Demystifying the Challenges of Formally Specifying API Properties for Runtime Verification

Leopoldo Teixeira, Breno Miranda, Henrique Rebêlo, Marcelo d'Amorim
Federal University of Pernambuco
Recife, PE, Brazil
{lmt,bafm,hemr,damorim}@cin.ufpe.br

***Abstract*—Runtime Verification (RV) is a technique to monitor formally-specified properties of the software during its execution. RV has shown to be very effective for bug finding. Unfortunately, RV typically relies on formal specification languages and learning those languages be costly for developers. This paper reports on a study to assess the challenges to specify API properties for the purpose of RV. To that end, we wrote SIESTA, a minimalist specification language, extending Java with two features (the ability to catch calls to specified methods and the ability to access the event history of a given object), and asked inexperienced developers (students) to write specifications in that language for certain parts of the Java API. Among our findings, we observed that 40% of the specifications written by the students matched the ground truth perfectly. The main messages of this work are that 1) it is feasible to use a simple imperative language for specifying properties without significant loss of generality; and that 2) developers are capable of writing specifications in the (programming) language they feel comfortable.**

***Index Terms*—runtime verification, specification languages, user studies**

I. Introduction

Runtime Verification (RV) [38] is a technique to monitor software properties during execution. In RV, properties are instrumented into the code and monitored during program execution. If an execution does not satisfy a given property, a property *violation* is reported to developers. RV helps to find bugs that occur when behavioral properties are violated. We use the definition of behavioral property coined by Robillard et al. [56] —"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". Legunsen et al. [33], [35], [36] recently showed that performing RV during test execution detected hundreds of bugs—violations of properties of the Java API—that existing tests written by developers missed.

One important hurdle to the adoption of RV is the availability of specifications to check against at runtime. To write specifications, developers need to be familiar with a specification language, typically a formal language. However, the cognitive effort that developers need to put to learn those languages can be too high and justify the poor adoption of RV in Software Engineering practice. Designers of specification languages often assume these languages should be programming-language agnostic and support a variety of formalisms to specify properties. Practitioners, on their turn, often assume that good background in logic and deep understanding of the application domain are critical to specify properties.

Let us consider the case of JavaMOP [20] to illustrate the complexity of a specification language. Figure 1a shows the specification of an API property written in the JavaMOP syntax [21]. The property states that any access to an element of a synchronized collection must be protected by the monitor associated with that collection [2]. Although the property is conceptually simple to understand for a programmer who is familiar with Java concurrency, the example shows that learning a rich specification language can be daunting to developers. They need to understand the constructs of the language (e.g., declaration of events) and the formalisms supported by the language (e.g., Extended Regular Expressions, Finite State Machines, Linear Temporal Logics, etc.). It is worth noting that the study conducted by Legunsen et al. [35], [36], mentioned above, used JavaMOP specifications, which were written by the JavaMOP team as part of a separate study [49].

JavaMOP is an example of a declarative specification language. Imperative specification languages, in contrast, build on a given target language to circumvent some of the aforementioned problems. JML [30], for instance, enables a developer to annotate Java code with pre- and post-conditions and class invariants written in a Java-like language. Unfortunately, JML does not offer support for developers to write protocol properties, i.e., properties that impose restrictions on the ordering of events associated with an API. These properties are common, considering the set of Java API properties that we analyzed from the propertyDB dataset [49]. To specify such properties, a developer would need to use a JML extension, such as Typestate JML [28], to precisely describe the usage protocols of Java classes and interfaces in terms of explicit states and transitions (e.g., particular method call sequences). Unfortunately, the more features one adds to a language, the more effort is needed from practitioners to learn, to practice, and to use those features. To sum up, rich specification languages impose an important burden on developers and is certainly one factor to justify the poor practical adoption of techniques that rely on them [15], [40].

Study Questions. This paper reports on a study to assess the challenges of formally specifying API properties for RV. The study aims to answer three questions:

RQ1. Are rich specification languages necessary for RV?

RQ2. Can developers with no training in logic express properties about an API by reading its documentation?

```
1  Collections_SynchronizedCollection(Collection c, Iterator i) {
2   Collection c;
3   creation event sync after() returning(Collection c):
4    call(* Collections.synchronizedCollection(Collection)){ this.c = c;}
5   event syncMk after(Collection c) returning(Iterator i) :
6    call(* Collection+.iterator()) && target(c) && Thread.holdsLock(c){}
7   event asyncMk after(Collection c) returning(Iterator i):
8    call(* Collection+.iterator())&& target(c)&& !Thread.holdsLock(c){}
9   event access before(Iterator i) :
10   call(* Iterator.*(..)) && target(i) && !Thread.holdsLock(this.c){}
11  ere: (sync asyncMk) | (sync syncMk access)
12  @match{ RVMLogging.out.println(/*violation message*/); }}
```

(a) Collections_SynchronizedCollection (CSC) property

```
public class CharSet ... {
 ...
 set = Collections.synchronizedSet(...);
 public boolean contains(final char ch) {
+ synchronized(set) {
    for (final CharRange range : set) {
     if (range.contains(ch)) {
      return true;
     }
    }
+ }
  return false; }}
```

(b) Commons Lang Bug found from CSC violation

Fig. 1: The CSC JavaMOP property and a related bug in Apache project. (Example from [44].)

RQ3. What are the perceptions of developers about the practice of writing specs?

The goal of the first question is to assess the necessity of rich specification languages to describe properties for RV checking. Instead of answering this question directly, which is challenging, we evaluated whether SIESTA, a minimalist specification language we proposed, was sufficient to describe most behavioral properties from an existing set of properties [36]. The goal of the second question is to evaluate the ability of developers with no training in logic and short training (≈1h) in our specification language to write properties. Finally, the goal of the third question is to understand the perceptions of developers about writing specs in general, about using a rich spec languages, like JavaMOP, and about writing specs in the minimalist spec language we proposed.

Summary of Method. To run our experiment, we selected API properties from three popular Java packages—*java.io*, *java.lang*, and *java.util*. We used properties from the PropertyDB [49] dataset as ground truth. PropertyDB is a dataset of JavaMOP specs that includes specs for various parts of the Java API. We restricted our selection to properties listed as "error" and "warning" since those properties are more likely to be associated with real problems [36]. A total of 99 properties satisfied this criterion. Considering RQ1, we selected a sample of those properties and evaluated if they could be specified in our proposed spec language. If not, we revised the language and repeated the process with another sample of 99 properties until exhausting the set of properties. Considering RQ2 and RQ3, the paper reports on an experiment where undergraduate CS students were asked to write specs in the proposed language. We consider students a great fit for this experiment as they are inexperienced. Intuitively, if students do well on this task, more experienced developers are likely to succeed. The students participated in a 1h training session on the proposed specification language and then received an assignment to specify 5 properties of the Java API. The assignment needed to be turned in within a week. It is worth noting that the students shared a spreadsheet with their answers as to enable them to see each other's specs and the example specs provided by instructors.

