Automatic Detection and Repair Recommendation of Directive Defects in Java API Documentation

Yu Zhou, Changzhi Wang, Xin Yan, Taolue Chen, Sebastiano Panichella, Harald Gall

Abstract—Application Programming Interfaces (APIs) represent key tools for software developers to build complex software systems. However, several studies have revealed that even major API providers tend to have incomplete or inconsistent API documentation. This can severely hamper the API comprehension and, as a consequence, the quality of the software built on them. In this paper, we propose *DRONE* (**D**etect and **R**epair of d**O**cumentatio**N** d**E**fects), a framework to automatically detect and repair defects from API documents by leveraging techniques from program analysis, natural language processing, and constraint solving. Specifically, we target at the directives of API documents, which are related to parameter constraints and exception handling declarations. Furthermore, in presence of defects, we also provide a prototypical repair recommendation system. We evaluate our approach on parts of the well-documented APIs of JDK 1.8 APIs (including javaFX) and Android 7.0 (level 24). Across the two empirical studies, our approach can detect API defects with an average F-measure of 79.9%, 71.7%, and 81.4%, respectively. The API repairing capability has also been evaluated on the generated recommendations in a further experiment. User judgments indicate that the constraint information is addressed correctly and concisely in the rendered directives.

Index Terms—API documentation; directive defects; natural language processing; repair recommendation

1 Introduction

Application Programming Interfaces (APIs) are widely used by developers to construct or build complex software systems [1]. Popular applications such as Facebook, Pinterest, Google Maps, or Dropbox are well-known examples of major API providers. A developer of a mobile app, for instance, can use Google Maps APIs for implementing an application that requires user localization information. APIs are beneficial not only for software developers to build software, but also for users of software [2], [3], [4]. For example, many Facebook users enjoy the possibility to sign into Web sites and applications using their Facebook ID, a feature that works on top of the Facebook APIs.

API documents represent the most important references for developers to seek assistance or instructions on how to use a given API [5], [6]. API documents need to express the assumptions and constraints of these APIs, i.e., the usage context, so that the clients can follow these guidelines and avoid pitfalls when using them [7]. For instance, Javadoc documents usually provide the main assumptions and constraints of these APIs, as well as useful examples for various usage contexts or scenarios. However, software evolution in turn may lead to API changes. With these changes, the corresponding documents are accidentally overlooked and not adapted accordingly, so that defective API documents are frequently introduced in practice [8]. By *defective API documents*, we mean incomplete or incorrect, and therefore no longer accurate documentation of an API. Consequently, according to several studies, API providers tend to release incomplete or inconsistent API documentation, which deviates from the actual API implementation [4], [9], [10], [11].

Thus, defective API documents are frequently encountered in practice: developers and API users get confronted with inconsistencies present in these documents. This can severely hamper the API comprehension and the quality of the software built on top of it. To address defective API documents, developers try to infer the correct or required knowledge from the source code of the API itself or by searching source code descriptions reported in external artifacts (e.g., StackOverflow) [10], [11], [12], [13]. For instance, some developers, while discussing the use of popular APIs of Facebook, PHP, or JavaScript, state that: *"...the functionality was there, but the only way to find out how to accomplish something was to dig through Stack Overflow"* [14]. However, many times API users get easily frustrated by repeated bugs and inconsistencies in API documents; hence, they tend to abandon the API in favor of another vendor's API [15], [16].

Let us first provide some concrete examples of *API documentation defects* from JDK1.8 and Android 7.0 (API

- *Y. Zhou, C. Wang and X. Yan are with College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China. Y. Zhou is also with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China. E-mail: {zhouyu,czwang,xin_yan}@nuaa.edu.cn*
- *T. Chen is with Department of Computer Science and Information Systems, Birkbeck, University of London, UK. He is also with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, and RISE, Southwest University, Chongqing, China. E-mail: taolue@dcs.bbk.ac.uk*
- *S. Panichella is with Department of Informatics, Zurich University of Applied Science, Switzerland. E-mail: spanichella@gmail.com*
- *H. Gall is with Department of Informatics, University of Zurich, Switzerland. E-mail: gall@ifi.uzh.ch*

Manuscript received xxxx, 201X; revised xxxx, 201X.

Level 24) to illustrate how they typically look like and why it is important to detect and fix them.

1.1 Real-life cases of API documentation defects

Type compatibility. In JDK1.8, the API document for the method javax.swing.JTabbedPane.addTab(String title, Component component) states that this method is to "add a component represented by a title and no icon, the title—the title to be displayed in this tab, component—the component to be displayed when this tab is clicked." For a developer, this means that it should be possible to pass the method an instance of the javax.swing.JFrame type, since this argument is compatible with the Component type in Java, based on the API documentation. Such kind of API usage will also pass the static check. However, when running, an exception will be thrown. By manually analyzing the code, we found that addTab invokes insertTab which, in turn, invokes addImpl. The body of addImpl contains an assertion to check whether or not one of the arguments (i.e., Component in this case) is of the Window type. The documentation of addImpl does clarify that, if a Window object is added to a container, an IllegalArgumentException will be thrown. However, this essential constraint is not mentioned at all in the API documentation for insertTab or addTab.

Range assumptions. In JDK1.8, the API document for java.awt.font.TextLayout.getBlackBoxBounds (int firstEndpoint, int secondEndpoint) only states that the argument firstEndpoint is "one end of the character range" and the argument secondEndpoint is "the other end of the character range. Can be less than firstEndpoint." This description turns out to be incorrect. Indeed, the corresponding code actually requires that the firstEndpoint is not less than 0, and the secondEndpoint is no more than the value of the character counts; otherwise, an IllegalArgumentException will be thrown.

Type checks. In JDK1.8, the API document for javax.swing.JTable.getDefaultEditor(class columnClass) states that "columnClass return[s] the default cell editor for this columnClass." However, in the corresponding implementation, the code actually first checks whether or not the argument c is null. If it is, the method directly returns a null value without throwing an exception. But this information is not even mentioned in the documentation, instead, the API documentation discusses what will happen if c is *not* null.

Parameter values. Again in JDK1.8, the API document for java.awt.event.InputEvent.getMaskForButton(int button) states that "if button is less than zero or greater than the number of button masks reserved for buttons." However, in the corresponding source code, one may find that the exceptional condition is button $<=$ 0 $||$ button $>$ BUTTON_DOWN_MASK.length, i.e., the code actually requires that the value of button should be no greater than 0 — the documentation is incorrect in specifying the range of the argument button.

In Android 7.0, the API document for android.view.Choreographer.removeCallbacks(int callbackType, Runnable action, Object token) *only* explains that the parameter callbackType is the "callback type" (without further constraints). Meanwhile, the code does state that the value of the parameter has to be between 1 and a constant CALLBACK_LAST, otherwise, an IllegalArgumentException will be thrown. As another example in the same API library, the documentation of android.media.FaceDetector.findFaces(Bitmap bitmap, FaceDetector.Face[] faces) states that "faces, ..., must be sized equal to the maxFaces value ... ". However, the code requires that the size of parameter faces must *not* be smaller than the maxFaces value, otherwise an IllegalARgumentException will be thrown.

The above examples are simple but evident examples of issues that we refer to as "API documentation defects." Indeed similar problems can be found in many API documents. On StackOverflow, a contributor complained that "[t]he Javadocs are often flat-out wrong and contradictory, and Sun has often not even bothered to fix them even after 10 years."[1] Note that the projects JDK and Android are generally considered to be of high quality in their API documentation. Hence, we may consider that the API documents of other projects may suffer from similar or even more severe issues.

