Technical Report http://hdl.handle.net/2142/30006, March 2012

Towards Categorizing and Formalizing the JDK API

Choonghwan Lee
University of Illinois
Urbana, IL 61801, U.S.A.
clee83@illinois.edu

Dongyun Jin
University of Illinois
Urbana, IL 61801, U.S.A.
djin3@illinois.edu

Patrick O'Neil Meredith
University of Illinois
Urbana, IL 61801, U.S.A.
pmeredit@illinois.edu

Grigore Roşu
University of Illinois
Urbana, IL 61801, U.S.A.
grosu@illinois.edu

ABSTRACT

Formal specification of correct library usage is extremely useful, both for software developers and for the formal analysis tools they use, such as model checkers or runtime monitoring systems. Unfortunately, the process of creating formal specifications is time consuming, and, for the most part, even the libraries in greatest use, such as the Java Development Kit (JDK) standard library, are left wholly without formal specification. This paper presents a tool-supported approach to help writing formal specifications for Java libraries and creating documentation augmented with highlighting and formal specifications. The presented approach has been applied to systematically and *completely* formalize the runtime properties of three core and commonly used packages of the JDK API, namely `java.io`, `java.lang` and `java.util`, yielding 137 formal specifications. Indirectly, this paper also brings empirical evidence that *parametric specifications* may be sufficiently powerful to express virtually all desirable runtime properties of the JDK API, and that its informal documentation *can be formalized*.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing/Debugging

Keywords

Java, benchmark, formal specification, software engineering

1. INTRODUCTION

A *formal specification* defines behaviors that systems or parts of systems must or are recommended to obey. An example of a formal specification is a linear temporal logic (LTL) formula, $open \rightarrow \Diamond close$, where *open* and *close* represent creating a `FileOutputStream` object and calling `FileOutputStream.close()`, respectively. This specification says that an opened `FileOutputStream` object should be eventually closed. In spite of its simplicity, it is effective in finding a common error: forgetting to invoke `close()` on a `FileOutputStream` object of local scope in `catch` blocks.[1]

There is no doubt that formal specifications are very useful, when available. They can be used, for example, for finding actual or potential errors in programs, such as illegal usage of the API like above, security vulnerabilities, performance degradation, and so on. Despite their usefulness, the reality is that formal specifications rarely exist in practice. We do not attempt to elucidate why this is the case, but we believe that the following could be some valid reasons:

1. They are *not easy to produce*, often requiring a deep understanding of or even a duplication effort of the implementation. This is particularly problematic when the formal specifications are intended to capture the full functional correctness of the implementation.

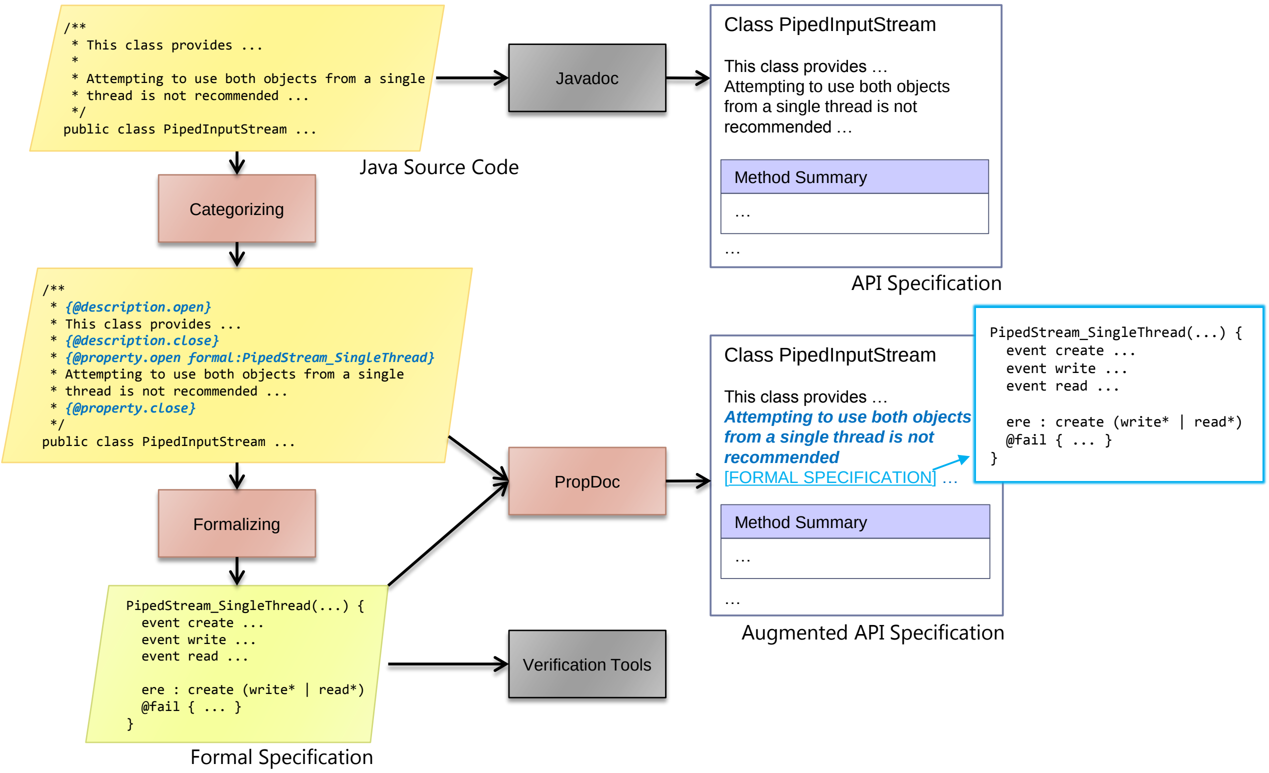
2. There is *no clear formalism* that we should use for writing specifications. We want a formalism which is both expressive and practical, so that one can express all properties of interest and at the same time be able to immediately use them for scalable software analysis.

3. There are *no tools* to systematically help us digest large bodies of informal documentation, to categorize the text in a way that reduces the likelihood to miss properties and that offers a measure of the effectiveness and the coverage of the formalization effort.