Summary of Results. Regarding RQ1, we were able to specify 92% of the 99 selected properties using SIESTA, our minimalist spec language. Considering RQ2 and RQ3, the students were able to successfully write specifications perfectly matching the ground truth (modulo variable renaming) in 40% of the cases. If we also consider those that wrote specifications that closely match the ground truth, this number rises to 64%. Overall, students manifested satisfaction in writing specifications and showed preference towards SIESTA. These results show initial, yet strong, evidence that developers could and should write specs. They could because our results indicate that even inexperienced developers can do well and they should because previous studies have shown that writing specs can reveal lots of bugs.

Main Lessons Learned. The main lessons learned from this study are as follows: ⋆ The intrinsic complexity of writing specs is overstated. ⋆ Documentation is a great source of information to write specifications (and it should be available for APIs, at least). ⋆ Leveraging the crowd is important for learning specifications. ⋆ Engagement of developers in writing specs should be considered as an educational investment as opposed to a deterrent of productivity. Section VI elaborates on these lessons.

II. Finding Bugs with RV

In this section, we discuss how to find bugs during testing with RV by means of a running example. The RV tool receives as inputs formally specified properties to check at runtime, a program, and tests to exercise the program. It then instruments the properties into the program, and, during program execution, it observes relevant events using monitors to check for violations of properties. When a violation happens, the tool informs the users that the corresponding property was not satisfied. In what follows, we use JavaMOP, a tool that has been used in RV research [19], [23], [41], [42], [50], [51], to illustrate this process.

Figure 1a shows the JavaMOP specification [21] of the `Collections_SynchronizedCollection` property (henceforth called `CSC`). Properties have three parts: (1) *event definitions*, which specify relevant events triggered during program executions; (2) *a specification*, which is a logical formula over the events; and (3) *a handler*, which consists of code that executes if events violate the specification.

`CSC` defines four events (lines 3–10). The `sync` event (lines 3–4) is triggered after calling `Collections.synchronizedCollection` to create a synchronized `Collection` `c`. The `syncMk` event (lines 5–6) is triggered after obtaining an iterator `i` of `c` in a thread that locks `c`. In contrast, `asyncMk` (lines 7–8) is triggered after getting `i` from `c` in a thread that does not lock `c`. Lastly, `access` (lines 9–10) is triggered before accessing `i` in a thread that does not lock `c`.

Line 11 specifies the property using an Extended Regular Expression (ERE), which matches if either (1) `i` is created from `c` without locking `c`, or (2) `i` is created from `c` after locking `c` but `i` is subsequently accessed in a thread that does not lock `c`. JavaMOP supports additional logical formalisms for expressing property specifications, e.g., FSM, CFG, LTL.

If the event sequence established by the specification is matched during runtime, then the handler (line 12) is triggered. Arbitrary code might be placed inside a handler. In this case, the handler prints a violation warning users that the `CSC` property was violated and the program may have a bug.

Prior work [32], [39] used JavaMOP to specify properties of Java API packages (e.g., `java.lang`, `java.io`, `java.util`, and `java.net`) by reading through their documentation. They classified each property into one of three *severity levels*. In this work, we focus on two levels, namely *error*, which are violations that likely indicate bugs, and *warnings*, which are violations that might reveal bugs in some cases. The third level refers to violations that indicate bad coding practices.

Figure 1b illustrates how RV can be used to find a bug. The code snippet shows a bug in the code of Apache Commons Lang [3], a utilities library providing additional methods for the core classes in the `java.lang` API. This bug was revealed by JavaMOP with a violation of the `CSC` property. Lines of code starting with "+" highlight the bug fix. If we ignore such lines and consider only the buggy version, executing line 3 creates the synchronized `set Collection`, which triggers the `sync` event. Afterwards, an iterator is created without locking `set` (line 6), triggering `asyncMk`. The `CSC` specification establishes that this event sequence is a violation, so JavaMOP reports it.

III. The Specification Language

This section informally presents the syntax and semantics of SIESTA (**S**imple **I**mperativ**E** **S**pecifica**T**ion l**A**nguage), the specification language that we proposed and used in this work.

A. Rationale

This paper reports on a study about writing specifications. We realized that developers would feel discouraged to participate in the study had we expected them to have a good grasp of logic (to formally describe properties) and a good grasp of some technology to describe program events (e.g., join points [27]). As such, we decided that a simple specification language was critical to increase engagement of developers in participating in the study. The central requirements we established for SIESTA are:

1) It should be developed on top of Java, which is the programming language whose API properties we wanted to specify and the language that lots of developers are familiar with [22], [61];
2) It should include the minimal number of features necessary to express various kinds of properties. We do *not* aim for generality (see Section III-E).

It is worth noting, as per requirement 1, that SIESTA follows an *imperative rather than declarative* style [30], [43]. The key reason for that choice was to enable developers to relate the semantics of the specification language with the semantics of the programming language they already know. The methodology we used to design the specification language was driven by examples. The authors partitioned a set of properties and independently wrote specs for the properties on each partition. As needed, they revised the language to cover different cases.

B. Syntax

A monitor in SIESTA is 1) a non-empty sequence of field declarations, with optional initialization code, and 2) a list of handlers. The field declarations part enables a monitor to store state. Unlike JavaMOP, a SIESTA monitor does not support parameterization [23]–[25], [41], [42], which could be useful to avoid redundancy across monitors. The handlers part enables a monitor to capture program events. Figure 2 shows the structure of a monitor in our language. Figure 3 shows a concrete example of a monitor.

```
(<@Before|@After>(<MethodSelector>))+
void <PropertyName>(<ListOfArguments>) { <Body> }
```

Fig. 2: Syntax of a monitor (method calls only).

The **<@Before|@After>** annotations indicate when an event handler will be triggered during program execution: at the entry of a method call (`@Before`) or at the exit of a method call (`@After`). **<MethodSelector>** is a "."-separated sequence of strings. Most often this string will be a fully-qualified name of a method, as the examples from Figure 3 illustrate. Alternatively, one can use the name of the class to denote a constructor or an "*" after a fully-qualified class name to denote all methods of a given class. For simplicity, we chose not to support arbitrary regexes to express event patterns, but we do allow multiple annotations for the same handler function, as indicated by the + symbol, as long as `@Before` and `@After` annotations are not mixed. **<PropertyName>** is the name of the function which contains the property to be monitored. **<ListOfArguments>** is a comma-separated list of arguments. If `<MethodSelector>` denotes a selector of an instance method or class constructor, then it should include the target object and then the list of parameters. Figure 3 shows an example of a monitor that uses this pattern. If `<MethodSelector>` denotes a selector of a class method then it should include the list of parameters of the corresponding class. Finally, if `<MethodSelector>` denotes a list of methods–specified with a string of the form "<class-selector>.*"–then the list of arguments should be "`String methodName, boolean isStatic, Object[] args`" (parameter names are not enforced, similar convention is used in the Java Reflection

API). For `@After` handlers, the list of arguments should be prepended with the two additional arguments “`Object res, Throwable t`”, indicating the values returned on normal and exceptional exits. Note that all of these rules can be statically checked in a precise and efficient way. Figure 4 shows an example of this case. **<Body>** is a list of Java statements, which are responsible for checking and reporting whether a violation occurred or not.