1.2 Goals and research questions

Saied et al. [17] enumerated categories of common API usage constraints and their documentation. Undoubtedly, high-quality documentation is indispensable for the usability of APIs [18], [19], and a complete and correct API document is key for API users. However, given the bulk of API documents and code, it is practically infeasible to check and discover such problems manually. Even if it is manageable on a small scale, the manual examination would be tedious, inefficient, costly, and error-prone. Automated solutions to address these problems are needed.

In this paper, we present *DRONE* (***D***etect and ***R***epair of d***O***cumentatio***N*** d***E***fects), an *automated* approach to detect the defects of API documents, as well as to recommend repair suggestions. *DRONE* combines program analysis, natural language processing, and logic reasoning. It performs the following four main steps (illustrated in Figure 1): (1) First, it extracts an annotated document from the source code; (2) Second, it parses the source code to obtain an *abstract syntax tree* (AST). Based on the AST, control flow decisions and exception handling, as well as call invocations between the methods, are analyzed; (3) Next, it uses natural language processing techniques to tag the text features present in API documents and to extract the relevant parts on restriction and constraints; (4) Finally, it represents the extracted information in first-order logic (FOL) formulae and leverages satisfiability modulo theories to detect potential inconsistencies between documentation and code.

A "defect" in our context encompasses two scenarios. In the first scenario, the constraint description of API usage is incomplete in the documentation (see the examples in JDK and in Android); in the second scenario, the description exists but it is semantically incorrect with respect to the code. Indeed, as reported in a study conducted by Novick et al. [7], completeness and accuracy represent the two most

1. cf. http://stackoverflow.com/questions/2967303/inconsistency-in-java-util-concurrent-future

important attributes of good documentation. We do not treat syntactic errors in the documents as defects, since most of such errors could be detected by grammar checkers and may not be relevant for developers. Instead, we focus on the *semantic* aspects. We argue that by identifying and correcting these defects the quality of API documents can be increased, which would positively enhance their usability.

The goal of our study is to investigate to what extent *DRONE* is able to automatically detect defects and provide solutions to fix them, taking into account the following two scenarios: (i) when the constraint description of API usage is incomplete; (ii) when the description exists but is *semantically incorrect* with respect to the code. Specifically, we measure the accuracy of *DRONE* in the context of the well-documented APIs of JDK 1.8 and Android 7.0-level 24 (as described in Section 3). [20] defines *directives* as statements on function signatures (i.e., related to parameter types and values) and exceptions. These are the main focus of our work.

We designed our study to answer two main research questions (RQs) on basis of the well-documented APIs of JDK1.8 and Android 7.0:

RQ1: *To what extent does DRONE discover directive defects in Java/Android API documentation?*
We assess the accuracy of *DRONE* in detecting *directive defects* in API documentation.

RQ2: *To what extent does DRONE provide coherent repairing solutions for the detected API documentation defects?*
We qualitatively analyze the capability of *DRONE* in automatically recommending repairs of the detected defects of API documentation directives.

1.2.1 Main assumptions of the research

In this paper, we assume that the API code is correct. The rationale is that they have gone through extensive tests and validation before delivery, hence they are more reliable compared to the documentation. (This assumption can be relaxed though; cf. Section 4.) On the other hand, the API documents are usually a combined description of various pieces of information, such as general descriptions, function signature descriptions, exception throwing declarations, code examples, etc.

Among these, we hypothesize that directives provide the most crucial information for API using developers. Particularly, we focus our attention on method parameter usage constraints and relevant exception specifications. They belong to the method call directive category which represents the largest portion of all API documentation directives (43.7%) [20]. Indeed, all of the discussed illustrative examples (cf. Section 1.1) are directives of this category. We are convinced that automatic detection of such defects in API documents will be of great value for developers to better understand and avoid inappropriate use of APIs. In Java programs, this kind of directive is generally annotated with *@param, @exception, @throws, etc.* tags. Such structured information makes it easier to automatically extract the document directives.

1.3 Contributions

In summary, this paper makes the following contributions:

1) We propose an approach, which can automatically detect and help repair the defects of API document directives. The approach includes static analysis techniques for program comprehension as well as domain-specific, pattern-based natural language processing (NLP) techniques for document comprehension. The analysis results are presented in the form of first-order logic (FOL) formulae, which then are fed into the SMT solver Z3 [21] to detect the defects in case of inconsistency. A pattern-based patch will be recommended to the users, suggesting how to repair the defects.
2) The approach handles four types of document defects at the semantic level, and these are evaluated on parts of the JDK 1.8 APIs (including javaFX) and Android 7.0 APIs (Level 24) version and their corresponding documentation. The experimental results show that our approach is able to detect 1689, 1605 and 621 defects hidden in the investigated documentation of selected APIs respectively. Moreover, the precision and recall of our detection are around 76.4%, 83.8%, 59.4%, 90.3%, 74.7% and 89.4% respectively.
3) We define more than 60 heuristics on the typical descriptions of API usage constraints in API documents, which can be reused across different projects.
4) We have implemented a prototypical API document defect repair recommender system *DRONE*. Not only can it facilitate the detection and repair of API defects for JDK1.8 and Android 7.0, but it also has the potential of wider applicability in other APIs.

This paper is an extended version of our previous work [9], which has been significantly extended in the following ways:

1) We substantially extended our original study with *new data and experiments*: We added more libraries to the data analysis including the latest Android APIs to investigate the broader applicability of *DRONE*. Moreover, we extended the size of the original artifact study, extending Experiments I and II of our original paper, by adding further libraries, analyzing in total 27 API libraries belonging to the latest JDK and Android versions. In particular, we also considered javaFX as the javax.swing and java.awt packages are legacy packages which are being replaced by javaFX.
2) We integrated *new constraint tags* in the detection functionality of defects in API documentation of *DRONE*. We integrated the null value constraints related tags, i.e., @NonNull/NotNull and @Nullable as they are increasingly used nowadays. By including these tags in our approach, it enables the detection of further documentation defects (as described in the discussion section).
3) We then *investigated the extensibility and reliability* of *DRONE*, also on the selected Android libraries, observing to what extent the approach is able to

detect API documentation defects in such new and well-documented libraries. Thus, we extended the original study with a further experiment (i.e., Part 2 of Experiment II) reporting the results of DRONE when detecting defects in the Android API documents.

4) We have further integrated *an automated repair mechanism* into *DRONE* that recommends fixes for the detected directive defects. In particular, we added a completely new artifact study (i.e., Experiment III), answering an additional research question (RQ2) and involving different analyses to demonstrate the practicality of our approach to repair API documentation defects of software libraries. We *clarified the exact nature and scope of our approach*, i.e., that it is highly generic and applicable to further contexts (e.g., applicable to Android libraries as reported in Section 3.2) and how it relates to other existing approaches (in Section 5).
5) We provide a replication package[2] of the work with (i) materials and working data sets of our study, (ii) raw data (to support future studies), (iii) the NLP patterns, the defined NLP heuristics and the repair solutions provided by *DRONE*. Furthermore, we present a prototypical implementation (described in Section 3) of the approach, which is based on the Eclipse plugin architecture. The source code of the prototype is also available for future study and extension.[3]

Main findings. We performed three experiments to demonstrate the effectiveness of our approach. Results of the first experiment, related to API defect detection capability of *DRONE*, show that our approach achieved high F-measure values, ranging between 71.7%, and 81.4% for the various analyzed libraries. In our second experiment, we investigated the applicability on many more API packages, including the one of Android. Results of the second experiment demonstrate that the performance of *DRONE* is still encouraging. Interestingly, for libraries of the latest Android APIs, *DRONE* achieved a precision rate of 74.7% and a recall rate of 89.4%.

In a third experiment, we involved developers to judge the quality of the generated documentation repair recommendations provided by *DRONE*. According to this study, *DRONE* achieved average scores of 4.48, 3.82, 4.53, 4.31 (out of 5), in terms of accuracy, content adequacy, and conciseness & expressiveness, respectively. Qualitative answers from the involved participants suggested the need to improve *DRONE* by generating more elaborated descriptions/templates for "if statement" with many conditions.