In this paper we propose an approach to producing formal specifications for the JDK API by addressing the above three aspects as follows. First, we compromise by not attempting to produce functional correctness specifications. Instead, we only focus on API usage correctness properties, more precisely on the subset of safety properties monitorable at runtime. Second, we choose a specification formalism that has been shown to be expressive and efficiently monitorable by the runtime verification community over the last decade, namely that of parametric specifications, supported in one form or another by several runtime monitoring systems, e.g.,

[1] `finalize()`, invoked by the garbage collector, eventually calls `close()` to release the resources, but such delayed action can cause file corruption—it occurs because the modification is not visible to other file-handling objects or processes until the buffer is flushed by `close()` or `flush()`—and file operation failure—some file systems disallow moving or deleting a file when the file is opened.

.



```
/**
 * This class provides ...
 *
 * Attempting to use both objects from a single
 * thread is not recommended ...
 */
public class PipedInputStream ...
```

```
/**
 * {@description.open}
 * This class provides ...
 * {@description.close}
 * {@property.open formal:PipedStream_SingleThread}
 * Attempting to use both objects from a single
 * thread is not recommended ...
 * {@property.close}
 */
public class PipedInputStream ...
```

```
PipedStream_SingleThread(...) {
  event create ...
  event write ...
  event read ...

  ere : create (write* | read*)
  @fail { ... }
}
```

```
PipedStream_SingleThread(...) {
  event create ...
  event write ...
  event read ...

  ere : create (write* | read*)
  @fail { ... }
}
```

Figure 1: Categorizing and Formalizing the JDK API using PropDoc.

Tracematches [2], EAGLE [4], RuleR [5], and JavaMOP [14] among others. Third, we have developed a tool for generating API documentation augmented with text categorization, formal specifications, and coverage statistics.

A *parametric specification* (see, e.g., [14] for a recent and formal definition) is a type of formal specification where each parameter is bound to a concrete object instance at runtime. For example, a parametric specification can express the following behavior: for each binding of parameters ⟨PipedInputStream, PipedOutputStream, Thread⟩ to concrete objects $\langle i, o, t\rangle$, if i and o are connected then t cannot access both i and o (because that may deadlock t). In contrast, a non-parametric specification cannot capture the object bindings, forcing the pattern to be globally obeyed. For example, accessing an unconnected pair of a PipedInputStream object and a PipedOutputStream object, which may happen from the same thread or not, would yield a false warning. A parametric property does not have this problem, since each binding is considered separately—for example, if i_1, o_1 and respectively i_2, o_2 are connected, behaviors where t_1 accesses i_1, o_2 and t_2 accesses i_2, o_1 are still permitted.

Parametric specifications can bind zero, one or more parameters. They directly generalize *typestates* [18], since a typestate is a particular parametric specification with only one parameter. Typestates can capture properties referring to individual object instances, but they are inadequate for expressing properties referring to two or more related object instances. For example, the property above refers to three different object instances and there is no way to flatten it into a typestate referring to only one of them. We have found that many properties in the JDK API cannot be expressed with typestates, which is why we adopted the more general formalism of parametric specifications in this paper.

We have investigated a series of automatic specification mining approaches, including [3, 10, 20, 9], and have even developed a new one fit to our specific purpose [13]. Unfortunately, our experience with automatic specification mining techniques and tools was not very positive: in general they tend to be hard to use, require significant user involvement, yield overfit or overabstract specifications, and support only very particular properties (e.g., alternating ones of the form $(a\, b)^*$, etc.). Since the formal specifications resulting from this project are made publicly available and are intended to be used by formal analysis tool, their quality is of crucial importance. Consequently, we decided to manually formalize the specifications. We thoroughly inspected the documentation of three packages and wrote a formal specification for each text block that implies a desired or undesired runtime behavior. Given that mature libraries have good documentation, our approach is likely to achieve good coverage.

Figure 1 gives an overview of our approach and the PropDoc tool. The informal Java source code documentation is manually augmented with new tags, which allow to categorize the text and associate formal specifications to it. PropDoc then generates a more informative HTML documentation (see http://fsl.cs.uiuc.edu/annotated-java/ for the latest one), with highlighted text and in-place links to formal specifications. The user can hide the additional information with the click of a button (top-right corner), thus flipping between PropDoc and JavaDoc output.

Using PropDoc, we categorized all the documentation of the widely used JDK[15] packages java.io, java.lang and java.util, and formalized all their documented runtime properties. This way, we produced a total of 137 parametric specifications. We then extensively evaluated them using the JavaMOP [14] runtime verification system against the DaCapo [6] and the SPECjvm [17] benchmark suites. Monitoring each of 137 specifications, we found several bugs and design flaws from open source, real world applications. For example, in both suites, many serializable classes do not have a version number declared; this can lead to improper deserialization of objects. In DaCapo, the h2 benchmark tries to register a shutdown hook thread during its shutdown process, while jython does not properly synchronize a thread-safe list.

Contributions This paper's contributions are as follows:

- PropDoc, a publicly available tool for generating API documentation augmented with text categorization, formal specifications, and coverage statistics;
- The only comprehensive set of formal specifications for three widely used packages of the JDK API. These properties also serve as an extensive benchmark suite for runtime monitoring systems, larger by more than an order of magnitude than the previous ones.

Both the PropDoc tool and the formal specifications are available at http://fsl.cs.uiuc.edu/annotated-java/.

The rest of paper is organized as follows. Section 2 highlights our approach to formalizing the JDK API, monitoring its formal specifications, and generating more informative documentation. Section 3 explains documentation categorizing and formalizing in detail, and Section 4 shows a few example specifications. Sections 5, 6 and 7 discuss experiments, limitations and related work. Section 8 concludes.

2. APPROACH OVERVIEW

An *API Specification* describes all aspects of the behavior of each method on which a caller may rely. For example, the API specification for PipedInputStream states:

> Typically, data is read from a PipedInputStream object by one thread and data is written to the corresponding PipedOutputStream by some other thread. Attempting to use both objects from a single thread is not recommended, as it may deadlock the thread. The piped input stream contains a buffer, decoupling read operations from write operations, within limits.

This explicitly states behavior that a programmer should avoid. A modern API specification is typically a set of HTML pages that is available online or distributed as part of the package, but it may be any sort of document.

A Java API specification is defined by documentation comments embedded in the Java source code. The above quote, for example, is embedded in PipedInputStream.java in one of the comments. Each comment intended to be part of the API specification starts with /** and ends with */ [11]. Java documentation comments are written in HTML with a few extensions, such as the {@link} tag. Javadoc, a tool included with the JDK, extracts these comments and generates a full set of interlinked HTML pages describing the features of the API.