C. Semantics

SIESTA *extends Java* with *two features*: (1) the `@Before` and `@After` annotations, used in the definition of handler functions and (2) a special method, `Object.history()`, that enables the monitor to access the history of events on any given object. A handler with the annotation `@Before(`*fn*`)` is always called at the entry of the functions denoted by *fn*. Likewise, a handler with the `@After` annotation is always executed at the exit of the referred functions. We made the conscious decision to not track field accesses as our dataset had no property that required that feature. The method call `o.history()` yields a list of strings corresponding to the names of methods called on `o` since its creation. The order of the list represents the order in which the methods are called on the respective object. A monitor reports property violations during execution with the call to the `Log.violation()` method, which outputs a string on the standard output.

D. Examples

This section illustrates examples of specs for properties of the Java API.

1) Flush Before Retrieve: Consider the following property about `java.io.OutputStream` objects: “*When an* `OutputStream` *instance is built on top of an underlying* `ByteArrayOutputStream` *instance, it should be flushed or closed before the underlying instance’s* `toByteArray()` *is invoked. Failing to fulfill this requirement may cause* `toByteArray()` *to return incomplete contents.*”.[1] Figure 3 shows the corresponding specification for this property.

```
Set<ByteArrayOutputStream> set = ...

@Before("java.io.FilterOutputStream")
void onCreate(FilterOutputStream fos,
    ↪ ByteArrayOutputStream bos){
  set.add(bos); /* track bos */ }

@Before("java.io.ByteArrayOutputStream.toByteArray")
void vioFlushBeforeRetrieve(ByteArrayOutputStream os){
  if (set.contains(os)) {
    List<String> history = os.history();
    String[] interesting = new String[]{"flush", "close"};
    if (filter(history, interesting).size() == 0)
      Log.violation("Vio: Flush before retrieve");
  }}
```

Fig. 3: Flush Before Retrieve.

The `onCreate` handler is triggered on the `OutputStream` constructor that takes a `ByteArrayOutputStream` object as argument. It records that argument in a set. The **`vioFlushBeforeRetrieve`** handler monitors invocations to the method `toByteArray` on `ByteArrayOutputStream` objects (`os`). The monitor raises a violation if it can’t find `flush` or `close` in the event history of the object `os`. The `filter` method is syntactic sugar for the usual filter operation on lists.

2) Pass Empty Map & No Direct Access: Figure 4 shows the specification of a property about `java.util.Map` objects.[2] The property states that the map object passed as argument to the static method `Collections.newSetFromMap()` should be empty when the call is made. In addition, that object should not be directly accessed after returning from the call to `newSetFromMap`. The first handler checks if the `Map` object passed as argument to `newSetFromMap()` is empty. It reports a violation if the collection is non-empty. Otherwise, it records that object in the set `marks`. The second handler checks—at a different point in time—if a message is eventually sent directly to the marked collection. A violation is reported in that case.

```
Set<Map> marks = new HashSet<Map>();

@Before("java.util.Collections.newSetFromMap")
void vioCollections_SynchronizedCollection_1(Map map) {
  if (map.isEmpty()) marks.put(map);
  else Log.violation("Vio: Pass Empty Map."); }

@Before("java.util.Map.*")
void vioCollections_NewSetFromMap_2(String name, boolean
    ↪ isStatic, Object[] args) {
  if (!isStatic && marks.contains(args[0]))
    Log.violation("Vio: Direct Access Forbidden."); }
```

Fig. 4: Pass Empty Map & No Direct Access.

3) StringBuilder Single Thread: Figure 5 shows the specification of a property stating that `StringBuilder` objects should only be accessed by one thread as they are not protected by any lock.[3] The first access to a given `StringBuider` object records, in the field `map`, the `Thread` object that accessed the `StringBuider` object, i.e., the “owner” of the object. The spec omits this parts for space. A violation report is produced if subsequent accesses occur through a different thread.

```
Map<StringBuilder,Thread> map = ...

@Before("java.util.StringBuilder.*")
void vioCollections_NewSetFromMap_2(String name, boolean
    ↪ isStatic, Object[] args) {
  if (isStatic) return false;
  Object sb = args[0];
  Thread own = map.get(sb);
  Thread cur = Thread.currentThread();
  if (own == null) map.put(sb, cur); // first access
  else if (cur != own)
    Log.violation("Vio: StringBuilder Single Thread.");
}
```

Fig. 5: StringBuilder Single Thread.

E. Limitations

The proposed language sacrifices generality in favor of simplicity. The language does have limitations. For example, the language does not allow definition of parametric monitors,

[1] https://bit.ly/3dqUH86

[2] https://bitly.com/31jsAD2

[3] https://bit.ly/3lWjsMn

which could be used to avoid redundancy and to optimize efficiency of property checking. Also, the language for defining join points is limited. In particular, it does not enable a developer to capture read or write field accesses, which could be useful for checking data races, for instance. It is worth noting, however, that we could not find any property in our sample where that feature would be useful.

IV. Study settings

The focus of the study is to assess the challenges of formally specifying API properties for RV. The initial part of the study (RQ1) consisted of building the SIESTA language, as described in Section III, capable of expressing most of the properties from our dataset, a fragment of the PropertyDB dataset [49]. Using the specs in the PropertyDB dataset as ground truth, we assessed the completeness of our proposed language before its adoption in the second part of this study. This activity produced two documents: (i) a language specification document and (ii) a spreadsheet containing example properties specified in SIESTA. The remainder of this section describes the settings of the study related to RQ2 and RQ3.

A. Subjects

We conducted an experiment with 14 CS students taking a course on Software Testing and Debugging. The students received no specific training in logic for this task and, to the best of our knowledge, had no prior experience on it beyond propositional logic and first-order logic, which are superficially covered in a Discrete Math course taken in their junior year.

B. Tasks and Procedures

The study with students was conducted in two stages. In the *first stage*, the students attended a 1h online training session where the course instructor explained the goal of RV, the syntax and semantics of SIESTA, and the individual task they needed to complete. The task assigned to students consisted of (1) selecting five properties listed in a shared spreadsheet, (2) specifying them using SIESTA, and (3) responding to a questionnaire about the task. We used a spreadsheet to enable students to choose disjoint sets of properties to specify and to enable them to share answers and discuss. In the *second stage* of our study, the students attended another 1h online training session where the instructor first explained JavaMOP's syntax and semantics through examples and then explained the task they needed to complete. The session was recorded for offline view. The task assigned to students consisted of (1) comparing the specification written in the first stage with the JavaMOP specification; (2) updating the written specification, if necessary; and (3) answering a questionnaire about their findings. In each of the two tasks, the students had one week to turn in their answers.

C. Materials

Considering the first stage of the study, we made available to the students a spreadsheet, containing all properties from the PropertyDB [49] dataset associated with the packages *java.io*, *java.lang*, and *java.util*. We discarded properties labeled as "suggestions", which are less likely to indicate a problem; only properties with the "error" and "warning" labels were considered. We also released to the students a short language specification document to describe the syntax and semantics of SIESTA through examples (see Section III-B).

Considering the second stage of the study, we released to the students the specifications of the properties they specified in the first part of the study in JavaMOP. Those specs are part of the PropertyDB [49] dataset and were prepared as part of prior work [32]. We also made available the video of the training session explaining JavaMOP itself.