In summary, with *DRONE* we are able to discover various API directive defects in JDK and Android API documents. This is in contrast to what is generally believed that widely used and well-documented APIs would not exhibit such defects.

Paper structure. The remainder of the paper is structured as follows. Section 2 illustrates the technical details of our approach. The prototype implementation and experiments with performance evaluation are presented in Section 3 followed by a discussion in Section 4. Section 5 puts our approach in the context of related work, and Section 6 provides some conclusions and outlines future research.

2. https://github.com/DRONE-Proj/DRONE/tree/master/Replication
3. https://github.com/DRONE-Proj/DRONE/tree/master/DRONE

2 Approach

In the current work, we mainly focus on four cases of parameter usage constraints following [17]. These constraints include *nullness not allowed*, *nullness allowed*, *range limitation*, and *type restriction*, a brief explanation of which is given as follows.

- "Nullness not allowed" refers to the case that the null value cannot be passed as an argument to a method. If this constraint is violated, an exception (e.g., NullPointerException) will be thrown.
- "Nullness allowed" refers to the opposite case of "Nullness not allowed." In this case, the null value can be passed as an argument and no exception will be thrown. When the method is invoked, there is a default interpretation of the null value.
- "Type restriction" refers to the case that some specific type requirements must be imposed on the argument. Apart from the common one that argument types must be compatible with the declared parameters, they usually include some additional, implicit rules which must be respected. This is usually due to the features of object-oriented languages, in particular, inheritance.
- "Range limitation" refers to the case that there are some specific value ranges for the arguments. If, otherwise, the values of the arguments are out of scope, then usually exceptions will be thrown.

Common rationale expects such usage constraints are specified explicitly in the accompanying documents for a better understanding and application of APIs, otherwise it could potentially mislead the API users. This is, unfortunately, not the case in practice, which has led to numerous defects in API documentation. The aim of our work is to detect these defects automatically.

To this end, we propose an approach based on program analysis, natural language processing, as well as logic reasoning. Our approach consists of four main steps, which are illustrated in Figure 1 with the following brief description.

1) We first extract an annotated document out of the source code which is a relatively simple procedure. We then have two branches (cf. Figure 1).
2) In the upper branch, we exploit static code analysis. We parse the code to obtain the *abstract syntax trees* (AST). Based on the AST, the statements of control flow decisions and exception handling, as well as the call invocation relation between methods, can be analyzed. The results are given in a (simple, meaning no quantifiers) form of FOL expressions. The details of this step will be elaborated in Section 2.1.
3) In the lower branch, we exploit natural language processing techniques. In particular, we tag the POS features of the directives of the API documents and extract the relevant parts on restrictions and

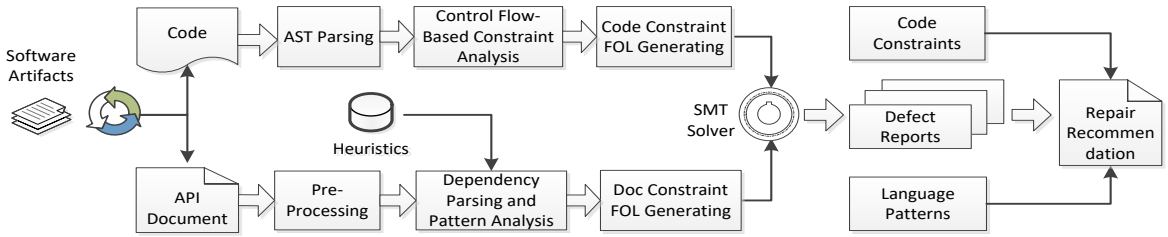


Fig. 1. Approach overview

constraints. They are also encoded as simple FOL expressions. The details of this step will be elaborated in Section 2.2.

4) The upper and lower branches meet when an SMT solver is employed to solve the logical equation relation between the pair of FOL formulae derived from the respective procedures. Potential inconsistencies will be reported based on the result returned from the SMT solver.

For technical reasons and for the scalability of the approach, we make the following assumptions.

- For the second step, i.e., the code analysis: (1) we usually bound the depth of the call graph, which is specified as a hyperparameter of the procedure and is provided by the user; (2) we disregard private methods since they are invisible to end users; (3) we disregard method calls in the conditions of statements; and (4) we do *not* consider aliasing or dynamic dispatching for exception propagation.
- For the third step, i.e., handling directives, we concentrate on the directives of the form

  @tag target description

  Here, tag primarily ranges over "param", "exception" and "throws". We note that the null value constraints related tags, i.e., @NonNull/NotNull and @Nullable are increasingly used nowadays. Thus we also include these tags in our approach. Target refers to the tagged entity and description refers to the constraint related to the target expressions. We hypothesize that API developers tend to use recurrent linguistic patterns to describe the constraints in these expressions.

In the following subsections, we will describe the two processing branches consecutively.

2.1 Extract constraints from API source code

In this subsection, we illustrate the workflow of the upper branch in Figure 1. This branch mainly involves static code analysis techniques based on AST and call hierarchy. In particular, a complete program execution usually includes the participation of multiple procedures, and their invocation relation constitutes a call hierarchy. The parameter related constraints are usually introduced by such a hierarchy. The procedure goes through the following steps with API source code as the input, and a FOL formula as the output.

Step 1: Construct AST. By parsing the API source code, we extract an AST $tree_m$ for each method m. This step is usually a routine of program analysis. In addition, we generate the *static call graph* G by Eclipse's *CallHierarchy*[4]. From the call graph, we define the *call relation* $\mathsf{call}(m, n)$ by computing the transitive closure of the edge relation in the call graph such that $\mathsf{call}(m, n)$ holds if and only if method m calls method n. Note that, as specified in the assumption (1), we bound the depth of the call graph, so technically we compute a sound approximation of $\mathsf{call}(m, n)$; this is usually sufficient in practice.

Step 2: Extract exception handling information. For each public method m in the API, by traversing the AST $tree_m$, we locate each throw statement and collect the associated exception information. We store the exception information of each method m as a set $\mathrm{ExcepInfo}_m$ of tuples, each of which is of the form (m, P, t, c) where

- m is the current method name,
- P is the set of formal parameters of m,
- t is the type of the exception, and
- c is the trigger condition of this exception.

After this step, the directly throwable exception information, as well as the propagating exception information introduced by the method invocation, is obtained.

The pseudo-code of $expExtractor$ is given in Algorithm 1. The AST parsing part is implemented with the aid of the Eclipse JDT toolkit. In particular, the methods *isThrowable*, *isComposite* and *isMethod* in the pseudo-code are from there. The inputs of $expExtractor$ comprise the statement sequence of the source code (represented as an AST) and the depth of the call hierarchy. The algorithm first iterates over all the statements in m. If a statement contains an exception-throw, the relevant information—the exception type, trigger condition, the relevant parameter, as well as the method name—will be recorded and inserted into the list $infoList$ (line 5-6). Note that we use backtracking to calculate a conjunction of the trigger conditions in case of multiple enclosed branches. (For instance, for the snippet `If (A){...If (B) throw...}`, both A and B are collected as conjuncts of the trigger condition.) To handle statements with a composite block, we recursively go through the internal statements of the block and extract the corresponding exception information (line 7-10). To handle method calls (for instance, method m calls method n), if m's argument(s) is(are) passed onto the callee method n, we will use the recursion with parameters, i.e., the statement body of the callee method as the parameter, together with the depth value decreased by 1 (line 12). Note that we require parameter match in the invocation case to track and to guarantee the constraints are on the same parameter list as the caller method. This recursion continues until the depth