As the source for the API specification, we used OpenJDK 6, an implementation of the Java SE 6 specification. We chose OpenJDK 6 because it is reliable, given that its basis, OpenJDK 7, is based on Oracle's JDK 7 and is the official Java SE 7 Reference Implementation. In particular, more than 96% of the class library has been released by Sun Microsystems. Thus, almost all source code, including the documentation comments, of OpenJDK 6 is identical to that of Oracle's official JDK 6.

Some parts of a documentation comment imply formal specifications—we can infer a formal specification that warns if a single thread performs both read and write operations according to the above quoted text on PipedInputStream—but other parts are descriptive: they merely explain what a method does; e.g. the documentation comment for one of PipedInputStream's constructors is:

> Creates a PipedInputStream so that it is connected to the piped output stream src. Data bytes written to src will then be available as input from this stream.

We believe that it is imperative for programmers to pay attention to specification-implying text in order to write safe and reliable code. To provide a way to make a distinction between comments which imply specifications and those which are merely descriptive, we introduce two pairs of tags: {@property.open} and {@property.close} and {@description.open} and {@description.close}, respectively. {@property.open} can take optional parameters for classifying and referring to formal specifications corresponding to the surrounded text, which will be explained in detail in Section 3.1.

Because it is written in plain English, specification-implying text is merely informative: one cannot check the implied specifications against programs. To enable users to check these specifications at runtime, we formalized them in such a way that an existing Runtime Verification tool can use them. Although we wrote all the specifications for the tool JavaMOP in this paper, they can be easily translated into other formats, such as Tracematches [2].

For example, we defined from the documentation comment above the formal specification shown in Figure 2. At line 1, we name the specification PipedStream_SingleThread, and specify three parameters from which JavaMOP generates code that creates a monitor for each distinct tuple of objects bound to the three parameters. We define six events at lines 2–18: the first four events, named create, represent the creation of a connected pair of PipedInputStream and PipedOutputStream, the write event represents a write() method call on PipedOutputStream, and the read event represents a read() method call on PipedInputStream. At line 20, we specify, in extended regular expression (ERE), that a thread should perform only writes or reads, once a connected pipe is created. According to our definition, an occurrence of an event triggers a transition of the internal state for each related monitor, and a violation of the pattern triggers our handler at lines 22–24. Detailed explanation of the JavaMOP syntax can be found in [14].

To make the newly introduced tags and formal specifications more readable and accessible, we implemented the tool PropDoc as an extension of Javadoc. PropDoc highlights specification-implying and descriptive text with distinct colors, in order to help differentiate them. It also provides

```
PipedStream_SingleThread(PipedInputStream i, PipedOutputStream o, Thread t) {
  creation event create after(PipedOutputStream o) returning(PipedInputStream i) :
    call(PipedInputStream+.new(PipedOutputStream+)) && args(o) {}

  creation event create before(PipedInputStream i, PipedOutputStream o) :
    call(* PipedInputStream+.connect(PipedOutputStream+)) && target(i) && args(o) {}

  creation event create after(PipedInputStream i) returning(PipedOutputStream o) :
    call(PipedOutputStream+.new(PipedInputStream+)) && args(i) {}

  creation event create before(PipedOutputStream o, PipedInputStream i) :
    call(* PipedOutputStream+.connect(PipedInputStream+)) && target(o) && args(i) {}

  event write before(PipedOutputStream o, Thread t) :
    call(* PipedOutputStream+.write(..)) && target(o) && thread(t) {}

  event read before(PipedInputStream i, Thread t) :
    call(* PipedInputStream+.read(..)) && target(i) && thread(t) {}

  ere : create (write* | read*)

  @fail {
    System.err.println("a violation was detected");
  }
}
```

Figure 2: A JavaMOP specification PipedStream_SingleThread

links from specification-implying text to the associated formal specification, if one is provided (see Section 3.1).

3. CATEGORIZING AND FORMALIZING THE JDK API

In this section, we explain the syntax of the new Javadoc tags that we have introduced and how we used them to categorize the JDK API documentation. We then discuss creating formal specifications from the implied specifications in the API documentation.

3.1 Syntax of Documentation Comments

A documentation comment for use with Javadoc consists of two parts: a description and block tags. The former is where both specification-implying and descriptive text is placed, and the latter are a list of tags for explaining parameters, a return value, runtime exceptions, and so forth. All of the new tags introduced for annotating comments are applied to only the former.

As mentioned in Section 2, two pairs of tags were added to make a distinction between two different types of text. Any chunk of text in a description is to be marked as either a `@property` block or a `@description` block.

Since a `@property` block needs further details, the tag accepts a space-separated list of parameters to the `{@property.open}` tag. Each parameter is either a string for further classification or the name of the formal specification that corresponds to the text betwen `{@property.open}` and `{@property.close}`. Parameters specifying formal specifications are prefixed with "`formal:`". For example, we used the following tag at the beginning of the specification-implying text quoted in Section 2 for referring to the specification shown in Figure 2:

```
{@property.open runtime warning do-not-check
                formal:PipedStream_SingleThread}
```

Here, the first three keywords represent the classes associated with this specification. The runtime keyword indicates that the specification can be monitored at runtime. The other two keywords respectively mean that a violation is not necessarily erroneous but potentially dangerous, and that the underlying system does not check violations, as we will explain in Section 3.4.

A `@property` block that explains some desired behaviors can be vague and misleading. To resolve such problems by adding further explanation, we introduce another pair of tags: `{@new.open}` and `{@new.close}`. This pair is not top-level; it is nested in a `@property` or `@description` block.

3.2 Categorizing Documentation Comments

Using the syntax mentioned in Section 3.1, we annotated every documentation comment in `java. io`, `java.lang` and `java.util`, and added further explanation if the text was deemed to need clarification.

We marked each section of text as one of the two top-level tag pairs: `@property` or `@description`. Although the criterion of categorizing descriptions—text should be in `@property` if it implies formal specifications; otherwise, it should be in `@description`—seems clear, there are many fuzzy cases.

One such case is a description of a behavior that does not have the specific and desired pattern. For example, `FileInputStream.available()` is explained as follows:

> Returns an estimate of the number of remaining bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream.