D. Metrics

RQ2 is quantitatively evaluated by adopting the distribution of students' grades as a proxy of their ability to effectively write specs. We measure the discrepancy between the answers provided by students and the specifications the instructors wrote to each property, which were used as ground truth. For the qualitative part of the study (RQ3), we used two questionnaires with Likert scale questions to collect students' opinions towards various aspects of the practice of writing specs. The first questionnaire was applied after the students concluded the task of writing specifications adopting our proposed language (first stage of our study); the second questionnaire was applied after the students were introduced to the formal specifications in JavaMOP (second stage of our study).

E. Replication Package

The replication package—including the dataset of properties from PropertyDB, the language specification document, the applied questionnaires (with students' responses), and the tool prototype—is publicly available at the following link: https://github.com/STAR-RG/SIESTA.

V. Results and Discussion

This section reports on the results of our study.

A. Answering RQ1: Are rich specification languages necessary for RV?

We were able to specify 92 out of the 99 properties from our dataset using SIESTA. Table I shows a breakdown of the number of properties we analysed by category. We found that for six of the cases it is not necessary to specify these properties as associated library code already does. We also found that a total of seven properties could not be specified with the language, but we considered them irrelevant to RV. In what follows, we discuss the rationale for these categories by presenting examples.

TABLE I: Breakdown of number of properties per category.

Category	Total
Supported and relevant	**86**
Supported, but irrelevant	**6**
Unsupported, but irrelevant	**7**
Σ	99

***Supported and relevant* properties.** We were able to specify the majority of the properties using the language presented in Section III. The examples from Section III-D are representative of the properties that SIESTA can specify. Overall, we found that SIESTA can specify a large number of properties with different characteristics.

***Supported, but irrelevant* properties.** A number of properties can also be specified with our language, but they do not contribute to RV as the Java API includes checks for these cases. Figure 6 shows a code snippet to illustrate this scenario.

```
import java.io.*;
public class Sample {
 public static void main(String[] args) throws Exception {
  InputStream is = new FileInputStream("helloworld.txt");
  InputStreamReader isr = new InputStreamReader(is);
  isr.mark(0); // <-- throws exception; mark not supported
}}
```

Fig. 6: Example of *Supported, but irrelevant* property.

The corresponding property in PropertyDB for this case is called Reader_MarkReset [52]. It states that the `mark` and `reset` methods cannot be called on certain subtypes of the `InputStream` class. The API implementation already checks (defensively) for the property violation and RV is unable to detect the issue earlier. Compiling this code with the Java compiler—without any additional instrumentation for RV—and running it raises an exception. Compiling the JavaMOP spec would result in a duplicate (unecessary) check. Note that this is a limitation of the dataset as opposed to a limitation of this experiment. The dataset we used does not distinguish between the parts of the documentation that can benefit from specification and the parts that cannot (as this one). However, this is irrelevant to the central goal of our study, which is evaluating the ability of developers to specify properties regardless of their usefulness.

***Unsupported, but irrelevant* properties.** We found that some properties in the dataset refer to a "program termination" event, which SIESTA does not support, as we found no practical use for it. For instance, consider the property `Console_FillZeroPassword` from PropertyDB [9]. This property states that after calling the `readPassword()` method on a `java.io.Console` object (to read a password as an array of bytes), the application should manually clean the store of the password after processing it. The rationale of this property is to minimize the lifetime of security-sensitive data in memory. As one can expect, there is no way to enforce the number of operations allowed between the call to `readPassword` and the recommended cleaning operation. Therefore, in principle, the clean check could be done as the last event emitted by a running program, which is what JavaMOP does. Our interpretation is that checking the property at that point is meaningless. Similar cases as the one above occur on objects that implement the `ObjectInput` and `ObjectOutput` interfaces. The documentation shows that instances of these types must be closed if opened, to release any resources associated with the stream. Finally, four other properties are associated with subtypes of non-serializable classes that want to allow serialization, as well as `Collection` and `Map` implementations. Such properties require the presence of one (or more) constructors following certain rules. There is no support for checking that using SIESTA, but these properties could be statically checked.

Summary: *Considering the 99 properties from PropertyDB [49] that we selected, we manually-specified and cross-checked 92% of them using our minimalist spec language.*

B. Answering RQ2: Can developers with no training in logic express properties about an API by reading its documentation?

To address this question, we measured the discrepancy between the specs written by the students and the ground-truth, i.e., specs written by the instructors. The specs of students and instructors matched perfectly (modulo variable renaming) in 40% of the cases. Figure 7 shows the distribution of grades (0-10 scale), which we used as proxy of the ability of students to effectively write specs. In the histogram to the left, the x-axis displays the grades for each spec, whereas the y-axis reports the frequency that a given grade was given by the instructors.

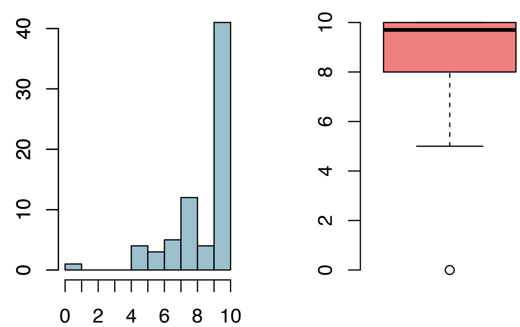

Fig. 7: Distribution of grades of students for the task of writing specs. The left plot shows a histogram of the scores associated with each task whereas the right plot shows a box plot aggregating results.

The average grade attributed to each task was 8.7 with a median of 9.7 (represented by the notch in the box plot). A total of 64% of the specifications written by the students (45 out of 70) received a grade $\geq$ 9. A grade of 9 or more was awarded for the cases where (i) the property was correctly specified but contained small syntactic mistakes; or (ii) the property was over-specified, i.e., the specification was more restrictive than necessary. A grade of 10 was awarded only in the cases where the specification written by the students matched precisely the one written by the instructors (modulo variable renaming). There was one grade zero given to a student who classified the property as *Unsupported, but irrelevant* and did not write any specification but, in reality, the property is *Supported and relevant*.

A total of 24% of the specifications written by the students (17 out of 70) received a grade within the range $[7, 9[$. Grades within such range were assigned for the cases where (i) the property was only partially, but correctly specified (i.e., it missed at least one step to be able to fully enforce the property); or (ii) the specification, although correct, covers only one of the many methods impacted by the property.

Figure 8 shows one example where the developer made only one small mistake and was penalized with 2/100 points for his mistake. A violation should be signaled if one or more calls to `close` is found in the history of messages to the object `is` at the point of a call to any methods in the `@Before` list. Instead of this, the student conditioned the violation to exactly one call (line 9 uses `==1` instead of `>1`), i.e., multiple calls to `close` would not result in a violation.

```
1  @Before("java.io.BufferedInputStream.read")
2  @Before("java.io.BufferedInputStream.available")
3  @Before("java.io.BufferedInputStream.reset")
4  @Before("java.io.BufferedInputStream.skip")
5  void vioManipulateAfterClose(BufferedInputStream is) {
6    List<String> history = is.history();
7    String[] interesting = new String[]{"close"};
8    // should be >1 instead of ==1
9    if (filter(history, interesting).size()==1)
10     Log.violation("Manipulate after close");
11 }
```

Fig. 8: Student specification of `BufferedInputStream` ManipulateAfterClose property [1].