4. org.eclipse.jdt.internal.corext.callhierarchy, used in the plugin environment

Data: $stmList$: AST statement block of a method m, and dep:integer
Result: $infoList$: list of exception information, which records the flow-exception tuples, i.e, (m, P, t, c)

```
1  begin
2    infoList ⟵ ∅;
3    if dep≥0 then
4      foreach stm ∈ stmList do
         /* If stm throws an exception,
            records all information in a tuple
            and add to the list                  */
5        if isThrowable(stm) then
6          infoList ⟵ infoList∪ {(m, P, t, c) | P :
             parametert, t : exceptiontype, c : condition}
         /* Recursively invoke itself, in case
            of composite statement               */
7        else if isComposite(stm) then
8          List subList ⟵ (Block)stm.getBody();
9          infoList ⟵
             infoList ∪ expExtractor(subList, dep);
         /* If the statement contains a method
            call of n, check the invoked
            method recursively                   */
10       else if isMethod(stm) ∧ (stm's args ∈ m's list) then
           /* n is the callee of m in stm       */
11         mList ⟵ n.getBody();
12         infoList ⟵
             infoList ∪ expExtractor(mList, dep − 1);
13     end
14   end
15 end
```

Algorithm 1: $expExtractor$ algorithm

decreases to 0. Since the recursion happens only when there are composite blocks and method invocations, the depth condition guarantees the termination of the algorithm.

Step 3: Calibrate exception handling information. In Step 2, we have collected a list of ExcepInfo_m for each method m by directly analyzing the ASTs. Now we refine them in the following two steps: (1) We remove exceptions irrelevant to the parameter constraints. Namely, for each $(m, P, t, c) \in \text{ExcepInfo}_m$, if none of the parameters in P appear in the condition c, this piece of information is deemed to be irrelevant, hence we update $\text{ExcepInfo}_m := \text{ExcepInfo}_m \setminus \{(m, P, t, c)\}$. (2) For two methods m, n such that $\mathsf{call}(m, n)$, assume furthermore that we have $(m, P, t, c) \in \text{ExcepInfo}_m$ and $(n, Q, t, c') \in \text{ExcepInfo}_m$, which means there is some exception propagated to m from n. In this case, we again traverse the AST of m. If n is enclosed in some try block of m and there is a compatible exception type handled and no new exception is thrown in the *catch* or *finally* statements of m, (n, Q, t, c') is removed from ExcepInfo_m. Otherwise, a new exception is thrown in the *catch* or *finally* statement, and then the related information is recorded and used to update (n, Q, t, c'). Note that this step requires a second traverse of the AST $tree_m$.

Step 4: Classify exception handling information. In this step, we further classify the cleaned exception information generated from Step 3 into the following *categories*, which is used to formulate parameter usage constraints.

(1) Category "**Nullness not allowed**", which consists of exceptions (m, P, t, c) such that c implies $p = null$ for some $p \in P$;
(2) Category "**Type restriction**", which consists of exceptions (m, P, t, c) such that c contains $instanceOf$.
(3) Category "**Range limitation**", which consists of exceptions (m, P, t, c) where some comparison operators exist in condition c unless it is compared with *null*. In that case, (m, P, t, c) would *not* be included.

Note that we do *not* have a category "nullness allowed", as the related constraints cannot be fully handled by the exception conditions; for them, we utilize and adapt the technique proposed in [17]. Similar to [17], we are interested in API methods where a parameter's null value is not prohibited and where the null value has a semantics. This is usually reflected by a particular behavior of the method, for example, to instantiate a default object in case of null value. Therefore, we parse the statements inside the API body. If there is an explicit choice statement, we check whether the condition relates to the null value of the parameter. If the condition handles the case where the value is null and no exception is thrown, we would label it as "nullness allowed". Different from [17], in our approach, the checking process is executed recursively in the case of method invocations where the same parameter is passed to the called method. We set the value of call hierarchy depth the same as for the other three categories in our approach.

Step 5: Constraints generation. We formulate the collected information regarding the parameter usage constraints as a FOL formula Φ_{API}. According to the four types of the parameter usage constraints, we introduce the following *predicates*: (1) $\mathsf{NullAllow}(m, a)$, (2) $\mathsf{NullNotAllow}(m, a)$, and (3) $\mathsf{Type}(m, a, cl)$, where m is a method, a is an argument of m, and cl is a datatype in Java.

Accordingly, for each method m, we generate a formula Φ_m which is a (logic) *conjunction* of

(i) $\mathsf{NullNotAllow}(m, p)$, if p is a parameter of m and (m, P, t, c) is in the "nullness not allowed" category from **Step 4**.
(ii) $\mathsf{NullAllow}(m, p)$, if p is a parameter of m and "nullness allowed" category. For such constraints, there are no exceptions thrown. In this part, we do not consider aliasing problems either.
(iii) $\neg\mathsf{Type}(m, p, cl)$, if p is a parameter of m, and there exists (m, p, t, c) such that

$$c \implies (p = \mathsf{instanceOf}(cl)).$$

(iv) $\bigwedge_{m,p \in P} \bigwedge_{(m,p,t,c) \in \text{ExcepInfo}_m} \neg c$ which specifies the range of each parameter available from the exception information.

2.2 Extract constraints from directives

In this section, we illustrate the workflow of the lower branch in Figure 1. In particular, we describe an approach to extract constraints out of the directives in the API documents automatically. The underpinning observation of this approach is that constraints reported in textual descriptions of API documents usually have specific/recurrent grammatical structures—depending on the constraint category—that share some common characteristics. Consequently, such commonalities can be captured by the notion of heuristics through domain knowledge [22], [23] for enabling the automatic extraction of constraints based on specific natural language processing (NLP) techniques.

Particularly relevant to our approach, JDK's documentation is generated automatically by Javadoc. The content of the document is grouped inside pre-defined delimiters (e.g., /** and */). Standard tags are defined to describe the different pieces of information (for instance, version, author, method signatures, exception conditions, etc.) of the target documents to be generated. Our goal is mainly to detect the defects regarding the parameter constraints and exception declarations of the methods. Javadoc tags provide a useful indicator to extract the relevant textual description from the documentation.

We use NLP techniques, e.g., *part-of-speech (POS) tagging* and the *dependency parsing* to process API documents. In a nutshell, POS tagging is the process of marking up a term as a particular part of speech based on its context, such as nouns, verbs, adjectives, and adverbs, etc. Because a term can represent more than one part of speech in different sentences, and some parts of speech are complex or indistinct, it turns out to be difficult to perform the process exactly. However, research has improved the accuracy of the POS tagging, yielding various effective POS taggers such as TreeTagger, TnT (based on the Hidden Markov Model), or the Stanford tagger [24], [25], [26]. State of the art taggers reaches a tagging accuracy of around 93% when compared to the tagging results of humans.

In the first step, we perform POS tagging with the Stanford *lex parser*[5] to mark all terms of the words and their dependency relation in the constraint-related directives extracted from the documents. In particular, we focus on the sentences annotated with *@param*, *@exception*, and *@throws* tags.

We then pre-process the texts before carrying out dependence parsing. The pre-precessing is necessary because API documentation is usually different from pure natural language narrations (for instance it is frequently mixed with code-level identifiers). The tag headers, i.e., *@param, @exception*, and *@throws*, will be removed, but their type and the following parameter will be recorded. In addition, some embedded markers (such as <*code*>) will be removed, whilst the words enclosed with such markers are recorded, since these are either the keyword or the corresponding variable/method/parameter names in the code. Undoubtedly, there are more complicated cases, making the pre-processing a non-trivial task. A typical situation is that there are code-level identifiers and expressions in the documents. For example, the document of java.awt.color.ColorSpace.getMinValue(int component) states "@throws IllegalArgumentException if component is less than 0 or greater than numComponents -1". To tackle that, we first recognize the special variable names and mathematical expressions via regular expression matching. The naming convention of Java variables follows the *camelcase* style. If an upper case letter is detected in the middle of a word, the word is regarded as an identifier in the method. Likewise, if a word is followed by some mathematical operator, it will be regarded as an expression. Other cases include the identification of method names (with the affiliation class identifier "."), constant variables, etc. Composite statements also need to be divided into simple statements. The statistics of the POS, such as the number of subjects, the number of verbs and the most common verbs are given in the repository of the replication package.[6] We observe that the number of directives with verbs is significantly higher than the ones with subjects. For example, in the latest Android APIs, 6915 directives are with verbs, whereas just 3970 directives are with subjects.