This explains the consequence of calling `read()` when `available()` returns 0; it will block. Although it apparently describes the behavior of the input stream, we did not consider it as a `@property` block because the desired behavior is not clearly implied. One may be tempted to write a specification that prevents calling `read()` in such case, but it can be against the programmer's intention. In fact, many multi-threaded programs use blocking I/O. In contrast, we would consider the following description from `hasNext()` of `Iterator` as a `@property` block:

> Returns `true` if the iteration has more elements. (In other words, returns `true` if `next` would return an element rather than throwing an exception.)

This looks somewhat similar to the description of `available()`, but it implies a desired pattern: check if `hasNext()` returns true, before calling `next`.

Another case is a description of conditions that involve external environments. For example, the constructor of `FileOutputStream` states:

> If the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason then a `FileNotFoundException` is thrown.

Given that a runtime exception is to be avoided, one may think this implies a specification: check if a directory exists, or a file does not exist but a file cannot be created or opened, before creating a `FileOutputStream` object. However, we do not classify this as a `@property` block because the state of the file system dynamically changes without notifying the verification system and, consequently, it is impossible to reliably check whether a file can be created or opened. We decided to mark a section of text with `@property` only if the potentially implied specification solely relies on the internal state of the monitored program.

It is difficult to formalize a specific set of rules that resolves all of the fuzzy cases, but the rule of thumb during categorization was that a section of text is specification-implying only if a behavior is defined in terms of noticeable events, such as class loadings, method invocations and field accesses.

While categorizing documentation comments, we added explanation to vague or misleading sections of texts using the `@new` tag. An example of vague explanation can be found in the explanation of `unread()` of `PushbackInputStream`:

> `IOException` - If there is not enough room in the pushback buffer for the byte, or this input stream has been closed by invoking its `close()` method.

Since it does not clearly state how the buffer size is determined, one cannot realize exactly when the buffer becomes insufficient. Even worse, the entire documentation never mentions the default size of the buffer when the constructor with no arguments is used for creating an object. Since the default size is 1 in OpenJDK 6, we clarified the condition by inserting the following into the vague block:

> `{@new.open}` The size of the pushback buffer is fixed when an object is created. If the size is provided, the buffer size will be as specified; otherwise, it will be 1. `{@new.close}`

Some explanation is even misleading. The documentation comment for `mark()` of `PushbackInputStream` states:

> Marks the current position in this input stream. The `mark` method of `PushbackInputStream` does nothing.

Two sentences conflict: the first sentence states that this method does mark the current position, but the second sentence denies it. We clarified it as well by stating that the method has an empty body and does nothing.

3.3 Writing Formal Specifications

We formalized `@property` blocks after categorizing documentation comments. Since we used a runtime monitoring system in our work, we formalized only runtime-monitorable specifications. Consider the following documentation comments for `Comparable.compareTo()`:

> The implementor must also ensure that the relation is transitive:
> `(x.compareTo(y) > 0 && y.compareTo(z) > 0)` implies `x.compareTo(z) > 0`.

Since this apparently implies a specification, we marked it as `@property`. However, checking if it holds is infeasible at runtime. Not having a means of describing and checking, we decided not to formalize such cases; we simply added the static keyword to `{@property.open}`, indicating that it may be statically checked for some cases.

Among runtime-monitorable specifications, there are a few cases where runtime monitoring tools are incapable of observing necessary events. For example, the documentation states the following for `InputStream.available()`:

> Note that while some implementations of `InputStream` will return the total number of bytes in the stream, many will not. It is never correct to use the return value of this method to allocate a buffer intended to hold all data in this stream.

It is ideal to keep track of uses of the return value of `available()` and check if any of them is used to allocate a buffer. Apart from performance degradation it causes, most runtime monitoring tools do not support local variable tracking. In such cases, we added the uncheckable keyword and did not attempt to formalize it.

Other cases that we did not formalize include those where the specification is enforced by compilers. For example, the documentation of `InputStream` states the requirement of a subclass, as follows:

> Applications that need to define a subclass of `InputStream` must always provide a method that returns the next byte of input.

Java compilers enforce the requirement because `read()`, the method implied by the comment, is an abstract method. We used the enforced keyword in such cases. Another case of not having formal specifications is when the involved events are never exposed to clients; e.g., all the events in the implied specification are private method invocations or field accesses. We marked such cases as internal. Although many tools are capable of monitoring them, we decided not to formalize them because there is no benefit from a user's perspective.

Except these cases, we formalized all the runtime-monitorable specifications using JavaMOP. As shown in Figure 2, a typical JavaMOP specification contains three parts: a set of events, a desired behavioral pattern, and a handler for violations. An event in our specifications is mostly a method invocation, a field access, an end of an execution, or an initialization of an object. We expressed a pattern in either an extended regular expression (ERE), a finite state machine (FSM) or a linear temporal logic (LTL) formula. Depending on the pattern, we tried to choose the most intuitive formalism. Our handler simply outputs a warning message in case of a violation, but one can easily alter this behavior since JavaMOP allows arbitrary code in the handler. However, there are some specifications where the occurrence of a given event, in any context, indicates a violation. In such cases, we omitted the pattern and the handler, and let the events output messages, as will be shown in Figure 4.

3.4 Classifying Formal Specifications

The formal specifications implied by the JDK API documentation have many different characteristics. For example, since a violation of a specification means either an actual error, a potentially dangerous state, or a bad practice, one may want to suppress all the messages, except the ones that definitely indicate errors. To allow users to look up such specifications and turn them off, we classified our specifications according to two criteria.

3.4.1 Severity

According to the severity of a violation of the desired behavior, we classified specifications into three groups: suggestion, warning and error. We use suggestion if a violation is merely a bad practice. StringBuffer_SingleThreadUsage, which will be discussed in Section 4.2, is one such specification. If a violation is not necessarily erroneous but potentially wrong, we use warning; e.g., Serializable_UID (Section 4.3). We use the last group error if a violation indicates an error; e.g., ShutdownHook_PrematureStart (Section 4.4).