Summary: *Students were successful in this task. Out of the 70 specs they answered, 64% received a grade equal or above 9. The specifications written by students matched the ground truth perfectly (modulo renaming) in 40% of the cases.*

C. Answering RQ3: What are the perceptions of developers about the practice of writing specs?

We answer this research question by interpreting the answers we obtained to the two questionnaires that the students answered during the course of our experiment (see Section IV). Each of the following sections focuses on the discussion of one of those questionnaires.

1) Questionnaire about writing specs in SIESTA: Recall that during the first stage of our study, the students had to specify five properties of the Java API. For that part, the questionnaire focused on capturing students' perceptions about writing specs. Students had no prior experience in other specification languages.

Figure 9 displays a diverging stacked bar graph for each question answered by the students. The y-axis of the plot shows the questions made to students whereas the x-axis shows the percentage of responses, to a given question, that fall in a given category (e.g., "strongly disagree", "neutral", etc.). More precisely, the length of a bar shows the percentage of answers in a given category. Neutral responses are centered around the zero vertical line. Positive responses appear on the right side of the zero line and all negative responses appear on the left side. This representation helps us visualize the amount of positive and negative responses.

Q1.1 is a sanity-check to evaluate that the students understood the task correctly. All students answered this question either in a neutral or in a positive way, meaning that the requested activity was clear to them. **Q1.2** asked how confident the students were about their answers, i.e., how confident they were that their specifications were correct. For this question, $\approx$36% of the students were neutral whereas $\approx$64% were confident about their specifications. **Q1.3** asked the students if the spreadsheet with examples, shared as supporting material, was useful for preparing their answers. We found that $\approx$79% of the students answered this question positively. This was the question that received the highest number (7) of "strongly agree" answers. This result confirmed our expectation that students can learn a great deal from examples of similar properties, and even from unrelated examples, as reported by the students. **Q1.4** relates to students' satisfaction while conducting the requested task. While the vast majority of the answers lies on the neutral and positive side of the plot, one student answered this question with "disagree". We found the answers to this question could be rather irrational and preferred not asking the reasons for dissatisfaction. **Q1.5** collected the students' perception on the quality of the text describing the property. Opinions diverged for this question: half of the students stated that the description was enough to write the specification whereas the other half considered the documentation insufficient. We realized that this part of the experiment could have been improved. We did find some properties that could benefit from context or even code examples to elucidate intent. This also highlights the importance of good API documentation and methods to identify and aggregate parts of the documentation relevant to specify properties.

2) Questionnaire about the comparison between reading JavaMOP vs. SIESTA specs: In the second stage of our study, the students were introduced to JavaMOP and were asked to compare the specifications written by themselves against the corresponding JavaMOP specification. The students were also given the opportunity to update their specifications, if necessary. The answers collected with the questionnaire are displayed in the diverging stacked bar graph on Figure 10. We also included an open-ended final question asking the students whether they missed important features in the minimalist language after being introduced to JavaMOP. We elaborate on each question in the following.

Q2.1 asked the students if the effort required to understand a JavaMOP specification and a specification written in SIESTA were similar. We found that the majority of the students disagreed or strongly disagreed ($\approx$71%) with the statement. While Q2.1 tells us that the students do not consider the two specifications (formal and minimalist) to be equivalent in terms of effort, it does not indicate preference. This is addressed in questions **Q2.2** and **Q2.3**. Notice that these questions are similar. We intentionally replaced the subjects in the sentences to check if the answers provided by the participants were consistent. We confirmed consistency of the answers. Overall, $\approx$29% of the students were neutral while $\approx$50% agreed or strongly agreed that the minimalist specification was easier to read/understand when compared with the JavaMOP specifications. It is also worth noting that $\approx$21% considered JavaMOP easier to read/understand. This is endorsed by the following comments from the students: *"In my opinion, once you learn how the JavaMOP specification works, it is easier to read/understand (the spec) because of the feature to name the checks which helps the programmer to*

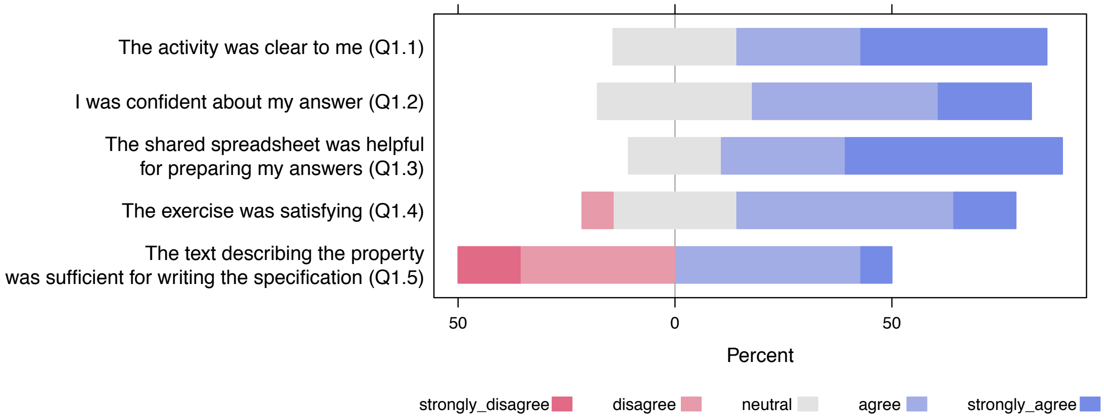

Fig. 9: Developers' perceptions while writing specifications using SIESTA

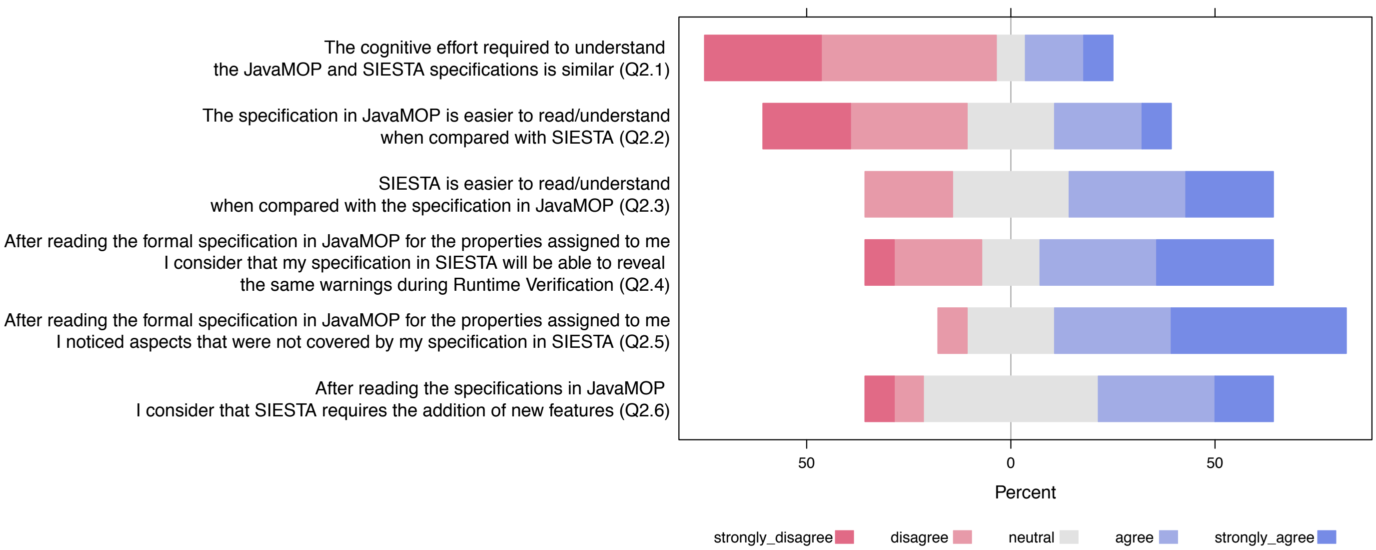

Fig. 10: Developers' perceptions while comparing the specifications written by themselves against the formal specifications from JavaMOP.