In general, we defined 29 regular expressions and rules to detect these cases. One example to recognize the member functions in the description is of the following form: "\W[A-Za-z_]+[A-Za-z_0-9]*(\.[A-Za-z_]+[A-Za-z_0-9])*(#[A-Za-z_]+[A-Za-z_0-9]*)?\([^()]*\)\W". After the specific identifiers and expressions in the description are recognized, they are replaced by a fresh labeled word to facilitate the dependency parsing.

The dependency parsing and linguistic analysis require identification of certain heuristics. For this purpose, we adapted an approach used in previous work [22]. We note that, however, the adaptation to the new context was not a trivial task. Specifically, different from the work by Di Sorbo *et al.* [22], we are not interested in leveraging the recurrent linguistic patterns used by developers while writing text messages in development emails to automatically infer their intentions (e.g., discussing a bug report or a feature request). Our goal is to leverage the existing *linguistic patterns reported in API documents* to automatically extract/detect API documentation constraints. As consequence, the heuristics/tools provided in the paper by Di Sorbo *et al.* [22], [23] are based on linguistic patterns but not applicable for our purpose. This required us to adapt the concept of linguistic patterns/heuristics definition to the API documentation context. For that reason, we explain in detail the steps required for the definition of the heuristics and its utilization in our context:

1) We perform a (manual) analysis of the existing *linguistic patterns* of constraints described in API documents which admit similar (recurrent) grammatical structures. This step included a manual examination of 459 documents of *java.awt*, *javax.swing* and *javaFX* packages for extracting a set of linguistic patterns according to each of the four constraint types. Specifically, we recognized several discourse patterns related to each of the four constraint types. As a simple example, in javax.swing.UIManager.getFont(Object key), the constraint states that an exception would be thrown "if key is null"; while in java.awt.Component.list(PrintStream out), the constraint states similarly that an exception would be thrown "if out is null." In this case, "is null" is the recurrent pattern and will be extracted, therefore. This manual analysis required approximately 1 week of work.
2) For each extracted linguistic pattern we define an NLP heuristic responsible for the recognition of the specific pattern. The formalization of a heuristics requires three steps: (1) discovering the relevant details that make the particular syntactic structure of the sentence recognizable; (2) generalizing some

5. cf. http://nlp.stanford.edu/software/lex-parser.shtml

6. https://github.com/DRONE-Proj/DRONE/tree/master/Replication/StatisticsOfVerbsAndSubjects

pieces of information; and (3) ignoring useless information. At the end of this process, a group of related *heuristics* constitutes the pattern for a specific constraint category.

At the end of this process, we formalized 67 heuristics (available in the replication package). It is important to mention that, similarly to the work by Di Sorbo *et al.* [22], the accuracy of our approach in identifying API constraints strictly depends on both the accuracy of the aforementioned manual analysis and the overall quality of the API documentation. In particular, as described before, to ensure a proper manual analysis of the recurrent linguistic patterns described in constraints of API documents it required us approximately 1 week of work. Clearly, a poor API documentation with non-standard ways to describe/report API constraints can affect the general accuracy of our results (cf. Section 4 for further discussions). However, adapting the approach to API documents of higher/lower quality may require additional manual analysis.

A brief statistics of heuristics for each constraint type is given in Table 1. Since these heuristics are different from each other, during the linguistic analysis phase, one directive will be accepted by at most one heuristics (possibly none, in case that no constraints are specified). We remark that these heuristics are interesting in their own right, and can potentially be reused and extended in other related researches.

TABLE 1
Summary of Heuristics

Constraints types	Heuristic number
Nullness not allowed	22
Nullness allowed	12
Type restriction	10
Range limitation	23
In total	67

Once the heuristics are identified, we perform dependency parsing and pattern analysis. For this, we largely follow the methodology of [22]. As a concrete example for heuristic-based parsing, the document of java.awt.Choice.add-Item(String item) states "@exception NullPointerException if the item's value is equal to <code>null</code>". We first record the exception type, and then remove the pair of "<code>" and "</code>". The sentence "if the item's value is equal to null" is finally sent to the parser.

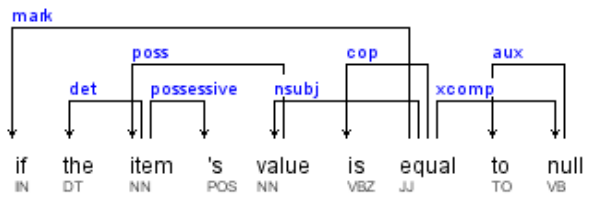


Fig. 2. POS tagging and dependency example[7]

7. The meaning of POS tags and phrasal categories can be found via http://www.cis.upenn.edu/~treebank/

Figure 2 illustrates the dependency parsing result of our example document description. In this sentence, we disregard useless words, such as "if", since its part-of-speech is *IN*, i.e., the proposition or subordinating conjunction. The subject of the sentence (nsubj) is "value", but the value does not appear in the parameter list of the method. We thus check again the neighboring noun (NN), i.e., item, and find it matches the parameter, so we mark it as the subject of the directive. We observe that "equal to" is a recurring phrase that appears in many directives. It indicates an equivalence relation to the following word. The *xcomp* of such phrase—*null* in this case—will be the object of the real subject. We can thus define the language structure with "(subj) equal to null" as heuristics during matching. In this way, the subject(subj) and object("null") of "equal to" will be extracted and be normalized into the expression $subj = obj$. In practice, "[verb] equal to", "equals to" and "[verb] equivalent to" are of the same category, and they will be normalized into the same expression. In this example, the parsing result ends up to $item = null$.

We are now in a position to generate the parameter usage related constraints for the documentation, again represented by a FOL formula. From the previous steps, we have identified the relevant sentences via tagging and dependency parsing, with necessary pre-processing. We further divide these sentences into shorter sub-sentences. In the above example, the sentence is transformed to "if component is less than 0 or greater than [specExpression]". Since *"component"* is parsed as the subject and *"or"* is parsed as *cc* (conjunction in linguistics), the sentence can be further divided into two sub-sentences, i.e., "component is less than 0" and "component is greater than [specExpression]", and then each sub-sentence is subject to the analysis.

As the next step, we define a set of rewriting rules to translate the obtained sub-sentences into FOL formulae. For instance, "or" is rewritten into a logic disjunction, and "less than" is rewritten as a comparison operator $<$. As a result, the above example can be rewritten into $(component < 0) \vee (component > [specExpression])$. Finally, we replace the labeled word by the original expression, yielding the output FOL formula of the procedure. In our example, we have $(component < 0) \vee (component > \text{numComponent} - 1)$.

2.3 Identify defects

Recall that from the preceding two steps, we have obtained two FOL formulae, namely, Φ_{API} and Ψ_{DOC}, over the same set of predicates introduced in the **Step 5** in Section 2.1. Intuitively, they represent the collected information regarding the API source code and the directives of the documents with respect to the four types of parameter usage constraints. The main task of the current step is to detect the mismatch between these two. To this end, our approach is to check whether the two formulae Φ_{API} and Ψ_{DOC} are equivalent. If this is the case, one can be reasonably confident to conclude that all constraints (with respect to the four types of method parameter usage constraints considered in the paper) in the API are captured in documentation and vice verse. If, on the other hand, this is not the case, we will be able to identify the mismatch referring to the relevant predicate, by which we can trace the

method and the parameter thereof, as well as the involved exception. Then we can examine whether such a mismatch is a genuine defect of the API document.

