128.86	9	3	7	1
P12	6512	35	579.83	0	0	248.97	945	66	558	16
P13	856	220	665.28	236	95	245.26	0	0	0	0
P14	79	5	208.81	74	1	123.09	75258	9	75242	9
P15	0	0	8.80	0	0	2.30	289	5	287	3
P16	3514	85	149.07	167	13	72.25	–	–	–	–
P17	58	24	242.24	37	13	126.38	18	1	18	1
P18	261	47	189.34	2	1	104.53	6797	28	6717	6
P19	903	92	643.10	746	5	284.76	12522	5	12520	3
P20	28473	21	205.76	27977	1	105.98	1493	3	1493	3
P21	517	52	805.23	21	4	324.51	3935	29	7241	4
P22	251125	11	655.35	181430	4	338.72	0	0	0	0
P23	1305	44	164.34	1038	16	67.46	0	0	0	0
P24	787	57	637.20	88	5	228.58	–	–	–	–
P25	227	84	147.16	31	10	69.88	–	–	–	–
P26	239	55	120.15	7	5	51.50	–	–	–	–
P27	10	4	180.33	0	0	84.23	7941	58	1322	8
P28	3868	88	38.99	414	13	14.57	173	5	0	0
P29	3127	110	88.72	29	13	4.30	0	0	0	0
P30	13166	188	1308.77	467	29	616.59	–	–	–	–
F69	737644	2977	31744.19	85120	269	13297.49	2468500	574	96111	64
N101	907063	817	22381.38	0	0	8106.04	2052	81	0	0
SUM	2027151	5263	66027.60	299753	533	25877.95	2638319	904	201536	119
AVG	10135.75	26.32	330.14	1498.77	2.67	129.39	17472.31	5.99	1651.93	0.98
MIN	0	0	-7.44	0	0	-28.37	0	0	0	0
MAX	693388	275	3289.99	181430	95	1036.57	1585833	83	75242	16

the garbage-collection behavior of the instrumented program, causing it to run faster (Jin et al., 2011, 2012b; Meredith et al., 2008).

The average runtime overhead was 129.39% when monitoring only the 42 inspected specs and 330.14% when monitoring all 182 manually written specs. Therefore, the overhead of simultaneously monitoring all specs is under 4.3× on average. We believe this runtime overhead may be acceptable during testing (not in production), 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. Times are not shown in Table 3. Also, the fact that the overhead significantly varies with the number of monitored specs suggests that *incremental monitoring* (Legunsen et al., 2015) during software evolution, where only specs that are related to code changes are monitored in newer versions, may not only be desirable but actually critical as more and more effective specs will be made available. The relatively small increases in total test-running time with JavaMOP reflects the tremendous progress made in the research community over the last decade to make runtime verification more efficient. However, as discussed further in sections 4.3 and 4.4, our experiments also showed that the false alarm rate when monitoring these specs is still too high. For runtime verification to become more useful at finding bugs

during software development, the research community needs to start paying more attention to finding more effective specs to monitor.

Table 3 also shows the number of dynamic (DV) and static (SV) violations from monitoring 182 and 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 5 minutes for 13 projects). Even when the number of dynamic violations, DV, is relatively high, the overhead remains reasonable. Further, although the average overhead is 4.3× (with all 182 specs) and 2.3× (with 42 specs), several projects have much higher overheads, e.g., `P9` and `P30`, and the maximum overhead was 33x.

4.2 RQ2: Bugs Found

We found a total of 114 SV (static violations) 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 necessarily 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 main experiments (with then latest revisions of the projects), 2 unique bugs that we found were already fixed in the current latest revisions. For the remaining 95 bugs, we submitted pull requests (as described in Section 3.5), with 76 already accepted and only 3 rejected; the remaining 16 pull requests are still pending.

Table 4: Bug Reports. Submitted/Accepted/Rejected: bug reports submitted/accepted/rejected, Pending: awaiting developer response, AFixed: fixed in more recent version, VB: non-unique bugs, UB: unique bugs.

Status	VB	UB
Total Bugs Found	114	97
Submitted	111	95
Accepted	88	76
Pending	19	16
AFixed	3	2
Rejected	4	3

Table 4 summarizes the status of the bug reports that we submitted to the developers of the open-source projects. Counting only unique bug reports (UB), 78 of them were either accepted by the developers (76 bugs) or were already fixed by the developers in a more recent version of the code, before we submitted a bug report (2 bugs). In Table 4, VB is the number of bugs found by a direct count of static violations that we classified as TrueBug. However, because some violations occur in third-party libraries shared by multiple subjects, we also report UB, which is the number of unique bugs that we found. We discuss some bugs that were accepted and some that were rejected in Section 5.

Table 5: 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

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 the inspected violations. For manually written specs, we inspected a total of 652 violations—533 from manually written tests and 119 from automatically generated tests. Table 5 shows, for each project in which we inspected violations, the project ID (PID), the number of inspected static violations (SV), the number of violations in each classification (HTI, TB, and FA), and the 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 SUM 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).

In Table 5, 40 of the 110 TrueBugs, i.e. 36.4%, were in project P13—`joda-time`, a very mature and widely used project. Without P13, the average overall FAR in our main experiment would have been even higher. Therefore, we checked to see if there were any

Table 6: Per-spec inspection summary. Column headers are same as in Table 5

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

special characteristics that made P13 have so many TrueBugs (note that 57.89% of violations in P13 were FalseAlarms). As far as we can see, there are no special characteristics that make P13 have so many TrueBugs. 37 of the 40 TrueBug that we found in P13 were violations of the `OutputStream`-related spec, `ByteArrayOutputStream_FlushBeforeRetrieve` (`BAOS`). `BAOS` is the spec whose violations helped us find the most number of TrueBugs in both the main experiments and in the validation study (118 TrueBugs in total). P13 had many unit tests—33, to be specific (Emopers, 2015)—in multiple classes that used similar logic to create and write data to `OutputStream` objects in a manner that violated `BAOS`. We counted each static violation of `BAOS` in P13 as a separate TrueBug since they occurred at different program locations. If the `OutputStream`-manipulating code had been refactored by extracting a single method that all 33 unit tests invoke, we would have found only 4 TrueBugs in P13. However, our experimental methodology is to perform runtime verification on open-source projects as they existed, same as what developers working with that version of code would see. All the pull requests that we submitted for the 33 violations in P13 were accepted by the developers, and we discuss the accepted pull requests of P13 in more detail in Section 5.1.1.

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, note that 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 the developers of some project(s). Second, 119 specs were never violated, so they only increased the runtime overhead (if their monitors were created). These specs may get violated if monitored on other projects. We leave as future work to manually inspect the specs that were not violated. 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 among bug-finding specs, `Iterator_HasNext` with 97.40% FAR, is the *de facto* pedagogical example spec in research papers on spec mining and runtime verification. Section 5.2.2 discusses why specs generate so many FalseAlarms.