3.4.2 Guarantee of the underlying system

Another criterion is what the underlying system, including the Java Virtual Machine and the Java Class Library, guarantees. We classified specifications into three groups: always-check, sometimes-check and do-not-check. One example of the first group is that an `FileOutputStream` object cannot perform write operations after the stream has been closed. As mentioned in Section 1, the library can detect write operations that occur after closing the stream; thus, it is guaranteed that an invalid write operation is always caught by the system. An example of the second group is the *fail-fast* behavior of an iterator: a fail-fast iterator throws an exception if the underlying collection is structurally modified, but there is no guarantee. PipedStream_SingleThread and StringBuffer_SingleThreadUsage belong to the third group; the underlying system never warns any potential danger.

3.5 Augmented API Specifications

We believe API specifications in HTML are the most familiar form for programmers to refer to, and our work should be provided similarly. To this aim, we developed PropDoc for generating API specifications that make a distinction between `@property` and `@description` blocks, and show formal specifications in such a way that the augmentation does not radically change the usual form.

PropDoc consists of two parts: a series of taglets[19], and a driver. A taglet is a Java program that is attached to the Javadoc tool and handles custom tags, such as `{@property.open}`. The driver is a Perl script that takes package names as input and generates a complete API specification, as Javadoc does. It runs Javadoc with the taglets enabled, so that Javadoc dispatches our taglets.

Our taglets highlight text in different colors according to the category, in order to improve the readability. As mentioned in Section 1, the augmented documentation produced by PropDoc has a toggle button to turn on/off the highlighting and to hide text added using the `@new` tag. Additionally, the taglet for `@property` blocks adds links to the implied formal specifications, if any are specified using “`formal:`”.

```
StringBuffer_SingleThreadUsage(StringBuffer s) {
  Thread th = null;
  boolean flag = false;

  creation event init after(Thread t)
                returning(StringBuffer s) :
    call(StringBuffer.new(..)) && thread(t) {
    this.th = t;
  }

  event use before(StringBuffer s, Thread t) :
    call(* StringBuffer.*(..)) && target(s) && thread(t) {
    if (this.th == null) this.th = t;
    else if (this.th != t) this.flag = true;
  }

  event endprogram after() : endProgram() {}

  ere : init use+ endprogram

  @match {
    if (!this.flag)
      System.err.println("a violation was detected");
  }
}
```

Figure 3: A JavaMOP specification StringBuffer_SingleThreadUsage

4. EXAMPLES

In this section, we explain a few examples among 137 formal specifications. More specifications and the augmented API specification can be found at our project website[12].

4.1 PipedStream_SingleThread

This specification, shown in Figure 2, warns if a single thread attempts to use both a `PipedInputStream` object and a `PipedOutputStream` object. It originates from `PipedInputStream`'s comment, mentioned in Section 2.

The severity of this specification is warning as a violation does not always lead to deadlock—if the buffer is large enough to hold the data to be written, write operations and subsequent read operations will not block. That said, a violation implies a potential error because the size of the buffer is system-dependent and can be small in some systems. The underlying system does not check the behavior; thus, it belongs to the do-not-check group.

4.2 StringBuffer_SingleThreadUsage

StringBuffer_SingleThreadUsage checks if a `StringBuffer` object is solely used by a single thread. If this is the case, it outputs a suggestive message stating that `StringBuffer` can be replaced with `StringBuilder` for the performance benefit, as the documentation states:

> `StringBuilder` is designed for use as a drop-in replacement for `StringBuffer` in places where the string buffer was being used by a single thread (as is generally the case). Where possible, it is recommended that `StringBuilder` be used in preference to `StringBuffer` as it will be faster under most implementations.

```
1  Serializable_UID() {
2    event staticinit after() :
3      staticinitialization(Serializable+) {
4      Signature initsig =
5        thisJoinPoint.getStaticPart().getSignature();
6      Class klass = initsig.getDeclaringType();
7
8      if (klass != null) {
9        try {
10         Field field =
11           klass.getDeclaredField("serialVersionUID");
12         int mod = field.getModifiers();
13         Class fieldtype = field.getType();
14
15         boolean isstatic = Modifier.isStatic(mod);
16         boolean isfinal = Modifier.isFinal(mod);
17         boolean islong = fieldtype.getName() == "long";
18
19         if (!isstatic) System.err.println("non-static");
20         if (!isfinal) System.err.println("non-final");
21         if (!islong) System.err.println("wrong type");
22       }
23       catch (NoSuchFieldException e) {
24         System.err.println("undeclared");
25       }
26     }
27   }
28 }
```

Figure 4: A JavaMOP specification Serializable_UID

The formal specification is shown in Figure 3. For each `StringBuffer` object, two monitor variables are defined at lines 2 and 3: `th` remembers the thread that first accessed it, and `flag` remembers if multiple threads have accessed it. A use event is emitted for each method invocation on a `StringBuffer` object and its advice sets the `flag` variable if it detects multiple threads accessing an object (lines 13–14) throughout its lifetime, which begins when a constructor is invoked (i.e., an init event occurs), and ends when either the object is garbage collected or the entire program terminates (i.e., an endprogram event occurs).

We classified this specification as suggestion because a violation does not indicate any potential error; it merely results in performance degradation. As the underlying system does not check such behavior, it is classified as do-not-check.

4.3 Serializable_UID

This specification warns if a class that implements `Serializable` does not declare the `serialVersionUID` field. Since the lack of the declaration does not cause an immediate error, we classified it as warning. We believe, for the same reason, the underlying system does not check the violation. Nevertheless, declaring it is strongly recommended according to the API specification:

> If a serializable class does not declare a `serialVersionUID`, then the serialization runtime calculates a default `serialVersionUID`. However, it is strongly recommended that all serializable classes explicitly declare `serialVersionUID` values, since the default `serialVersionUID` computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected `InvalidClassExceptions` during deserialization.

```
1  ShutdownHook_PrematureStart(Thread t) {
2    creation event good_register before(Thread t) :
3      call(* Runtime+.addShutdownHook(..)) && args(t)
4      && condition(t.getState() == Thread.State.NEW) {}
5
6    creation event bad_register before(Thread t) :
7      call(* Runtime+.addShutdownHook(..)) && args(t)
8      && condition(t.getState() != Thread.State.NEW) {}
9
10   event unregister before(Thread t) :
11     call(* Runtime+.removeShutdownHook(..)) && args(t) {}
12
13   event userstart before(Thread t) :
14     call(* Thread+.start(..)) && target(t) {}
15
16   ere : (good_register unregister)* (epsilon | userstart)
17
18   @fail {
19     System.err.println("a violation was detected");
20   }
21 }
```

Figure 5: A JavaMOP specification ShutdownHook_PrematureStart.