Formally, we make a query to check whether

$$\Phi_{\mathrm{API}} \Leftrightarrow \Psi_{\mathrm{DOC}} \quad (1)$$

holds. If this is indeed the case, we can conclude that the API source code and the related documents are matched. Otherwise (i.e., (1) does not hold), a counterexample is usually returned, suggesting where the mismatching happens. Let us take a simple example of method $f(x)$ with a single argument x. Suppose that, from the API source code, one finds that x must be strictly positive; in this case, we have $\Phi_{\mathrm{API}} = x > 0$. However, from the API document, we only see the statement such as x must be non-negative; in this case, we have $\Psi_{\mathrm{DOC}} = x \geq 0$. Under this circumstance, (1) is instantiated by $x > 0 \Leftrightarrow x \geq 0$, which clearly fails. By tracing the relevant predicate (in this case $x \geq 0$), we can detect the defect of the document and recommend possible repairs which will be discussed in Section 2.4. Note that, when one counterexample is returned, in principle we can only locate one inconsistency. To detect all inconsistencies, we have to update the formulae Φ_{API} and Ψ_{DOC} by removing the relevant part of the inconsistencies (defect) which have been detected before and make the query again to find more counterexamples (and thus further inconsistencies). Such a process must be repeated until no further counterexample is returned. In practice, one counterexample often suggests multiple sources of inconsistencies. Hence, only a small amount of rounds is needed.

To perform the check-in (1), we exploit a Satisfiability Modulo Theory (SMT). In general, SMT generalizes boolean satisfiability by adding useful first-order theories such as equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, etc [27]. An SMT instance is a formula in first-order logic, where some function and predicate symbols have additional interpretations. In practice, to decide the satisfiability (or the validity) of formulae in these theories, one usually resorts to SMT solvers. They have been applied in many software engineering related tasks as diverse as static checking, predicate abstraction, test case generation, and bounded model checking. Z3 [21] from Microsoft Research is one of the most widely used SMT solvers. It is targeted at solving problems that arise in software verification and software analysis, and thus integrates support for a variety of theories. It also provides programming level support in multiple languages, for example, Java and Python. Some important functions, such as creating a solver, checking the satisfiability, managing logic formulae, are provided as APIs for third-party usage.

In our scenario, clearly, (1) is equivalent to checking whether

$$(\Phi_{\mathrm{API}} \wedge \neg\Psi_{\mathrm{DOC}}) \vee (\neg\Phi_{\mathrm{API}} \wedge \Psi_{\mathrm{DOC}})$$

is satisfiable. Hence, off-shelf SAT solvers, such as Z3, can be applied.

We note that, however, in practice, there are some specific cases that need to be handled before checking (1). For instance, some constraints extracted from the code contain method calls (e.g., when they appear in the condition of branching statements), but the code analysis does not further examine the internal constraints of these embedded methods. (For instance, for if (isValidToken(primary) in class MimeType of java.awt.datatransfer, we do not trace the constraints of method isValidToken(primary).) We note that the aim of isValidToken(primary) is to check whether the value of primary is null or not. The document directive also states that an exception is thrown if primary is null. It is not difficult to see that, in these cases, the simple comparison of obtained logic formulae would inevitably generate many spurious defect reports. To mitigate this problem, we mark these constraints, ignore them when checking (1), and thus simply regard them as consistent.

To provide some statistics regarding the FOL formulae generated for conducting the experiments described in Section 3.2, in total there are 4,405 variables in Experiment 1, 4,369 variables in Part 1 of Experiment 2, and 1,150 variables in Part 2 of Experiment 2. However, each formula is simple in that it contains only a small number of variables. Indeed, the median values of variables per formula are all 1 in these experiments. As for the total number of clauses, there are 47 in Experiment 1, 17 in Part 1 of Experiment 2, and 10 in Part 2 of Experiment 2. In general, the formulae contain up to 5 clauses with most of them containing only a single clause.

2.4 Repair Recommendation

Once the defects are identified, the next step is to provide meaningful repair recommendations. As illustrated in Figure 1, the repair recommendation is built (or synthesized) on top of the extracted code constraints (in terms of a FOL formula) and document patterns. Specifically, since we mainly consider four categories of parameter constraints, the *extracted patterns of each category could be reused as the template for generated text*, and we strive to select most concise and informative templates for each category. It is important to mention that recent work in literature proposed the use of templates to document undocumented part of source or test code [28], [29], [30], [31], [32]. However, our scenario is different from the ones of such previous work, as we provide recommendations, based on NLP templates, for replacing, correcting API defects with appropriate repairing solutions, and thus complementing the available human-written documentation. Moreover, different from such previous work, *DRONE* generates such templates by analyzing both extracted code constraints (in terms of a FOL formula) and document patterns.

Table 2 illustrates some example templates. For categories "Nullness not allowed" and "Nullness allowed", it is straightforward to generate atomic repair directives given the FOL formula. That is, we generate a directive stating the corresponding parameter must not or could be null based on the templates.

In particular, for the category "Nullness not allowed", we append the directive after the tag "@throws" (resp. the tag "@param") for "Nullness not allowed" (resp. "Nullness allowed"). It is slightly complicated in case of composite FOL expressions since multiple atom formulae exist. To give more concise and meaningful recommendations, we combine the subjects of the elementary atomic sentences and yield a single

sentence for "Nullness not allowed". For example, in javax.xml.bind.JAXBElement.JAXBElement(QName name, Class<T> declaredType, Class scope, T value), the extracted FOL is Or(NullConstraint(declaredType; NEG); NullConstraint(name;NEG)). Accordingly, the generated document is "@throws IlleagalArgumentException If declaredType or name is null". Meanwhile, for the category "Nullness allowed", since each parameter has a separate tag, the atomic directive will be directly generated and appended after their corresponding tags based on the templates.

For the "type restriction" category, the simplest case would be to add the type restriction information to the exception related directives. We need to add the concrete type information to replace [SpecType] as shown in the template apart from the corresponding parameter name as shown in Table 2. Note that, despite that our first example indicates the parameter should not be of a particular type, in some other cases, some particular type of parameter is expected. In such case, we add "not" to specify this fact. For example, in java.security.UnresolvedPermissionCollection.add(Permission permission), the parameter permission is supposed to be the type of UnresolvedPermission, otherwise an IllegalARgumentException would be thrown. But this information is missing in the document, and *DRONE* will detect this and recommend the repair "@throws IllegalArgumentException If permission is not the type of UnresolvedPermission".

For the "range limitation" category, we define a set of translations for the numerical relation. For example, ">" will be translated to "greater than", and "==" translated to "equal to". There are multiple templates for this category as summarized in Table 2. As a concrete example in java.security.Signature.update(byte[] data, int off, int len) from JDK 1.8, *DRONE* detects the incompleteness of the documentation and recommends the repair as "@throws IllegalArgumentException If len or off is less than 0" following the template "If [param1] or [param2] {relation} [value]". In the case of Boolean value comparison, we just use "be TRUE" or "be FALSE" instead of "be equal to TRUE" or "be equal to FALSE".