have an idea about what the code does" from *student #3*; and *"JavaMOP is much more expressive and it enables the use of CFG, LTL, ERE, etc, for matching some scenarios."* from *student #7*. The answers to **Q2.4** indicate that, after reading the JavaMOP specifications, ≈57% of the students remained confident that their specifications were as good as JavaMOP's, i.e., students believe that RV with either spec would reveal the same set of violations. Note that questions Q2.4 and Q2.5 are similar. Again, our intention with this redundancy was to check consistency in the answers. **Q2.5** asked whether the students noticed behaviors that were not covered in their specifications after reading the JavaMOP specifications. Given that most students answered, in Q2.4, that their specs could capture the behaviors specified in the JavaMOP specs, we expected that only few students would answer positively to Q2.5. However, we found that 71% of the students agreed or strongly agreed that they found, after reading JavaMOP's specs, that their specifications missed some behavior.

To understand the extent of the modifications performed by the students, we manually analyzed the new versions of the 33 modified specs, which correspond to 47.1% of the total specs. We classified modifications into three categories: *minor*, *moderate*, and *major* changes. Changes were classified as major either because the student had classified the property as unsupported and then realized that it could be specified, or because the student performed severe changes. Only five modifications were classified as major. The majority of the modifications (20) were minor. We classified changes as minor if they consisted of variable renaming, small syntactic changes, or if the specification was modified only for the purpose of covering additional methods (e.g., the spec covered only the `add` method, and now it also covers the `addAll` method). The remaining eight cases were classified as moderate.

When we asked if SIESTA required the addition of new features (**Q2.6**), the percentage of agree or strongly agree was ≈43%. This might indicate that the students' perception of missing coverage is not associated with a lack of expressiveness of the proposed minimalist language, but rather with other aspects (e.g., unclear/incomplete textual description of the property to be specified).From the group of students who answered that SIESTA required new features, the following suggestions have been made: *student #13* suggested *"the possibility of defining events and telling the order of those events as an faulty scenario"*, whereas *student #7* stressed that *"the lightweight language does not support evaluation of future events"*. The majority of the students (≈57%) answered this question in either a neutral way or considered that SIESTA does not require the addition of new features: *student #11* stated *"no additions are required"*; and *student #8* commented *"I believe I was able to translate from JavaMOP to the minimalist language without requiring additional features"*.

Summary: *Overall, the students expressed satisfaction in writing specifications, even though the documentation is not always precise enough for the task, and expressed preference in using lightweight imperative specifications.*

VI. Lessons Learned

The main lessons we learned from this study are as follows:

⋆ The intrinsic complexity of writing specs is largely overstated. No participant of the study reported that the activity was time demanding. Consequently, engaging developers in the task of creating specs is beneficial;

⋆ API documentation is a great source of information to write specifications. Despite the fact that Natural Language is ambiguous, we found that carefully-written documentation, when exists, was central to writing good specs, especially when developers were not acquainted with the related code. Note that documentation is essential to enable usage of APIs.

⋆ Leveraging the crowd is important for learning specifications. We observed that (i) there are patterns in the API properties and (ii) participants can leverage those patterns to write specs. For example, a number of properties refer to `Iterator` objects, that can be obtained from different `Collection` implementations. Moreover, other properties related to Collections deal with ordered data structures that require their elements to implement the `Comparable` interface. We also found property patterns in the `java.io` package, specially regarding resource manipulation;

⋆ Engagement of developers should be considered as an educational investment as opposed to a deterrent of productivity. The participants of the study spontaneously reported that they learned from reading the API documentation and specifying corresponding properties.

VII. Threats to Validity

A threat related to RQ1 is the set of properties that were selected for establishing the ground truth. We focused on properties related to popular packages from the Java API. However, it remains as future work to evaluate whether properties from other packages would require enhancing our specification language with more features. Nonetheless, we do not aim for generality, as described in Section III-A. To answer RQ2 we measured the discrepancy between the specifications provided by students and those written by the instructors. The quality of our ground truth is tied to the level of expertise of the instructors and this might have impacted the specifications considered correct/wrong for the conduction of our experiment. To minimize this threat, the specifications considered as ground truth were prepared independently by two instructors that interacted at the end of this task to reach agreement. They were also reviewed by another author. Another potential threat related to RQ2 is the fact that we use the students' grades as a proxy of their ability to effectively write specifications. The students' grades could have been impacted by the selection of properties that are easier/harder to specify. To address this threat we instructed the students to choose each of the five properties from different classes. While this does not guarantee an equal level of difficulty assigned to each student, it maximizes our chances of a balanced assignment. We adopted this method for assigning properties to students primarily to let them chose their preference as to incentivize engagement. To note that we did not find properties in the dataset that are trivial or very challenging to specify.

VIII. Related Work

RV received intense attention over the last two decades. In what follows, we discuss work that is closely related to ours.

Runtime Verification. As mentioned, many RV techniques and tools were proposed in the two decades after the first papers on RV [11], [17], [18] came out. RV is suitable to find bugs when some given specified properties are violated. It improved in recent years [7], [14], [23], [39], [45], [62], to the extent where there are now proposals for using RV even to find bugs during software development and testing [33]–[37]. For instance, Legunsen et al. [33], [35], [36] recently showed that performing RV during test execution is scalable and can detect hundreds of bugs that existing tests written by developers did not find. They found these bugs as the result of a large-scale study [35], [36], involving hundreds of open-source projects, where they used JavaMOP [20] to specify and monitor parts of the standard Java library API [32], [48]. JavaMOP [20] allows one to express parametric properties and it also enables the dynamically monitoring of such properties in one test run. In addition to JavaMOP [20], we have several other tools for RV, such as jMonitor [26], JPaX [18], MarQ [55], MOPBox [5], Mufin [10], Ruler [4], and TraceMatches [6]. However, unlike the approach we present, none of these tools offers a simplistic way to perform runtime verification, with a minimalist specification language to enable behavioral property specification in a quite straightforward fashion.

Mining specs from documentation. @TCOMMENT [60] is an approach to testing Javadoc comments, specifically method properties about null values and related exceptions. The approach consists of two components: the first takes as input source files for a Java project and automatically analyzes the English text in Javadoc comments to infer a set of likely properties for a method in the files; the second component generates random tests for these methods, checks the inferred properties, and reports inconsistencies. Our approach also consists on analyzing the documentation from the original Javadoc comments of APIs, but we need to manually specify the behavioral properties of interest. In addition, we can write richer behavioral properties than those of @TCOMMENT, which focus on checking null values and exceptions.