The formal specification is shown in Figure 4. Unlike other specifications, where the desired or undesired condition can be specified solely by the pattern of method invocations or field accesses, more detailed information, such as the modifiers and the type of a field, should be retrieved to describe the undesired condition precisely. Thus, we placed the precise condition check inside the staticinit event handler (lines 3–27), emitted when a static initializer[2] of a serializable class is invoked. At lines 4–6, the enclosing class of the static initializer (i.e., a serializable class) is assigned to the `klass` variable. Then, the modifiers and type of the `serialVersionUID` field are retrieved using reflection (lines 10–17). Three conditional statements at lines 19–21 verify that the field is `static`, `final` and of type `long`, as stated in the reference. If the field does not exist, a warning message is printed at line 24.

4.4 ShutdownHook_PrematureStart

ShutdownHook_PrematureStart warns if a shutdown hook is either running at the time of registration or the user starts it after the registration. A shutdown hook is a `Thread` object for performing user-defined cleanup while the JVM is shutting down, and it is to be started only by the JVM. That is, the user needs to create a `Thread` object but should not start it, as the API specification states:

> A shutdown hook is simply an initialized but unstarted thread. When the virtual machine begins its shutdown sequence it will start all registered shutdown hooks.

Figure 5 shows the formal specification. A bad_register event (lines 6–8) occurs when a started thread is registered as a shutdown hook, and a userstart event (lines 13–14) occurs when the user starts the thread. The desired pattern is that the user can start a previously registered shutdown hook only after it is unregistered. Any violation of this pattern, such as bad_register or good_register followed by userstart, will result in a warning at lines 18–20.

The severity of this specification is error because a violation indicates that the cleanup operation has prematurely started performing. Although the underlying system warns

[2] A static initializer of a class is executed during class initialization after class loading.

	java.io	java.lang	java.util
Total Text	41003 words	77813 words	101038 words
Description Text	37229 words (90.5%)	73503 words (94.5%)	91764 words (90.7%)
Property Text	3774 words (9.2%)	4310 words (5.5%)	9274 words (9.2%)
Undecided Text	0 words (0%)	0 words (0%)	0 words (0%)
# of Specifications	30	49	58

Table 1: Statistics on categorizing the JDK API and the number of specifications from formalizing it.

if an already started thread is registered, it does not detect the user starting the registered thread explicitly. Thus we classified it as sometimes-check.

5. EVALUATION

In this section, we evaluate our categorizing and formalizing of the JDK API. We spent about four person-months to categorize and formalize three main packages of the JDK: `java.io`, `java.lang`, and `java.util`. The formal specifications from this can be used for any formal method (e.g., static analysis, dynamic analysis, runtime monitoring) on any Java program using the three packages. In this evaluation, we monitor each of formalized specifications on two benchmarks, as an example of runtime monitoring usage. Although finding bugs was not our main purpose in our experiments, we found 4 violations of error specifications, 4 violations of warning specifications, and 12 violations of suggestion specifications. A violation can be either a false alarm, a potential bug, or a real bug. Where potential or real bugs occur, we provide a discussion. Note that some specifications may have false positives; especially when they belong to the suggestion or warning group. To suppress false warnings, users have the option to selectively disable specifications.

5.1 Experimental Settings

For our experiments, we used a dual Xeon 2.66GHz (8 cores) / 16GB RAM / CentOS 5.7 machine and version 9.12 of the DaCapo benchmark suite [6] and the SPECjvm 2008 benchmark suite [17]. DaCapo is a set of open source, real world applications with non-trivial memory loads. The latest version 9.12 contains 14 benchmarks. SPECjvm is a benchmark suite for measuring the performance of a Java Runtime Environment, containing several real world applications and benchmarks focusing on core Java functionality. This benchmark suite contains 38 benchmarks. We used the default data inputs and the default settings for all benchmarks. While DaCapo provides its source code so that we can manually inspect the code to check if violations represent real bugs, SPECjvm does not provide its source code. Thus, we could not identify the violations from SPECjvm. We used the Sun JVM 1.6.0 for the entire evaluation. The AspectJ compiler (ajc) version 1.6.4 was used to weave the aspects generated by JavaMOP. We also used the most recent release version of JavaMOP, 2.3.2.

5.2 Results and Discussion

Table 1 shows statistics on categorization and the number of specifications that we found from formalization. In total, 41003, 77813, and 100963 words, respectively, were reviewed and completely categorized into either description text or property text, resulting in 0% undecided text for every package. Among them, 3774, 4310, and 9445 words implied specifications, respectively. Also, we formalized all runtime-monitorable specifications as JavaMOP specifications, resulting in 30, 49, and 58 formal specifications, respectively.

Table 2 shows the number of specifications for each category of specifications: error, warning and suggestion; and the number of violated specifications among those specifications for each benchmark in the benchmark suites. Some specifications, including error specifications, are violated in several benchmarks. After manually inspecting source code, we have found several potential problems and many suggestions for performance improvements. We discuss more detail for each package.

5.2.1 java.io

One error specification, one warning specification and three suggestion specifications from `java.io` are violated on benchmarks in DaCapo. The error specification, Reader_ ManipulateAfterClose is violated on all benchmarks but avrora. However, after analyzing the source code, we found that `SimpleCharStream` intentionally performs a read operation after closing the stream for checking if the stream is closed or not. Also, it handles thrown exceptions properly. There is no bug in this class related to this specification, but this is not a usual pattern of using the `Reader` class according to the JDK API; the code should probably be changed.

The warning specification, Serializable_UID is violated on all benchmarks. According to the JDK API, a `Serializable` class is strongly recommended to have its own version number called `serialVersionUID` to verify compatibility during deserialization. The lack of a version number can cause a problem when the implementation changes or different compiler is used. We found that many `Serializable` classes in DaCapo do not have `serialVersionUID`.

Three suggestion specifications, Closeable_MeaninglessClose, Closeable_MultipleClose, and File_DeleteTempFile are violated on a few benchmarks. In several `Closeable` classes, a close operation has no effect and other methods can be called even after a close operation (Closeable_MeaninglessClose). Also, closing a previously closed `Closeable` class instance has no effect (Closeable_MultipleClose). However, it is safer to call the `close` method multiple times or even one time for the classes that are not affected by it, than forgetting to call. Therefore, we are neutral on these specifications. But, for eclipse, we recomend deleting temporary files according to the File_DeleteTempFile specification.