For automatically mined specs that we monitored, 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

[1] These specs are publicly available (Pradel, 2015).

the existing specs are rather ineffective for finding bugs, because they raise too many false alarms. In Section 5, we discuss the bugs that we found, developer responses to these bugs and some opportunities for improvement that we found among the monitored specs, in order to give a qualitative view of the false alarm rates.

4.4 RQ4: False Alarm Rates by Program and Spec Characteristics

One question that we often received while presenting the results of RQ1 to RQ3 is whether there are certain characteristics of programs and/or specs which make runtime verification more (or less) prone to high FAR. There are pragmatic reasons why it is important to answer this question. Assuming there is a correlation between high FAR and some characteristics. Then, e.g., developers whose projects have those characteristics may choose to not do runtime verification, or they may choose to use only a subset of specs that that have a higher chance to find bugs for their projects.

To begin answering this question, we further analyzed the high FAR along several dimensions, to identify whether there are program and/or spec characteristics for which it may be lower. For now, we use simple program and spec characteristics that are easy to measure or readily available. Different sets of simple program features were shown to be effective for prediction in other software engineering tasks, e.g., predictive mutation testing (Mao et al., 2019; Zhang et al., 2016, 2018). Specifically, we analyzed whether FAR was lower by violation location (i.e., project code vs. third-party libraries), Maven structure (i.e, single-module vs. multi-module Maven projects), test type (i.e., manually written vs. automatically generated tests), or spec type (i.e., manually written or automatically mined specs). We also computed the correlation between FAR and code coverage, and between FAR and project size. Finally we checked the FAR across *severity levels* of manually written specs, which were assigned by the authors of the specs. When formalizing the specs, Lee et al. (2012) assigned a severity level of *suggestion* to a spec if its violations are expected to mostly indicate bad programming practice, not necessarily bugs. A severity level of *warning* means that a spec is expected to sometimes generate false alarms, but its violations can also be indicative of TrueBugs. Finally, a severity level of *error* means that violations are expected to mostly be indicative of bugs. We leave as future work to come up with other features that may help predict whether violations of a spec in a program are more likely to be false alarms.

FAR along different Dimensions of Program Structure: Table 7 (top part —Manually written specs) shows the FAR breakdown by violation location, Maven structure, test type, and spec type, for manually written specs. Violations in third-party libraries (Libraries) had 86.55% FAR, while violations in the project code (Project code) had 80.82% FAR. Violations in single-module (Single-module) vs. multi-module Maven projects (Multi-module) had 81.87% vs. 86.23% FAR, and violations for manually written tests (ManTests) vs. automatically generated tests (AutoTests) had 82.51% vs. 84.21% FAR. Along all dimensions, FAR for manually written specs ranged from 80.82% to 86.55%. The similar FARs across all these dimensions suggests that the FARs are mostly due to inherent (in)effectiveness of the specs and less due to these 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 7 (bottom part—Automatically mined specs) 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 were false

Table 7: Split of inspection results along various dimensions. Column headers are same as in Table 5

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

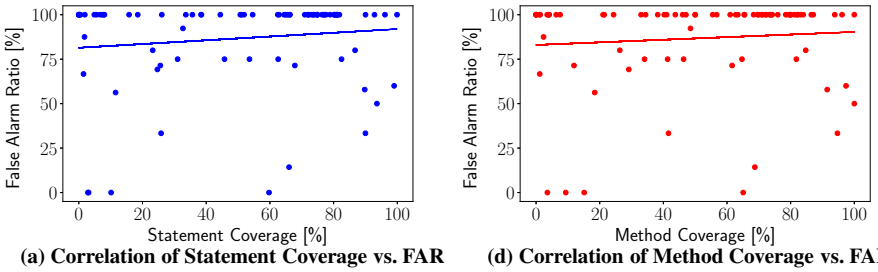
alarms). Compared within different dimensions, the FAR values were similar, e.g., 100.00% for violations in libraries vs. somewhat lower 94.87% for violations in the project code, showing a consistent relationship with violations of manually written specs. The FAR for violations in single-module Maven projects (97.84%) was about the same as that for multi-module Maven projects (98.04%).

Correlation of FAR with Code Coverage: Figure 4 shows the correlation of FAR with code coverage (Figures 4a—4e), and the correlation of FAR with code size (Figure 4f). Since we are interested to see the relationship between coverage and FAR, attempted to measure coverage only for the 99 projects in which there were violations. We ran JaCoCo (The JaCoCo Team, 2018) to collect coverage from these 99 projects, but could only obtain coverage information for 83 of them. For the other 16, either JaCoCo did not run "out of the box" (8 projects) or the projects could no longer be built (8 projects). Figure 4f plots the correlation for all 99 projects in which we inspected some violation, since measuring code size does not require any special project configuration. The captions of Figures 4a to 4f contain, in parentheses, the Pearson's r coefficients which indicate the strength of the correlation. The results show that FAR is slightly positively correlated with statement coverage (Figure 4a), line coverage (Figure 4b), branch coverage (Figure 4c), method coverage (Figure 4d) and class coverage (Figure 4e). This makes sense intuitively and suggests that the better a project is tested, the more likely it is that spec violations will be false alarms. On the other hand, we observe a negative correlation of FAR with code size (Figure 4f); violations in larger projects are less likely to be false alarms. This negative correlation of FAR and code size remains (but is weaker) even when the outlier in Figure 4f is removed, showing that larger (and likely more mature) projects may benefit more from runtime verification during testing, in terms of finding more bugs.

Table 8: False Alarm Ratios by Severity Level of Specs

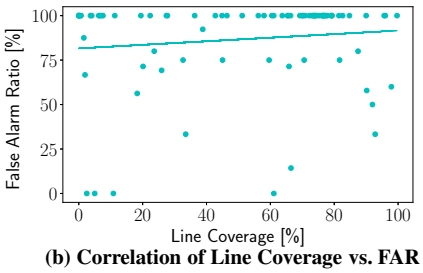
Severity	No. of Specs	SV	FAR[%]
Error	24	184	90.98
Warning	15	343	76.84
Suggestion	3	127	88.09

FAR by Spec Severity Level: Finally, Table 8 shows the FAR by severity level of the 42 specs whose violations we manually inspected. There, Severity is the severity level assigned

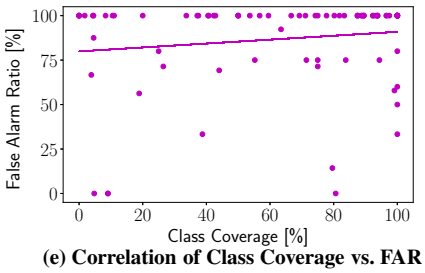


(a) Correlation of Statement Coverage vs. FAR (Pearson's r: 0.13)

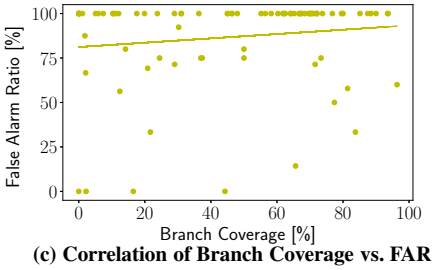
(d) Correlation of Method Coverage vs. FAR (Pearson's r: 0.09)