Zhai et al. [63] present a technique to build models for Java API functions by analysing the documentation. Their models are simpler implementations in Java compared to the original ones and hence easier to analyse. More importantly, they provide the same functionalities as the original functions. They argue that API documentation, like Javadoc and .NET documentation, usually contains information about the library functions, such as the behaviour and exceptions they may throw. Thus, it is feasible to generate models for library functions from such documentation. Unlike their approach for models of Java API functions, we use our minimalist specification language to derive behavioral properties from those Java API functions. However, their generated models could be useful and used as input to our specification language for behavioral property specification.

Sun et al. [59] propose CrowdSpec, an approach that leverages crowd intelligence to produce or improve specifications. They use human annotators to interpret whether automatically inferred specifications from input traces conform to the documentation. While the works are complementary, and we also find that leveraging the crowd is important, there are some distinctions in the way in which the studies have been conducted. They performed a screening process using Amazon Mechanical Turk, while we relied on students, which might not possess the same level of technical competence. Moreover, in our study, the participants produced the specification from the documentation, while in their study, the participants received an inferred specification to improve.

Contracts. As with RV, contracts are a popular tool for specifying and checking the functional behavior of software during runtime [43], [57]. A contract precisely and unambiguously specifies what must be true when a method is called (preconditions) and what must be true when it returns (postconditions) [16]. Similar to standard RV tools, languages such as JML [30], [54] and Code Contracts [13] might require training. Thus, it might become hard to read and write, and hence, often used sparingly, as reported by empirical studies on contract usage [8], [12], [47], [58]. Such studies show that developers only use simple, short, and straightforward contracts. This implies that little of a rich specification language is, in fact, used in practice by developers. Hence, a rich language like JML requires learning new features and, along with new supporting syntax, could lead to unexpected interactions [15], [40]. For example JML visibility rules for contracts [31], [53] do not follow the Java semantics, thus becoming a source of problems for beginners.

On the other hand, one point in favor of contracts is that there is evidence programmers are more likely to use contracts in languages that natively support them [8], such as Microsoft's Code Contracts [13]. However, any contract expressed by these built-in contract languages (or even the non-builtin ones, such as JML) may be useful for programmers (internal documentation), but it does not meet the needs of other readers (separate/external documentation), such as third-party libraries [29], [46]. To use those libraries, a programmer should not need to look in the code to find out how to use it.

The decision to keep a minimalist specification language avoids any semantic complications due to additional language constructs and ultimately lack of adoption. Also, even though the language is not defining contracts, building the language on top of Java also leverages the existing knowledge from developers. Moreover, our minimalist specification language does care about API specification. That is, similarly to JavaMOP, our language does provide the behavioral specifications properly separated from the client code and instrumented accordingly with the monitors for runtime verification.

Finally, one may argue that typestate protocol specifications could be written or encoded together with pre- and postconditions to properly specify particular method call sequences as well as the usage constraints of each method. In this direction, Kim et al. [28] extend JML syntax to include typestate protocol specification features. Although we do not support specific constructs for pre- and postconditions specifications along with explicit typestate protocol specifications, their notion of typestate JML is not actually implemented and, therefore, a programmer could not be benefited from RV of such specifications. Moreover, regardless of tool support, a typestate JML developer should learn, besides all the standard JML features, the additional ones for typestate specification support. This scenario is even less attractive for beginners.

IX. Conclusion

An important obstacle for adopting RV is having property specifications to monitor during runtime. Specification languages are typically rich in features and this may hold back developers from adopting RV. This paper reports on a study to assess the extent to which this can be simplified. We do so by conducting a study with students with superficial knowledge in logic using a simplistic language to express properties. Our results show that even using such a language, we are able to specify the majority of the properties we have selected from the existing PropertyDB dataset [49]. Moreover, students were also able to successfully write specifications that closely match the ground truth, and expressed satisfaction on performing this task. The main message from this work is that writing specs is not much harder than writing code. In fact, they are quite similar in our specification language. We intend to conduct further evaluations to better understand the tradeoffs between rich specifications and learning curves, as well as investigating it with senior developers. Moreover, we intend to explore crowd learning for writing and checking specifications.

Acknowledgments

This research was partially funded by INES 2.0, FACEPE grants PRONEX APQ 0388-1.03/14, APQ-0399-1.03/17, APQ-0751-1.03/14, and APQ-0570-1.03/14, CAPES grant 88887.136410/2017-00, and CNPq grants 465614/2014-0, 309032/2019-9, 406308/2016-0.

References

[1] "BufferedInputStream.close() API property documentation," 2020, shorturl.at/dpsuB.

[2] "Collections.synchronizedCollection() API property documentation," 2020, shorturl.at/fwDEN.

[3] "Apache Commons Lang," 2019, https://commons.apache.org/proper/commons-lang/.

[4] H. Barringer, D. Rydeheard, and K. Havelund, "Rule systems for run-time monitoring: From eagle to ruler," in *Runtime Verification*, O. Sokolsky and S. Taşıran, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 111–125.

[5] E. Bodden, "Mopbox: A library approach to runtime verification," in *Runtime Verification*, S. Khurshid and K. Sen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 365–369.

[6] E. Bodden, L. Hendren, P. Lam, O. Lhoták, and N. A. Naeem, "Collaborative runtime verification with tracematches," in *Proceedings of the 7th International Conference on Runtime Verification*, ser. RV'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 22–37.

[7] E. Bodden, P. Lam, and L. Hendren, "Finding programming errors earlier by evaluating runtime monitors ahead-of-time," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: Association for Computing Machinery, 2008, p. 36–47.

[8] P. Chalin, *Are Practitioners Writing Contracts?* Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 100–113.

[9] "Console_FillZeroPassword property," 2020, shorturl.at/fkG26.

[10] N. Decker, J. Harder, T. Scheffel, M. Schmitz, and D. Thoma, "Runtime monitoring with union-find structures," in *Tools and Algorithms for the Construction and Analysis of Systems*, M. Chechik and J.-F. Raskin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 868–884.

[11] U. Erlingsson and F. B. Schneider, "Irm enforcement of java stack inspection," in *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, ser. SP '00. USA: IEEE Computer Society, 2000, p. 246.

[12] H. C. Estler, C. A. Furia, M. Nordio, M. Piccioni, and B. Meyer, "Contracts in practice," in *FM 2014: Formal Methods*, C. Jones, P. Pihlajasaari, and J. Sun, Eds. Cham: Springer International Publishing, 2014, pp. 230–246.

[13] M. Fähndrich, M. Barnett, and F. Logozzo, "Embedded contract languages," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 2103–2110.

[14] V. Forejt, M. Kwiatkowska, D. Parker, H. Qu, and M. Ujma, "Incremental runtime verification of probabilistic systems," in *Runtime Verification*, S. Qadeer and S. Tasiran, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 314–319.

[15] W. H. Harrison, H. L. Ossher, and P. L. Tarr, "Asymmetrically vs. symmetrically organized paradigms for software composition," Research Report RC22685, IBM Thomas J. Watson Research, Tech. Rep., 2002.

[16] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson, "Behavioral interface specification languages," *ACM Comput. Surv.*, vol. 44, no. 3, Jun. 2012.

[17] K. Havelund and G. Rosu, "Monitoring programs using rewriting," in *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ser. ASE '01. USA: IEEE Computer Society, 2001, p. 135.