In SPECjvm, one warning specification and one suggestion specification are violated: Serializable_UID and Closeable_MultipleClose, respectively. We could not investigate the source code of SPECjvm as we mentioned, but we assume that similar things happened in SPECjvm. Similarly to DaCapo, we are neutral about Closeable_MultipleClose, but the `Serializable` classes need to have `serialVersionUID`.

5.2.2 java.lang

One error specification and eight suggestion specifications from `java.lang` are violated on several benchmarks in DaCapo. One of benchmarks in DaCapo, h2 violates the error specification ShutdownHook_LateRegister, which states that one cannot add/remove any shutdown hook once the shutdown procedure has begun. After analyzing the source code,

		java.io			java.lang			java.util		
		error	warning	suggestion	error	warning	suggestion	error	warning	suggestion
Total # of specifications		19	6	5	24	11	14	44	11	3
DaCapo	avrora	0	1	0	0	0	2	0	0	1
	batik	1	1	2	0	0	5	0	3	1
	eclipse	1	1	1	0	0	4	0	2	1
	fop	1	1	0	0	0	2	0	0	1
	h2	1	1	0	1	0	2	0	2	1
	jython	1	1	2	0	0	6	1	1	1
	luindex	1	1	0	0	0	2	0	0	1
	lusearch	1	1	0	0	0	2	0	0	1
	pmd	1	1	0	0	0	2	0	0	1
	sunflow	1	1	0	0	0	3	0	0	1
	tomcat	1	1	0	0	0	3	0	1	1
	tradebeans	1	1	0	0	0	2	0	0	1
	tradesoap	1	1	0	0	0	2	0	0	1
	xalan	1	1	2	0	0	4	0	2	1
SPECjvm	startup.* (17 benchmarks)	0	0	0	1	0	3	0	1	1
	compiler.compiler	0	0	0	1	0	3	0	1	1
	compiler.sunflow	0	0	0	1	0	3	0	1	1
	compress	0	0	0	1	0	3	0	1	1
	crypto.aes	0	0	0	1	0	3	0	1	1
	crypto.rsa	0	0	0	1	0	3	0	1	1
	crypto.signverify	0	0	0	1	0	3	0	1	1
	derby	0	0	0	1	0	3	0	1	1
	mpegaudio	0	0	0	1	0	3	0	1	1
	scimark.* (9 benchmarks)	0	0	0	1	0	3	0	1	1
	serial	0	1	0	1	0	3	0	1	1
	sunflow	0	0	0	1	0	3	0	1	1
	xml.transform	0	0	1	1	0	3	0	1	1
	xml.validation	0	0	0	1	0	3	0	1	1

Table 2: The number of violated specifications for each benchmark (startup.* and scimark.* benchmarks show the same violation pattern).

we found that a shutdown hook, `org.h2.engine.DatabaseCloser` is executed during the shutdown procedure, it calls the `close()` method of `org.h2.engine.Database`, and this method tries to remove the shutdown hook, which is not allowed. Although the exception is correctly handled, this design could potentially cause problems.

Most of violated suggestion specifications are suggestions on performance: to use faster constructors or to use faster data types in a particular situation. Although these suggestions are from the JDK API, some specifications might not be useful to some users. When an inefficient data structure is very lightly used, the improvement is very subtle. Despite small performance degradation, one may choose to use the slower data structures for better readability or easier maintenance. Also, a thread-safe data structure is suggested when multiple threads use it, but there might be external synchronization. In those cases, suggestions from violated specifications can be ignored. Nevertheless, it is still a good chance to review the implementation and look for possible improvements.

In SPECjvm, one error specification, System_NullArrayCopy and three suggestion specifications are violated. All suggestion specifications are suggestions on performance, which are discussed above. System_NullArrayCopy says that one should not use `null` in the source or in the destination when calling `System.arraycopy()`. Again, we could not investigate the source code of SPECjvm. Therefore, this violation could be a false alarm.

5.2.3 *java.util*

One error specification, three warning specifications, and one suggestion specification from `java.util` are violated on benchmarks of DaCapo. An error specification, Collections_SynchronizedCollection, is violated on jython. After inspecting the source code, we found that jython does not synchronize on a thread-safe list when iterating over the list using a thread-unsafe way, which can cause data races.

Three warning specifications, Dictionary_Obsolete, Iterator_HasNext and StringTokenizer_HasMoreElements, are violated on a few DaCapo benchmarks. The Dictionary_Obsolete specification, which states that the `Dictionary` class is obsolete and not to be used, is violated because some of its subclasses are actually used. Two benchmarks violate Iterator_HasNext, but it turned out that the violations do not indicate actual errors because it is legal to consecutively call `Iterator.next()` if it is assured that the next element exists. The StringTokenizer_HasMoreElements specification is violated by three benchmarks because they assume at least one token is always available. Although the assumption may be valid in some cases, it is recommended to first check whether or not a token is available.

A suggestion specification, Enumeration_Obsolete, is violated on all benchmarks. Using `Enumeration` is not erroneous, but the specification warns because the documentation recommends `Iterator`, which can replace `Enumeration`.

In SPECjvm, only two specifications, Dictionary_Obsolete and Enumeration_Obsolete, are violated. Thus, there is no potential bug related to API usages of `java.util`.

6. LIMITATIONS & LESSONS LEARNED

A few drawbacks stem from the fact that we manually inspected documentation comments. Manual inspection is time-consuming, although it gives sophisticated and precise specifications that other automated tools would not infer.

We estimate that about four person-months were spent for categorizing 219854 words and writing 137 formal specifications in three packages of the JDK.

Another drawback is that some inconsistency may exist in categorization of documentation comments and we may have overlooked specification-implying text. Categorization is a subjective process because specification-implying text is often implicit in JDK's API specification, like any other documentation written in natural languages. For example, we found the quoted text in Section 4.2 specification-implying, but others may think it is descriptive.