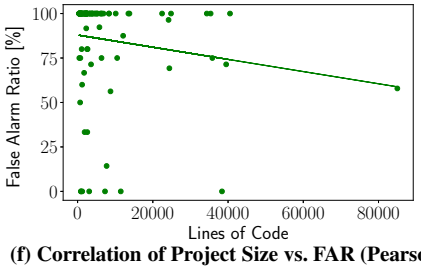
(b) Correlation of Line Coverage vs. FAR (Pearson's r: 0.12)



(e) Correlation of Class Coverage vs. FAR (Pearson's r: 0.15)



(c) Correlation of Branch Coverage vs. FAR (Pearson's r: 0.14)



(f) Correlation of Project Size vs. FAR (Pearson's r: -0.15)

Fig. 4: Correlation of FAR with Coverage and Code Size in our main experiment

by Lee et al. (2012), No. of Specs is the number of specs per severity level, SV is the number of violations that we inspected in each severity level, and FAR[%] is the false alarm rate per severity level. Recall that Lee et al. (2012) manually assigned severity levels to specs in the following way (each manually-written spec is formalized from the English language in the Javadoc). If the violation (or matching) of a spec is very likely indicative of runtime error or failure, then that spec is assigned the highest severity level of `error`. If a spec violation sometimes, but not always, indicates an error, then that spec is assigned the second severity level of `warning`. A spec is assigned the lowest severity level `suggestion` if it is a bad programming practice that cannot lead to an error or failure during execution.

An example spec with `error` severity level is `Collections_SynchronizedCollection`, formalized from language described in Section 3.2.1. Note the word, "imperative", indicates that each departure from the Javadoc recommendation is an error. An example spec that is assigned a `warning` is `URLDecoder_DecodeUTF8`, which is violated if a non-UTF-8 encoding is used to decode a URL string. The `warning` severity level arises from the

sentence in the Javadoc from which `URLDecoder_DecodeUTF8` was formalized: *"...UTF-8 should be used. Not doing so may introduce incompatibilites"* Oracle (2015c). In other words, if a non-UTF-8 encoding is used, it may or may not introduce incompatibilites. Finally, an example of spec with severity level `suggestion` is `Math_ContendedRandom`. It is formalized based on the following language in the Javadoc of `Math.random()` method: *"if many threads need to generate pseudorandom numbers at a great rate, it may reduce contention for each thread to have its own pseudorandom-number generator"* Oracle (2015b). In this case, `Math.random()` is guaranteed to be properly synchronized so no error or failure can arise from violating this spec. However, it is a bad programming practice that can lead to sub-optimal runtime performance. Therefore, Lee et al. (2012) assigned severity level `suggestion`. Observe from Table 6 that all three specs whose severity levels we used as examples helped us find at least one TrueBug. Therefore, it is reasonable to examine the FAR across these severity levels more broadly.

There are several interesting findings about severity levels and FAR from Table 8. First, specs with a severity level of `error` have the highest FAR. One can view these in two ways: either that these specs are rather ineffective since the Java runtime should capture most `error`s and developers would notice them more often, or that the specs are `error` severity level are so critical that even the bugs they find can be crucial so it is worth to inspect the violations. The second finding from Table 8 is that majority of the violations that we inspected are from specs with `warning` severity level, but these also have the lowest FAR among the severity levels. This is somewhat surprising because violations from specs at `warning` severity level are expected to have higher FAR than those with `error` severity level. The third finding is that even specs with `suggestion` severity level can find bugs (they did not have 100% FAR), so one may still want to monitor them. Finally, as far as we know, we are the first to analyze the false alarms from these specs by severity level, and Table 8 shows that "field testing" specs like we do may be better for assigning severity level than simply relying on the imprecise language of the API documentation. For example, specs with low FAR across many projects and inspected violations may be considered more important and assigned `error` severity level.

Conclusion on FAR Analysis: Along all dimensions of program structure, program characteristics, and spec severity level that we investigated, FAR was too high. We did not find some dimension along which the FAR was low.

4.5 RQ5: Validation Study

We carried out a validation study to see if the conclusions from our main experiment regarding overhead and false alarm rates hold on a different set of Java projects.

Project and Spec Selection: In our validation study, we used 18 projects with different characteristics than the 200 projects in our main experiment. We monitored 161 manually written specs while executing the tests in these 18 projects; these are the specs that remain after removing the 21 that we report as being broken in Section 5.2.1 from the full set of 182. We did not use automatically mined specs and automatically generated tests which did not yield many additional bugs in our main experiment. Table 9 shows statistics about the 18 projects in our replication study. The 18 projects in our validation study are single-module Maven Java projects from our previous work on optimizing regression testing (Legunsen et al., 2016a, 2017), and were selected because, compared with the 200 projects in our main experiments, they (1) have longer average test running times without JavaMOP (43.9 seconds vs. 5.1 seconds), (2) have more test classes on average (94.3 vs. 7.4), (3) have

Table 9: Statistics of projects used in our replication study

PID	Project	SHA	LOC	ManTests	DV	Overhead[%]
R1	apache-sling-event	139dab1d	12015	11	83	186.24
R2	commons-dbcp	c9bec5ce	31154	26	58	96.64
R3	commons-math	cbae75b9	174446	469	3448	674.29
R4	addthis.stream-lib	aa58062a	8641	24	3370	1606.68
R5	HikariCP	4bb67f76	11983	31	5109	710.87
R6	imglib.imglib2	4d782e21	49320	68	601	260.06
R7	commons-codec	208d3dba	20198	53	192675	361.72
R8	commons-io	06033035	30283	97	1049083	465.13
R9	OpenTripPlanner	12cb13bd	84446	126	107399	3493.50
R10	square.javapoet	59ba4332	8595	16	619	510.82
R11	commons-lang	fc409b57	77359	143	360	266.48
R12	commons-pool	3e9d9bb4	14029	17	16	24.52
R13	jackson-databind	277031a3	113946	379	644	945.40
R14	commons-imaging	2e8379b5	38655	67	49073	484.95
R15	commons-dbutils	82bb44fe	6885	22	1	324.66
R16	commons-email	e3b0d447	6772	14	49	213.02
R17	commons-fileupload	774ef160	4707	12	24350	384.46
R18	jackson-core	75f9a04e	44253	123	12724	469.32
		AVG	40982.61	94.33	80536.77	637.70
		SUM	737687	1698	1449662	11478.76
		MIN	4707	11	1	24.52
		MAX	174446	469	1049083	3493.50

slightly higher code coverage (statement coverage: 58% vs. 51%), (4) are relatively more mature (11 of them are Apache Software Foundation projects) (5) contain more code (40.9 vs 6.1 KLOC), and (6) are more are actively maintained (several of the 200 projects in our main experiments are now dormant). All of these characteristics hold despite the fact that we had to remove very few tests from some of the validation study projects in order to bypass errors due to JavaMOP instrumentation.