[18] K. Havelund and G. Roşu, "Monitoring java programs with java pathexplorer," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, pp. 200 – 217, 2001, in RV'2001, Runtime Verification.

[19] S. Hussein, P. Meredith, and G. Roşlu, "Security-policy monitoring and enforcement with javamop," in *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, ser. PLAS '12. New York, NY, USA: Association for Computing Machinery, 2012.

[20] "JavaMOP4," 2015, http://fsl.cs.illinois.edu/index.php/JavaMOP4.

[21] "JavaMOP4 Syntax," 2015, http://fsl.cs.illinois.edu/index.php/JavaMOP4_Syntax.

[22] "The state of developer ecosystem 2020 - jetbrains," 2020, https://www.jetbrains.com/lp/devecosystem-2020/.

[23] D. Jin, P. O. Meredith, D. Griffith, and G. Rosu, "Garbage collection for monitoring parametric properties," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 415–424.

[24] D. Jin, P. O. Meredith, C. Lee, and G. Roşu, "Javamop: Efficient parametric runtime monitoring framework," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 1427–1430.

[25] D. Jin, P. O. Meredith, and G. Roşu, "Scalable parametric runtime monitoring," Computer Science Dept., UIUC, Tech. Rep., 2012.

[26] M. Karaorman and J. Freeman, "jmonitor: Java runtime event specification and monitoring library," *Electronic Notes in Theoretical Computer Science*, vol. 113, pp. 181 – 200, 2005, proceedings of the Fourth Workshop on Runtime Verification (RV 2004).

[27] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ser. ECOOP '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 327–353.

[28] T. Kim, K. Bierhoff, J. Aldrich, and S. Kang, "Typestate protocol specification in jml," in *Proceedings of the 8th International Workshop on Specification and Verification of Component-Based Systems*, ser. SAVCBS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 11–18.

[29] G. T. Leavens, "The future of library specification," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 211–216.

[30] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, "How the design of JML accommodates both runtime assertion checking and formal verification," *Science of Computer Programming*, vol. 55, no. 1-3, pp. 185–208, Mar. 2005.

[31] G. T. Leavens and P. Muller, "Information hiding and visibility in interface specifications," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. USA: IEEE Computer Society, 2007, p. 385–395.

[32] C. Lee, D. Jin, P. O. Meredith, and G. Roşu, "Towards categorizing and formalizing the JDK API," Computer Science Dept., UIUC, Tech. Rep., 2012.

[33] O. Legunsen, Y. Zhang, M. Hadzi-Tanovic, G. Rosu, and D. Marinov, "Techniques for evolution-aware runtime verification," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 300–311.

[34] O. Legunsen, "Evolution-Aware Runtime Verification," Ph.D. dissertation, University of Illinois at Urbana-Champaign, USA, 2019.

[35] O. Legunsen, N. A. Awar, X. Xu, W. U. Hassan, G. Roşu, and D. Marinov, "How effective are existing Java API specifications for finding bugs during runtime verification?" *Autom. Softw. Eng.*, vol. 26, no. 4, 2019.

[36] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov, "How good are the specs? a study of the bug-finding effectiveness of existing java api specifications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 602–613.

[37] O. Legunsen, D. Marinov, and G. Roşu, "Evolution-aware monitoring-oriented programming," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. IEEE Press, 2015, p. 615–618.

[38] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293 – 303, 2009, the 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[39] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. ŞerbănuŢă, and G. Roşu, "Rv-monitor: Efficient parametric runtime verification with simultaneous properties," in *Runtime Verification*, B. Bonakdarpour and S. A. Smolka, Eds. Cham: Springer International Publishing, 2014, pp. 285–300.

[40] B. J. MacLennan, *Principles of Programming Languages: Design, Evaluation, and Implementation (2Nd Ed.)*. Austin, TX, USA: Holt, Rinehart & Winston, 1986.

[41] P. O. Meredith, D. Jin, F. Chen, and G. Rosu, "Efficient monitoring of parametric context-free patterns," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. USA: IEEE Computer Society, 2008, p. 148–157.

[42] P. Meredith and G. Rosu, "Efficient parametric runtime verification with deterministic string rewriting," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13. IEEE Press, 2013, p. 70–80.

[43] B. Meyer, "Applying "Design by Contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[44] B. Miranda, I. Lima, O. Legunsen, and M. d'Amorim, "Prioritizing runtime verification violations," in *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020, pp. 297–308.
[45] S. Navabpour, C. W. W. Wu, B. Bonakdarpour, and S. Fischmeister, "Efficient techniques for near-optimal instrumentation in time-triggered runtime verification," in *Proceedings of the Second International Conference on Runtime Verification*, ser. RV'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 208–222.
[46] D. Parnas, *Precise Documentation: The Key to Better Software*. Springer Berlin Heidelberg, 2011, pp. 125–148.
[47] N. Polikarpova, I. Ciupa, and B. Meyer, "A comparative study of programmer-written and automatically inferred contracts," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 93–104.
[48] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. USA: IEEE Computer Society, 2009, p. 371–382.
[49] "FSL Specification Database," 2016, https://runtimeverification.com/monitor/propertydb.
[50] R. Purandare, M. B. Dwyer, and S. Elbaum, "Monitor optimization via stutter-equivalent loop transformation," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 270–285.
[51] ——, "Optimizing monitoring of finite state properties through monitor compaction," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 280–290.
[52] "Reader_MarkReset property," 2020, shorturl.at/zGSV0.
[53] H. Rebêlo and G. T. Leavens, "Enforcing information hiding in interface specifications: A client-aware checking approach," in *Companion Proceedings of the 14th International Conference on Modularity*, ser. MODULARITY Companion 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 47–51.
[54] H. Rebêlo, G. T. Leavens, M. Bagherzadeh, H. Rajan, R. Lima, D. M. Zimmerman, M. Cornélio, and T. Thüm, "Aspectjml: Modular specification and runtime checking for crosscutting contracts," in *Proceedings of the 13th International Conference on Modularity*, ser. MODULARITY '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 157–168.
[55] G. Reger, H. C. Cruz, and D. Rydeheard, "Marq: Monitoring at runtime with qea," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 596–610.
[56] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2013.
[57] D. S. Rosenblum, "Towards a method of programming with assertions," in *Proceedings of the 14th International Conference on Software Engineering*, ser. ICSE '92. New York, NY, USA: Association for Computing Machinery, 1992, p. 92–104.
[58] T. W. Schiller, K. Donohue, F. Coward, and M. D. Ernst, "Case studies and tools for contract specifications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 596–607.
[59] P. Sun, C. Brown, I. Beschastnikh, and K. T. Stolee, "Mining specifications from documentation using a crowd," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 275–286.
[60] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 260–269.
[61] "TIOBE Index for October 2020," 2020, https://www.tiobe.com/tiobe-index/.
[62] C. W. W. Wu, D. Kumar, B. Bonakdarpour, and S. Fischmeister, "Reducing monitoring overhead by integrating event- and time-triggered techniques," in *Runtime Verification*, A. Legay and S. Bensalem, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 304–321.
[63] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin, "Automatic model generation from documentation for java api functions," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 380–391.