The fact that we completely rely on JDK's API specification can be another source of missing formal specifications. Unlike dynamic approaches, which infer specifications from frequently observed behaviors, our work fails to provide specifications that the documentation fails to mention. For example, a dynamic specification mining tool, such as jMiner[13], could infer from program executions that when an `OutputStream` (or its subclass) object is built on top of an underlying `ByteArrayOutputStream` object, it should be flushed or closed before the underlying object's `toByteArray()` is invoked. This behavioral pattern is indeed desired—because failing to fulfill the requirement may cause `toByteArray()` to return incomplete contents—but manual inspection would miss it as documentation does not mention it.

7. RELATED WORK

Providing augmented documentation Some works have been done to provide more informative documentation. One of them is eMoose[7], an eclipse[8] plugin. This tool helps users notice *directives*, which are similar to our `@property` blocks, by highlighting them when the documentation appears. It also identifies all method calls whose targets have directives and makes them easily noticeable from the source code editor. Unlike our work, eMoose does not formalize directives; thus, one cannot check them against programs.

Java Modeling Language (JML)[16] is a behavioral interface specification language for Java, and allows one to specify method contracts and invariants. Its JmlDoc tool generates the API specification from JML-annotated Java files. The generated documentation shows contracts and invariants for each method and class, if available. Since JML is not designed to categorize documentation comments, it does not highlight specification-implying text.

Formalizing desired behaviors Various ways of formalizing behaviors have been also proposed. Monitoring Oriented Programming (MOP)[14], a runtime verification framework, defines its own syntax for expressing specifications to be verified. In MOP, one can define events, such as method invocations, and specify the desired or undesired behavior over the events using logical formalisms with actions for handling violations or validations of the specified behavior. Another formalization approach is presented by JML[16], which allows one to add contracts and invariants for each method and class. Although behaviors are described differently, many of them can be formalized in both MOP and JML. For example, one can formalize the following description in both MOP and JML:

> Once the stream has been closed, further `read()`, `available()`, `reset()`, or `skip()` invocations will throw an `IOException`. Closing a previously closed stream has no effect.

In JavaMOP, the Java instantiation of MOP, one can define an event for each method, and then specify the undesired behavior using the following ERE:

```
close+ (read | available | reset | skip)+
```

When any of four manipulation methods is invoked after `close`, the pattern is matched, and JavaMOP invokes the user-defined handler, which can contain arbitrary Java code. In JML, one can define a model field of type boolean, called `isOpen`, which is set when a stream is created and unset when it is closed. One can then specify a precondition on the four manipulation methods to ensure `isOpen` is true.

Inferring specifications Since formalizing behaviors requires much human effort, many approaches to specification mining have been proposed. Daikon[9] observes program executions and infers pre- and post-conditions and invariants, at every public method entry and exit of a class, in JML and other formats. Although the inferred information is useful, it does not explicitly describe behavioral patterns. Ammons et al. [3] propose another dynamic approach that infers FSMs over functions, which show patterns explicitly. This can infer arbitrarily complex FSMs, but requires expert knowledge, such as functions of interest. jMiner[13] also infers arbitrarily complex FSMs, but it is capable of inferring methods of interest from unit test cases as well, eliminating necessity of the expert knowledge.

Rather than inferring arbitrarily complex FSMs, several techniques detect methods that behave as specified in the pre-defined patterns. Perracotta[20] infers all pairs of methods that satisfy the alternating pattern, $(ab)^*$, from execution traces. Gabel and Su [10] extend Perracotta; it considers an additional pre-defined pattern $(ab^*c)^*$, known as resource usage pattern.

There have been a number of static specification mining techniques as well. Unlike dynamic techniques, static ones do not need clients that can be rare, unless the target library is widely used. Many techniques infer specifications from the source code[1, 21], but some infer them from comments using natural language processing (NLP); e.g., Zhong et al. [22] propose an automated technique to infer the resource usage specifications from API specifications.

Verifying specifications Numerous static and dynamic techniques have been proposed to verify formal specifications. ESC/Java2[16] statically verifies a subset of JML annotations, although it is neither sound nor complete. MOP[14] synthesizes from formal specifications monitors that can dynamically check specifications at runtime. Being executed along with the target program, runtime monitoring can cause performance degradation. Although various optimizations have been devised, the performance benefit from them has been hard to compare due to the lack of an extensive benchmark suite—comparison has been made among only several specifications in the literature. We believe our work can be a useful benchmark suite for runtime monitoring systems.

8. CONCLUSION

One of hurdles in developing reliable software is the lack of formal specifications. We believe that one reason why documentation does not contain formal specifications is that no logical formalism provides a means of expressing all the various types of requirements that clients should obey. As a result, the requirements are informally written in plain English. Since these informal requirements are mixed with other descriptions, readers may not notice the existence of requirements. Another consequence of not having formal specifications is that potentially dangerous code is not revealed in early stage of development.

In this paper we show that many implicit requirements can be actually formalized and the inferred formal specifications can be utilized by existing runtime verification tools. As our experiments show, they were useful enough to provide suggestions and warnings, and even reveal flaws in mature software. Additionally, our extensive set of specifications can be used to compare performance among monitoring systems and possibly stimulate performance improvement.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *FSE*, 2007.

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA*, 2005.

[3] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, 2002.

[4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *VMCAI*, 2004.

[5] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from EAGLE to RULER. *J. Logic Computation*, November 2008.

[6] S. M. Blackburn, R. Garner, C. Hoffman, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[7] U. Dekel and J. D. Herbsleb. Improving API documentation usability with knowledge pushing. In *ICSE*, 2009.

[8] eclipse. `http://www.eclipse.org/`.

[9] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, 2000.

[10] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE*, 2008.

[11] How to Write Doc Comments for the Javadoc Tool. `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`.

[12] C. Lee, D. Jin, P. O. Meredith, and G. Roşu. Annotated Java API. `http://fsl.cs.uiuc.edu/annotated-java/`.

[13] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *ICSE*, 2011.

[14] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Tools for Technology Transfer (STTT)*, pages 1–41, 2011.

[15] OpenJDK. `http://openjdk.java.net`.

[16] E. Poll, P. Chalin, D. Cok, J. Kiniry, and G. T. Leavens. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO*, 2005.

[17] SPECjvm 2008. `http://www.spec.org/jvm2008/`.

[18] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12: 157–171, January 1986.

[19] Taglet. `http://docs.oracle.com/javase/6/docs/technotes/guides/javadoc/taglet/overview.html`.

[20] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*, 2006.

[21] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP*, 2009.

[22] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *ASE*, 2009.