Setup: We followed the same procedure as in our main experiment for using JavaMOP to monitor the test executions in the 18 projects in our validation study. We manually inspected all 742 spec violations that JavaMOP generated—which is a larger number than the 652 violations of manually written specs in our main experiment. During manual inspection, we again followed the same procedure as in the main experiment, whereby multiple reviewers inspected each violation and then double-checked to reach an agreement. Thus we have some confidence about the classification of true bugs and false alarms in our validation study.

Evaluation: We answer again RQ1 (overhead of JavaMOP, Section 4.1), RQ2 (bugs found), RQ3 (false alarm rates of JavaMOP, Section 4.3), and RQ4 (false alarm rates along different dimensions, Section 4.4) again, but only for the 18 projects and 161 manually-written specs in our validation study.

Runtime Overheads: The AVG row of the Overhead[%] column in Table 9 shows that the average runtime overhead of the 18 projects in our validation study was 7.4×. This is higher than the average overhead that we observed in our main experiment, even when running with all 182 specs. We point out three interesting findings about the overhead of runtime verification. First, the average overhead from our validation study confirms that the overheads of runtime verification are high, and were not just a function of the relatively shorter-running tests in the projects that we used for our main experiment. Second, while 7.4× may still be tolerable in some testing scenarios (e.g., perform runtime verification overnight), it will be important in the future to further reduce these overheads so that developers can use runtime

verification more frequently, ideally as code is being written. Therefore, we have developed the first set of evolution-aware runtime verification techniques that reduce the accumulated overhead of runtime verification across multiple program versions (Legunsen et al., 2015; Legunsen et al., 2019). Finally, as in the main experiment, we see that some projects have much higher overheads than the average, e.g, `R4` and `R9`.

Bugs Found: We recently submitted pull requests 75 of the 97 TrueBugs that we found during our validation study to the developers of the respective projects. Among these submitted pull requests, 13 have been accepted, 16 were rejected and the rest are pending. We are in the process of submitting pull requests for the remaining 22 TrueBugs. We plan to make this data public after we recieve more responses on more pending pull requests in our validation study, like we did for our main experiment (Legunsen et al., 2016c). The 13 TrueBugs that were accepted so far are from 5 different projects, while 15 of 16 rejected TrueBugs are from the same project—`imglib.imglib2`, PID: R6. All the TrueBugs that were rejected in R6 are from specs that monitor against fetching an element from an `Iterator` without first checking whether the `Iterator` has more elements. Interestingly, at least two R6 developers looked into our pull requests for these rejected TrueBugs and agreed that the changes in our pull requests were a good idea. However, they eventually rejected these pull requests because they thought that our particular way of going about the fixes would complicate their project's code and design. One of the developers said, "Adding these tests to every place where cursors are used bogs down the logic of the tests...we should be testing the cursor iteration in its own test method, and then in other tests, we can assume the iteration works properly" (Emopers, 2019). In keeping with the general methodology from our main experiments where we let the developers decide what is a bug in their own code, we have marked these as rejected pull requests. However, we still think that these are bugs, and that with more discussion with the developers, it may be possible in the future to work with the R6 developers to make more localized changes that they can accept. For the pull requests that were accepted so far, the developers either accepted them without comment or requested more changes before acceptance.

False Alarm Rates: Tables 10 and 11 show the false alarm rates among all 18 projects and all 32 specs that generated violations in our validation study. From Table 10, we see that only two projects had FAR at most 50%, and also that four projects had 100% FAR, including `R18` in which the most static violations were generated. In sum, the FAR among the projects in our replication study was high, and is similar to the FAR for projects in the main experiment (Table 5). From Table 11, we see that most specs that generated violations also had high FAR. All violations from five specs were classified as true bugs, and only one spec with non-zero FAR had below 50%. From Tables 10 and 11 we can also see that 640 out of 735 non-HTI violations were false alarms, leading to an overall FAR of 86.3% which is similarly as high as the 82.81% FAR that we found among inspected specs in our main experiment. Lastly, three specs had relatively low FAR in both the main experiment and in our validation study: `ByteArrayOutputStream_FlushBeforeRetrieve`, `Collections_SynchronizedCollection`, and `Collections_SynchronizedMap`.

False Alarm Rates along different Dimensions: Table 12, Figure 5, and Table 13 show the FAR in our validation study along different project dimensions, correlation of FAR with coverage and code size, and FAR by severity levels of the specs. The dimensions that we consider in Table 12 are fewer than in the main experiment, since all projects in our validation study are single-module Maven projects, and we consider only manually written specs and tests. However, for the violations in the "Project code" row of Table 12, we further analyze them based on whether the violation is in the code under test ("Code under test" row) or in the test code ("Test code" row). This analysis of violations along the dimensions

Table 10: Per-project inspection summary from our replication study. SV: static violations; HTI: hard to inspect; TB: true bugs; FA: false alarms; FAR[%]: false alarm rate

PID	SV	HTI	TB	FA	FAR[%]
R1	5	0	3	2	40.00
R2	4	0	2	2	50.00
R3	54	0	21	33	61.11
R4	19	0	6	13	68.42
R5	25	2	7	16	69.57
R6	75	0	17	58	77.33
R7	35	1	5	29	85.29
R8	90	0	12	78	86.67
R9	61	1	8	52	86.67
R10	9	1	1	7	87.50
R11	61	0	5	56	91.80
R12	13	0	1	12	92.31
R13	102	0	5	97	95.10
R14	52	0	2	50	96.15
R15	1	0	0	1	100.00
R16	8	2	0	6	100.00
R17	12	0	0	12	100.00
R18	116	0	0	116	100.00
SUM	742	7	95	640	–

Table 11: Per-spec inspection summary for our replication study. Column headers are same as in Table 5

Spec	SV	HTI	TB	FA	FAR[%]
Collection_HashCode	4	0	4	0	0.00
Collections_SynchronizedCollection	3	0	3	0	0.00
Collections_SynchronizedMap	1	0	1	0	0.00
Object_MonitorOwner	2	0	2	0	0.00
Serializable_NoArgConstructor	2	0	0	2	100.00
ByteArrayOutputStream_FlushBeforeRetrieve	94	1	63	30	32.26
StringTokenizer_HasMoreElements	6	0	1	5	83.33
ListIterator_hasNextPrevious	47	0	8	39	82.98
Iterator_HasNext	198	2	13	183	93.37
InputStream_ManipulateAfterClose	16	0	0	16	100.00
Reader_ManipulateAfterClose	8	0	0	8	100.00
24 Specs with 100% FAR	387	4	0	383	100.00
SUM	742	7	95	640	84.12

Table 12: False alarm ratios along different dimensions in our replication study

Dimension	SV	HTI	TB	FA	FAR[%]
All specs	742	7	97	638	86.80
Libraries	49	4	8	37	82.22
Project code	693	3	87	603	87.39
Code under test	217	0	23	194	89.40
Test code	476	3	64	409	86.47

of code under test and test code is new, and was not in our earlier paper about the main experiment (Legunsen et al., 2016b). The coverage plots in Figure 5 are only for 13 of the 18 projects; the other projects did not work with JaCoCo out of the box. As with the other RQs, the results of our analysis of FAR along several dimensions in our replication study was very similar to the results in the main experiment.