TABLE 2
Repair recommendation templates

Constraints types	Tags	Templates
Nullness not allowed	@throws	If [param] be null If [param1] or [param2] be null If [param1], ... , or [paramN] be null
Nullness allowed	@param	[param] could be null
Type restriction	@throws	If [param] be type of [SpecType] If [param] be not type of [SpecType]
Range limitation	@throws	If [param] {relation} [value] If [param] {relation} [value1],...,[valueN] If [param1] or [param2] {relation} [value] If [param] {relation} [value1] and {relation} [value2] If [param] {relation} [value1] or {relation} [value2]

3 Implementation and Case Studies

We have developed a prototype that implements the approach described in Section 2, based on the Eclipse plugin architecture. In this section, we will present the main graphical user interface of our prototype, while we evaluate its effectiveness in various case studies, as described in Section 3.2. Specifically, the effectiveness is mainly judged in terms of precision and recall.

The prototype takes API code and document directives as inputs, and outputs repair recommendations for directive defects. Particularly, the tool integrates the SMT solver Z3 to analyze generated FOL expressions written in the SMT-LIB 2.0 standard[8] and identify potential defects. Figure 3 and Figure 4 illustrate the main graphical user interfaces of the prototype.

Figure 3 demonstrates the *defect detection* view. The left panel displays the API packages to be explored by *DRONE*. The tool can be invoked by selecting it in the menu opened after right-clicking the corresponding package. The tab view of *DRONE* includes (i) *invocation relation analysis*, (ii) *code parsing and document analysis*, and (iii) *defect detection*. The console on the right part displays the execution traces, which also provide configuration support for saving intermediate files during analysis.

Figure 4 demonstrates the *defect repair* view. The list in the middle of the panel displays the detected defects from the previous step. Once a listed item is left-clicked, *DRONE* allows to automatically navigate to the corresponding defect API, which will be highlighted in color. Meanwhile, the repair recommendations will be given on the right part of the tab panel.

3.1 Settings

We conduct three experiments to evaluate our prototype implementation. In Experiment 1, we focus the evaluation on the packages *awt*, *swing* and *javaFX*. In Experiment 2, we reuse the heuristics defined in the first one and evaluate the performance for twelve additional packages in the same project (Part I) and the latest Android APIs (Part II). Finally, in Experiment 3, we evaluate the quality of generated documentation recommendations for detected defects. In all experiments, the evaluations are conducted on a PC with an Intel i7-4790 3.6 GHz processor and 32.0 GB RAM, running Windows 7 64-bit operating system. The depth of call hierarchy is set to 4.

The metrics used in the first two experiments are *precision*, *recall*, and *F-measure*. Precision measures the exactness of the prediction set, whereas recall measures the completeness, which are respectively calculated as follows.

$$precision = \frac{TP}{TP + FP} \tag{2}$$

$$recall = \frac{TP}{TP + FN} \tag{3}$$

F-measure considers both exactness and completeness, and thus balances the precision and recall.

$$F\text{-}measure = 2 \times \frac{precision \times recall}{precision + recall} \tag{4}$$

where TP, FP, FN stand for true positive, false positive, and false negative respectively.

In the third experiment, we hire twenty-four graduate students majoring in Software Engineering to evaluate the generated documentation recommendations. Similar to the

8. cf. http://smtlib.cs.uiowa.edu/

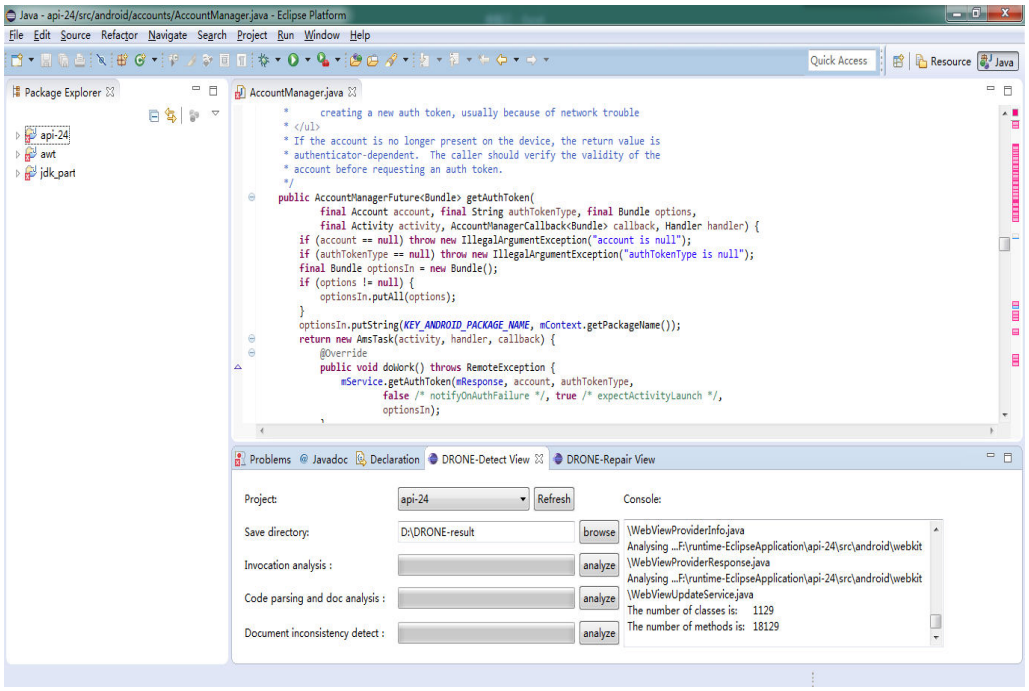


Fig. 3. Prototype overview—defect detection

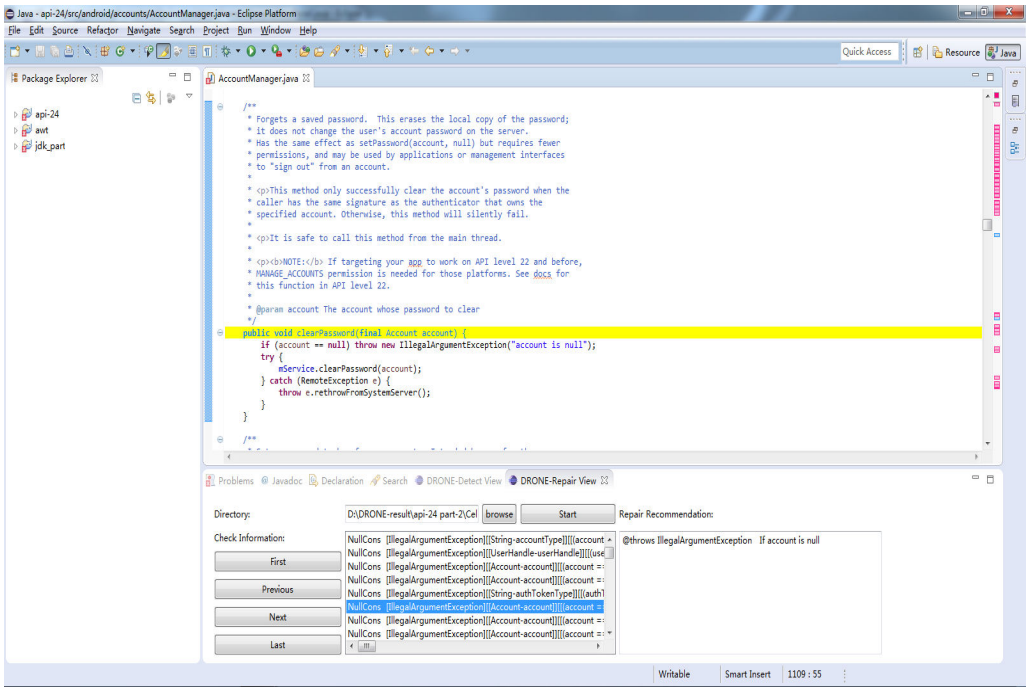


Fig. 4. Prototype overview—defect repair

previous work [29], [32], [33], [34], we primarily consider three aspects, i.e., *accuracy*, *content adequacy*, *conciseness* and *expressiveness*, as the main criteria to assess the quality of the recommendations.