Table 13: False Alarm Ratios by Severity Level of Specs in our replication study

Severity	No. of Specs	SV	FAR[%]
error	20	105	80.00
warning	9	373	87.99
suggestion	2	256	100.00

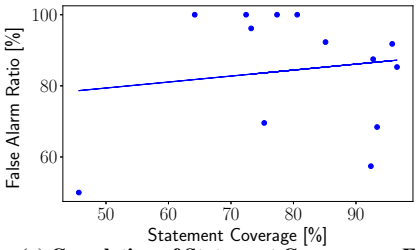
From Table 12, violations in third-party library code tended to have lower FAR (82.22%) than violations in the project's code (86.19%). For the 693 violations in the project's code, 476 were in the test code, which is more than twice as much as the violations in code under test (217). We are the first to analyze the proportion of all spec violations that happen in the code under test as well as in the test code. Interestingly, in our validation study, much fewer violations happened in third-party library code than in the project's code. Finally the FAR was very high in all of the dimensions that we considered.

One observation from Table 12 is that more than half of the TrueBugs were in the test code, and not in the code under test. Thus, it is reasonable to question whether it is worthwhile to monitor the test code against the specs, and whether one can expect TrueBugs that occur in test code to be fixed in practice. We answer both of these questions in the affirmative, for three reasons. First, it is essential to monitor test code in order to perform runtime verification during software testing. If one does not monitor test code, one can miss to find bugs because violations that happen due to a sequence of events originating in the test code but ending in the code under test will not be observed. Second, it is important for test code to be of high quality, because, during software evolution, developers commonly rely on the pass/or fail outcomes of tests to make decisions on how to proceed with their development. Thus, if a runtime verification of test code can help find bugs in the test code, then those should be fixed to improve the quality of the tests. A test did not previously fail due to the bug exposed by the violation may fail in the future due to that violation and not due to changes that developers make. Finally, we find empirically, that developers are quite open to fixing violations in the test code. For example, we discussed in Section 4.3 that, in our main experiments, developers of the widely-used `joda-time` accepted all 40 pull requests that we submitted, even though 33 of them were in the test code.

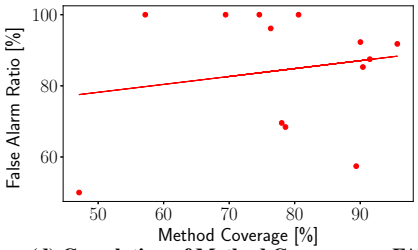
Concerning coverage and code size, we found similar correlations with FAR between our main experiment and the validation study. As can be seen in Figure 5, FAR is slightly positively correlated with code coverage (Figures 5a—5e), and negatively correlated with code size (Figure 5f); the negative correlation remains even when the outlier in Figure 5f is removed. The major dissimilarity with the results of the main experiment is with class coverage, which was much more weakly correlated with FAR than in the main experiment (Pearson's r of 0.15 in main experiment vs. -0.02 in the validation study).

FAR was high for all spec severity levels (Table 13). However, some of the trends are different than in the main experiment. As expected by Lee et al. (2012), violations from specs with `error` had the lowest FAR, and all violations from specs with severity level `suggestion` were all false alarms. As in the main experiments, most violations were from specs with severity level `warning`. Taken together with the results of analyzing FAR by spec severity level in the main experiment, we still conclude that testing the specs may still a better way to assign severity levels to the specs; even specs with severity level of `suggestion` helped find bugs.

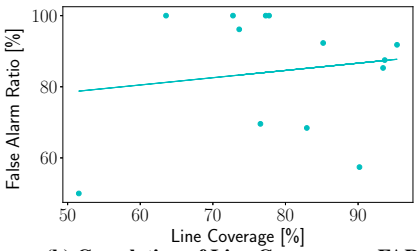
Validation Study Conclusions: Several of the results from our validation study confirm the results from our main experiment: we still find that the runtime overhead of monitoring test executions is high but potentially tolerable during testing, that runtime verification can find many bugs during testing, and that false alarm rates are very high across many dimensions of



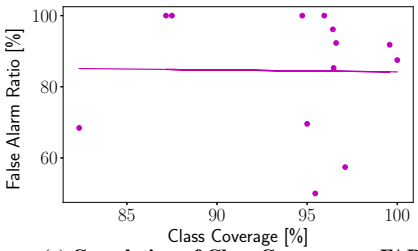
(a) Correlation of Statement Coverage vs. FAR (Pearson's r: 0.14)



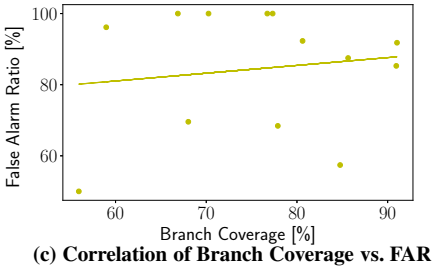
(d) Correlation of Method Coverage vs. FAR (Pearson's r: 0.18)



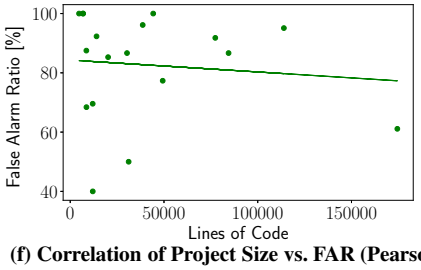
(b) Correlation of Line Coverage vs. FAR (Pearson's r: 0.15)



(e) Correlation of Class Coverage vs. FAR (Pearson's r: -0.02)



(c) Correlation of Branch Coverage vs. FAR (Pearson's r: 0.14)



(f) Correlation of Project Size vs. FAR (Pearson's r: -0.10)

Fig. 5: Correlation of FAR with Coverage and Code Size in our validation study

analysis. We found higher runtime overheads for the longer-running projects in our validation study, which suggests that there is still more work to be done in the research community on making runtime verification even more efficient during testing.

5 Analysis of Results

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

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 main experiment was `joda-time`, *"the* de facto *standard date and time library for Java prior to Java SE 8"* (Joda, 2016). 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` was formalized from the Javadoc of the `writeTo()` method of `java.io.ByteArrayOutputStream`, which states, *"Writes the complete contents of this byte array output stream to the specified output stream argument, as if by calling the output stream's write method... using `out.write(buf,0,count)`."* Lee et al. (2012) argue that "When an `OutputStream` (or its subclass) instance is built on top of an underlying `ByteArrayOutputStream` instance, it should be flushed or closed before the contents of the `ByteArrayOutputStream` instance is retrieved." Essentially, `BAOS` catches cases where an underlying `ByteArrayOutputStream` is not closed or flushed before retrieving the contents of the enclosing stream. The fix included in our pull requests was 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` which is defined only for `java.util.Map`. 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— `TestNG`, `XStream`, and `Spring-Beans`. 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. Part of the Java API text from which `java.net.InetSocketAddress` was formalized states, "A port number of zero will let the system pick up an ephemeral port in a bind operation".

Finally, we found 4 bugs from monitoring the specs that Pradel et al. (2012) 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 pending pull request) was a violation of `FSM33` (Figure 6), where `removeFirst()` was invoked on a `java.util.LinkedList` object without first checking that it was not empty.

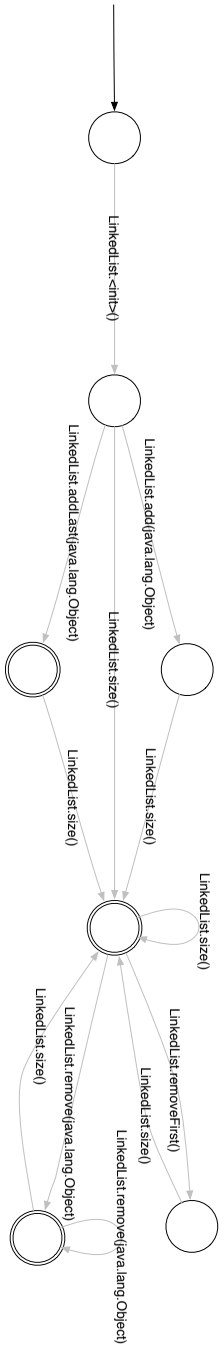


Fig. 6: Automatically mined spec from Pradel and Gross (2009) that found a bug

```
1 public void testWriteToSystemOutLikeStream() {
2  ByteArrayOutputStream bytes = new ByteArrayOutputStream();
3  new JsonAppendableWriter(new PrintStream(bytes)).object()...done();
4  assertEquals("...", new String(bytes.toByteArray(), ...));
5 }
6
7 class JsonAppendableWriter ... {
8  public void done() throws JsonWriterException {
9   super.doneInternal();
10   if (appendable instanceof Flushable) {
11    try {
12     ((Flushable) appendable).flush();
13    } catch (IOException e) {
14     throw new JsonWriterException(e);
15    }
16   }
17  }
18 }
```

Fig. 7: Code for rejected bug in `threerings.playn`

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, tagged it as `wontfix`, and responded: *"...there's no need to synchronize it... As explicitly stated in the documentation, XStream is not thread-safe during setup... There are a lot of places in the code base, where this applies. XStream is only thread-safe while you actually run the (un-)marshalling. Especially for XStream annotations you should preprocess the classes. If you process the annotations on-the-fly, you may no longer process XML concurrently. Again, 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 is shown in Figure 7 (lines 1–5). JavaMOP finds a violation because `toByteArray()` was invoked on line 4 for the `ByteArrayOutputStream` object, `bytes` (declared on line 2), without any intervening invocation of `flush()`. However, `done()` (lines 8–17) flushes the `bytes` stream. On line 12 `bytes` is cast 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. It is interesting that `BAOS` helped find many bugs, but it still misses some conditions and causes false alarms—some of the 55.28%

of inspected 123 violations of BAOS that we inspected were false alarms. We discuss other issues that we found with the specs in Section 5.2.

5.2 Issues with Monitored Specs

We next discuss the reasons 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: In our main experiment, 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 (`StringBuffer_SingleThreadUsage`, `File_DeleteTempFile`, `ObjectInput_Close` and `ObjectOutput_Close`) 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—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 that are not widely used in our projects; only 7 of 200 projects in our main experiment mention these packages; (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. (2012) 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. A major challenge is how to automatically mitigate the statistical nature of mining techniques, which makes them tend to overfit the projects from which they are mined (Thummalapenta and Xie, 2009).

5.2.2 Analysis of False Alarms

The monitored specs reported many false alarms mainly because the specs (i) did not encode all necessary 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. In Section 5.1.2 (Figure 7), we showed one violation of the BAOS spec that we wrongly classified as a TrueBug, but which turned out to be a FalseAlarm because the BAOS

```
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();
```

Fig. 8: False alarms from the `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 }
```

Fig. 9: False alarms from the `Map_UnsafeIterator` spec

spec did not encode the fact that a method could be invoked on an object after casting it to a supertype. Other specs with many false alarms also missed to capture some conditions.

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 `Iterator` object. Violations of `Iterator_HasNext` led us to discover 4 accepted bugs in `Thomas-S-B.visualee` project, and other researchers had previously used `Iterator_HasNext` to find some real bugs in production `AspectJ` (bug IDs #218167 and #218171 (Wasylkowski and Zeller, 2009)). However, `Iterator_HasNext` also reports a huge number of false alarms—150 of 154 non-HardToInspect violations were false alarms—with FAR of 97.40%. Figure 8 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 8 illustrate only a few of the valid uses of the `next()` method that were violations of the `Iterator_HasNext` spec during our main experiment. 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 underlying the `Iterator`.

`Map_UnsafeIterator` is another spec with a lot of false alarms because it does not capture enough; 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 the problem, consider the code snippet in Figure 9. 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... when writing [a JavaMOP] spec and it is up to the user; one can put more effort into writing more fine-grained specs... (using conditional pointcut to obtain more accurate instrumentation) 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 (`Closeable_MultipleClose` and `Closeable_MeaninglessClose`), 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 (Gabel and Su, 2010; Nguyen and Khoo, 2011; Pradel et al., 2012), 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. Note that these responses are from a small subset of pull requests for which the developers made comments. Majority of the accepted pull requests were without developer comments, and the fact that most of them were accepted shortly after we submitted suggests that developers considered them important.

Developers Asked for More: On the same day that we submitted a pull request for a `BAOS` violation, the `apache.gora` developers asked us to help check other portions of their code: *" Hi, thanks for the pull request....Are there any other instances of this behavior throughout the codebase? If so it would be real nice to catch them all at the same time...I just undertook a quick scan of the codebase for ByteArrayOutputStream, I found the following instances. Can you please check these out as well?"* We did not check those earlier because they were not dynamically executed and hence did not have similar violations. 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 when the reporter will report, and...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-profile r`, 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 of Specification Engineering

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, in the future.
(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; these may be improved to carry more program state in the monitor and be more precise. 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: (Legunsen et al., 2016c). We found interesting results from submitted pull requests, e.g., even "buggy" specs like `BAOS` can lead to accepted pull requests, and feedback from developers can help improve the specs.
(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. (Pradel et al., 2012) defined some heuristics-based automated techniques for filtering out violations while statically checking mined specs, while Gabel and Su (Gabel and Su, 2012) as well as Nguyen and Khoo (Nguyen and Khoo, 2011) proposed techniques for checking that mined specs are true specs. 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 (Formal Systems Laboratory, 2016); 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 218 projects that we used were quite diverse in size, number of tests, test coverage, 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 in the literature and allows to simultaneously monitor specs written in different formalism, 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 (76 so far) or reject (3 so far).

8 Related Work

Weimer and Necula (2005), Le Goues and Weimer (2009), and Gabel and Su (2012) all mention high false alarm rates when evaluating spec mining algorithms, but our work is different in the sense that we evaluate the bug-detection capability of existing "true specs". Because our main goal was to evaluate the effectiveness of existing specs on open-source projects, i.e., "in the wild", we used a larger number of projects in our study than in any prior work related to specs that we are aware of. Most of the literature on spec mining that we surveyed (see Section 3.2.2) were either evaluated on benchmarks (e.g., 7 of 17 papers were evaluated on DaCapo) or on a small set of open-source projects (6 of 17 papers were evaluated on 4–7 open-source projects). Some papers did not even use any projects but instead the mined specs were evaluated (in 4 of the 17 papers) by means of recall and precision against existing

specs derived from a small collection of classes or from prior work. Evaluating specs on a larger set of open-source projects can provide a more indicative picture of the bug-finding effectiveness of the specs in code that developers commonly write.

Our *process* for evaluating mined spec is similar in scale and approach to that used for evaluating Doc2Spec (Zhong et al., 2009). There, the authors obtained violations of mined specs in 138 open-source projects, manually inspected them to filter out false alarms (73.9% of the violations) and reported suspected bugs to the developers of the projects in which the violations occurred. Our work is different (i) in the way that the violations are obtained (they perform static analysis of selected client code that use the API from which the specs are mined, we find dynamic violations obtained from running tests that shipped with our subject applications), (ii) in scope (they only evaluate automatically mined specs, but we evaluate both manually written and automatically mined specs) and (ii) in purpose (their goal was to show, *specifically* how good Doc2Spec was for finding bugs, our goal is to show how good, *generally*, are specs when they are monitored against test executions in open-source projects).

Thummalapenta and Xie (Thummalapenta and Xie, 2009) proposed an automated approach, Alattin, for reducing the false alarms generated by automatically mined specs. Their idea is to not just report frequently-occurring code patterns as part of a mined spec, but to also include "alternate patterns"—code patterns that were seen less frequently during the spec mining process, but which, if not incorporated into the resulting spec can lead to many FalseAlarms when checking the spec. For example, we showed in Figure 8 some legitimate code that would nevertheless violate the `Iterator_HasNext` spec. Alattin would encode some of these as alternate patterns in the spec, so that some of those FalseAlarms are not generated at runtime. More ideas like Alattin, as well as extending Alattin to work on other kinds of formalism that JavaMOP supports, would be worthwhile to pursue in the future, for augmenting some of the more effective specs in our study.

9 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 extensive 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 (Legunsen et al., 2016c) 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 formalism, 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.

Acknowledgements Karl Hajal, Milica Hadzi-Tanovic and Igor Lima helped with inspecting violations in our validation study and submitting pull requests. 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, CCF-1439957, CNS-1646305, CNS-1740916, and CCF-1763788. Wajih Ul Hassan was partially supported by the Sohaib and Sara Abassi Fellowship. We gratefully acknowledge support for research on testing from Microsoft and Qualcomm.

References

Allan C, Avgustinov P, Christensen AS, Hendren L, Kuzins S, Lhoták O, de Moor O, Sereni D, Sittampalam G, Tibble J (2005) Adding trace matching with free variables to AspectJ. In: OOPSLA, pp 345–364

Arnold M, Vechev M, Yahav E (2008) QVM: An efficient runtime for detecting defects in deployed systems. In: OOPSLA, pp 143–162

Beckman NE, Nori AV (2011) Probabilistic, modular and scalable inference of typestate specifications. In: PLDI, pp 211–221

Blackburn SM, Garner R, Hoffmann C, Khang AM, McKinley KS, Bentzur R, Diwan A, Feinberg D, Frampton D, Guyer SZ, Hirzel M, Hosking A, Jump M, Lee H, Moss JEB, Phansalkar A, Stefanović D, VanDrunen T, von Dincklage D, Wiedermann B (2006) The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA, pp 169–190

Bodden E (2011) MOPBox: A library approach to runtime verification. In: RV Tool Demo, pp 365–369

Bodden E, Hendren L, Lam P, Lhoták O, Naeem NA (2007a) Collaborative runtime verification with tracematches. In: RV, pp 22–37

Bodden E, Hendren LJ, Lhoták O (2007b) A staged static program analysis to improve the performance of runtime monitoring. In: ECOOP, pp 525–549

Bodden E, Lam P, Hendren L (2008) Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In: FSE, pp 36–47

Chen D, Zhang Y, Wang R, Li X, Peng L, Wei W (2015) Mining universal specification based on probabilistic model. In: SEKE, pp 471–476

Chen F, Roşu G (2003) Towards monitoring-oriented programming: A paradigm combining specification and implementation. In: RV, pp 108–127

Cochran WG (1977) Sampling techniques. John Wiley & Sons

Dallmeier V, Knopp N, Mallon C, Hack S, Zeller A (2010) Generating test cases for specification mining. In: ISSTA, pp 85–96

Dwyer MB, Purandare R, Person S (2010) Runtime verification in context: Can optimizing error detection improve fault diagnosis? In: RV, pp 36–50

Emopers (2015) Closing ObjectOutputStream before calling toByteArray on the underlying ByteArrayOutputStream. `https://github.com/JodaOrg/joda-time/pull/339`

Emopers (2019) Checking the validity of input ListIterators. `https://github.com/imglib/imglib2/pull/259`

Forejt V, Kwiatkowska M, Parker D, Qu H, Ujma M (2012) Incremental runtime verification of probabilistic systems. In: RV, pp 314–319

Formal Systems Laboratory (2014) JavaMOP. `http://fsl.cs.illinois.edu/index.php/JavaMOP`

Formal Systems Laboratory (2015a) Collections_SynchronizedCollection. http://fsl.cs.illinois.edu/annotated-java/__properties/html/java/util/Collections_SynchronizedCollection.html

Formal Systems Laboratory (2015b) JavaMOPAgent Documentation. https://github.com/runtimeverification/javamop/blob/master/docs/JavaMOPAgentUsage.md

Formal Systems Laboratory (2016) FSL Specification Database. https://runtimeverification.com/monitor/propertydb

Gabel M, Su Z (2010) Online inference and enforcement of temporal properties. In: ICSE, pp 15–24

Gabel M, Su Z (2012) Testing mined specifications. In: FSE, pp 1–11

Hussein S, Meredith P, Roşu G (2012) Security-policy monitoring and enforcement with JavaMOP. In: PLAS, pp 1–11

Jin D, Meredith PO, Griffith D, Roşu G (2011) Garbage collection for monitoring parametric properties. In: PLDI, pp 415–424

Jin D, Meredith PO, Lee C, Roşu G (2012a) JavaMOP: Efficient parametric runtime monitoring framework. In: ICSE Demo, pp 1427–1430

Jin D, Meredith PO, Roşu G (2012b) Scalable parametric runtime monitoring. Tech. rep., Computer Science Dept., UIUC

Joda S (2016) Joda-Time. http://www.joda.org/joda-time/

Karaorman M, Freeman J (2004) jMonitor: Java runtime event specification and monitoring library. In: RV, pp 181 – 200

Krka I, Brun Y, Medvidovic N (2014) Automatic mining of specifications from invocation traces and method invariants. In: FSE, pp 178–189

Le Goues C, Weimer W (2009) Specification mining with few false positives. In: TACAS, pp 292–306

Lee C, Chen F, Roşu G (2011) Mining parametric specifications. In: ICSE, pp 591–600

Lee C, Jin D, Meredith PO, Roşu G (2012) Towards categorizing and formalizing the JDK API. Tech. rep., Computer Science Dept., UIUC

Legunsen O, Marinov D, Roşu G (2015) Evolution-aware monitoring-oriented programming. In: ICSE NIER, pp 615–618

Legunsen O, Hariri F, Shi A, Lu Y, Zhang L, Marinov D (2016a) An extensive study of static regression test selection in modern software evolution. In: FSE, pp 583–594

Legunsen O, Hassan WU, Xu X, Rosu G, Marinov D (2016b) How good are the specs? a study of the bug-finding effectiveness of existing Java API specifications. In: ASE, pp 602–613

Legunsen O, Hassan WU, Xu X, Roşu G, Marinov D (2016c) Supplementary material for this paper. http://fsl.cs.illinois.edu/spec-eval

Legunsen O, Shi A, Marinov D (2017) STARTS: STAtic Regression Test Selection. In: ASE, pp 949–954

Legunsen O, Zhang Y, Hadzi-Tanovic M, Roşu G, Marinov D (2019) Techniques for evolution-aware runtime verification. In: ICST, pp 300–311

Lemieux C (2015) Mining temporal properties of data invariants. In: ICSE SRC, pp 751–753

Lemieux C, Park D, Beschastnikh I (2015) General LTL specification mining. In: ASE, pp 81–92

Ley M (2015) CompleteSearch DBLP. http://www.dblp.org/search/index.php

Luo Q, Zhang Y, Lee C, Jin D, Meredith PO, Şerbănuţă TF, Roşu G (2014) RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In: RV, pp 285–300

Mao D, Chen L, Zhang L (2019) An extensive study on cross-project predictive mutation testing. In: ICST, pp 160–171

Meredith P, Roşu G (2013) Efficient parametric runtime verification with deterministic string rewriting. In: ASE, pp 70–80

Meredith P, Jin D, Chen F, Roşu G (2008) Efficient monitoring of parametric context-free patterns. In: ASE, pp 148–157

Navabpour S, Wu CWW, Bonakdarpour B, Fischmeister S (2011) Efficient techniques for near-optimal instrumentation in time-triggered runtime verification. In: RV, pp 208–222

Nguyen AC, Khoo SC (2011) Extracting significant specifications from mining through mutation testing. In: ICFEM, pp 472–488

Nguyen HA, Dyer R, Nguyen TN, Rajan H (2014) Mining preconditions of APIs in large-scale code corpus. In: FSE, pp 166–177

Oracle (2015a) java.lang.instrument. `http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html`

Oracle (2015b) java.lang.Math. `https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html`

Oracle (2015c) java.net.URL. `https://docs.oracle.com/javase/7/docs/api/java/net/URL.html`

Oracle (2015d) java.util.Collections. `https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html`

Pacheco C, Ernst MD (2007) Randoop: Feedback-directed random testing for Java. In: OOPSLA Companion, pp 815–816

Pacheco C, Ernst MD (2016) Randoop. `https://randoop.github.io/randoop/`

Pacheco C, Lahiri SK, Ernst MD, Ball T (2007) Feedback-directed random test generation. In: ICSE, pp 75–84

Pacheco C, Lahiri SK, Ball T (2008) Finding errors in .NET with feedback-directed random testing. In: ISSTA, pp 87–96

Pradel M (2009) Dynamically inferring, refining, and checking API usage protocols. In: OOPSLA Companion, pp 773–774

Pradel M (2015) Statically checking API protocol conformance with mined multi-object specifications (supplementary material). `http://mp.binaervarianz.de/icse2012-statically/`

Pradel M, Gross TR (2009) Automatic generation of object usage specifications from large method traces. In: ASE, pp 371–382

Pradel M, Gross TR (2012) Leveraging test generation and specification mining for automated bug detection without false positives. In: ICSE, pp 288–298

Pradel M, Bichsel P, Gross TR (2010) A framework for the evaluation of specification miners based on finite state machines. In: ICSM, pp 1–10

Pradel M, Jaspan C, Aldrich J, Gross TR (2012) Statically checking API protocol conformance with mined multi-object specifications. In: ICSE, pp 925–935

Purandare R, Dwyer MB, Elbaum S (2013) Optimizing monitoring of finite state properties through monitor compaction. In: ISSTA, pp 280–290

Reger G, Barringer H, Rydeheard D (2013) A pattern-based approach to parametric specification mining. In: ASE, pp 658–663

Robillard MP, Bodden E, Kawrykow D, Mezini M, Ratchford T (2013) Automated API property inference techniques. TSE 39(5):613–637

Shamshiri S, Just R, Rojas J, Fraser G, McMinn P, Arcuri A (2015) Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges. In: ASE, pp 201–211

Sun J, Xiao H, Liu Y, Lin SW, Qin S (2015) TLV: Abstraction through testing, learning, and validation. In: ESEC/FSE, pp 698–709

Tan SH, Marinov D, Tan L, Leavens GT (2012) @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In: ICST, pp 260–269

The JaCoCo Team (2018) JaCoCo Java Code Coverage Library. `https://www.jacoco.org/jacoco`

Thummalapenta S, Xie T (2009) Alattin: Mining alternative patterns for detecting neglected conditions. In: ASE, pp 283–294

Wasylkowski A, Zeller A (2009) Mining temporal specifications from object usage. In: ASE, pp 295–306

Weimer W, Necula G (2005) Mining temporal specifications for error detection. In: TACAS, pp 461–476

Wu CWW, Kumar D, Bonakdarpour B, Fischmeister S (2013) Reducing monitoring overhead by integrating event- and time-triggered techniques. In: RV, pp 304–321

Wu Q, Liang G, Wang Q, Xie T, Mei H (2011) Iterative mining of resource-releasing specifications. In: ASE, pp 233–242

Zhang J, Wang Z, Zhang L, Hao D, Zang L, Cheng S, Zhang L (2016) Predictive mutation testing. In: ISSTA, pp 342–353

Zhang J, Zhang L, Harman M, Hao D, Jia Y, Zhang L (2018) Predictive mutation testing. TSE pp 898–918

Zhong H, Zhang L, Xie T, Mei H (2009) Inferring resource specifications from natural language API documentation. In: ASE, pp 307–318