3.2 Results

RQ1: ***To what extent does DRONE identify directive defects in Java/Android API documentation?***

Experiment 1

We first evaluate the performance of our approach applied to the target packages and their documents (i.e., *java.awt*, *javax.swing* and *javaFX*). The packages parsed by our prototype contain around 1.2 million lines of code (LoC)[9] and 25,168 Javadoc tags in total. The details are summarized in Table 3. Over these dataset, the program analysis process takes around two hours, while the document analysis takes around 1 hour. Finally, our approach outputs 2510 constraints for the APIs methods.

To calculate the precision and the recall, the ground truth set is required. For this purpose, three computer science master students are hired, who have more than

9. The statistics includes comments and space.

three years of Java development experience, and are asked to manually inspect the obtained results, classifying the items into true/false positives and true/false negatives. In terms of recall, in principle, the *total false negatives are required*. However, it turns out that manual examination of all involved APIs and their documentation (25,168 Javadoc tags) would be practically impossible. In particular, the tremendous number of inter-procedure invocation makes the manual process both error-prone and time-consuming. Therefore, we only consider those APIs with the constraints detected by our tool as the sample. This means that for what concerns true positives and false positive we rely on the validation performed by the involved master students. We apply a *stratified random sampling strategy* to examine 10% of the APIs and their documentation outside of the set and only very few (less than 1% of them) are missing. Each report was examined by three subjects independently. A majority vote mechanism was used to resolve possible conflicts. The manual classification process required around five days.

TABLE 3
Data overview in Experiment 1

Package names	LoC (kilo)	@param No.	@throws No.	@exception No.
java.awt	178.8	5383	961	423
javax.swing	372.8	8531	448	533
javaFX	625.4	7832	1040	17
Total	1177.0	21746	2449	973

The results of Experiment 1 are summarized in Table 4. Overall, out of these reported 1689 defects (TP+FP), 1291 turn out to be real defects, giving rise to a precision of 76.4%. Combined with 398 false negatives, we get a recall of 83.8%. The average F-measure is 79.9%. In particular, all of the four defective document examples from JDK mentioned in Section 1 are detected successfully. Our approach performs well on the selected API packages where the heuristics are summarized. Moreover, Table 4 also gives the distribution and performance of each constraint category. *Range limitation* category takes up the largest portion of defective documentation in the selected dataset.

TABLE 4
Results of Experiment 1

Category	TP	FP	FN	TN	Precision	Recall	F-measure
Nullness Not Allowed	233	77	7	333	0.752	0.971	0.847
Nullness Allowed	599	63	27	34	0.905	0.957	0.930
Range Limitation	406	245	121	198	0.624	0.770	0.689
Type Restriction	53	13	95	6	0.803	0.358	0.495
Total	1291	398	250	571	0.764	0.838	0.799

Among these four constraint categories, we found the precision for the *range limitation* type and the *nullness not allowed* type is lower than the other two types. We then examined some false positives: for the *range limitation*, most false positives are attributed to some vague descriptions of the parameter range. For example, in java.awt.Container.java, the extracted constraint for add(Component comp, int index) from the API code is: $(index < 0 \wedge \neg(index = -1))$, which is propagated from the callee method addImpl(int). But the document directive just states "@exception IllegalArgumentException if the index is invalid." Some other similar vague descriptions are also frequently found, for example, simply been stated "out of range." Such implicit expressions prohibit the effective extraction of constraints and are deemed to be "defective" in our approach. To mitigate this issue, we can define some specific rules to rectify, i.e., treating such cases as non-defective.

On the other hand, there are some opposite cases where the descriptions are concrete, but difficult to resolve. For example, in java.awt.Container.areFocusTraversalKeysSet(int id), the document states that "if id is not one of KeyboardFocusManager.FORWARD_TRAVERSAL_KEYS, KeyboardFocusManager.BACKWARD_TRAVERSAL_KEYS, [...]", an IllegalARgumentException will be thrown. The document enumerates all of the valid values for the parameter id. But in the code, the condition for the exception is just $id < 0 \vee id \geq \mathrm{KeyboardFocusmanager.TRAVERSAL\_KEY\_LENGTH}$. In this case, since our current implementation does not interpret the constant values, we cannot detect either. But such false positive can be reduced by augmenting with more reference abilities via static analysis tools which is planned in our future research.

The *nullness not allowed* type suffers from the similar issue as *range limitation*. The slight difference we observe is the existence of some anti-patterns in the documentation. For example, the document of java.awt.Choice.insert(String item, int index) states "@param item the non-null item to be inserted". The linguistic feature of such directive is quite different from what we summarized before, and our approach does not successfully extract the constraints. But we could get around the problem by adding more such "anti-pattern" heuristics into our repository.

We also manually analyzed some false negatives reported by our experiment and found that many are introduced by the method calls embedded in the condition statements. To reduce the false positives, we skipped the constraints inside these embedded methods, and simply regard the accompanying documents as non-defective. This, however, is a double-edged sword, i.e., false negatives are also potentially introduced. For example, in java.awt.image.AffineTransformOp.AffineTransformOp(AffineTransform xform, [...]), the method invokes validateTransform(xform), and thus the constraint "Math.abs(xform.getDeterminant())<=Double.MIN_VALUE" can be extracted. This constraint is marked and skipped. whilst the document is considered to be sound (cf. Section 2.3). However, unfortunately, the document directive of xform is just "the AffineTransform to use for the operation", which is defective because it does not provide sufficient information, and indeed is found manually. This causes a false negative. In general, we strive to achieve a trade-off between false positive and false negative, but more precise program analysis would be needed which is subject to further investigation.

Experiment 2

In this study, we extend the exploration scale to cover more Java API libraries and different projects. In particular, we consider additional twelve JDK packages (i.e., *javax.xml*, *javax.man-agement*, *java.util*, *java.security*, *java.lang*, *java(x).sql*[10], *java.imageio*, *java.io*, *java.net*, *java.nio*, *java.text* and *java.time*) and the latest Android SDK API (level 24). We separate this experiment into two parts. The first part involves the twelve JDK packages, while the second involves the Android API. We reuse the heuristics from the first experiment. The information of these packages is given in Table 5 and Table 6.

TABLE 5
Data overview for additional JDK packages (Part 1)

Package names	LoC (kilo)	@param No.	@throws No.	@exception No.
javax.xml	61.4	1654	1031	141
javax.management	71.5	1503	295	822
java.util	212.1	4965	2547	290
java.security	41.1	908	164	421
java.lang	89.1	1732	754	335
java(x).sql	45.7	2016	610	1338
java.imageio	25.7	744	2	735
java.io	31.8	647	281	369
java.net	33.5	522	165	255
java.nio	71.3	813	1212	0
java.text	22.6	433	5	151
java.time	56.9	1470	975	0
Total	762.7	17407	8041	4857

TABLE 6
Data overview for Android SDK API (Part 2)

Package names	LoC (kilo)	@param No.	@throws No.	@exception No.
android.accounts	6.0	205	25	0
android.bluetooth	28.1	434	58	0
android.database	15.3	548	87	0
android.gesture	3.0	49	0	0
android.graphics	40.8	1226	43	0
android.hardware	74.0	948	246	0
android.location	12.3	172	109	0
android.media	93.3	1171	496	0
android.net	60.2	715	119	2
android.service	14.2	205	15	0
android.view	113.3	2357	151	1
android.webkit	11.9	320	4	0
Total	472.4	8350	1353	3

For the second experiment, again we ask the same subjects, as in the first one, to manually classify the obtained results and use majority vote to resolve possible conflicts. Table 7 summarizes the performance details for each constraint category in Part 1 of experiment.

Out of these 1605 detected defects, 953 turn out to be true positives, and 652 false positives, giving a precision rate of 59.4%. Taking the 102 false negatives, we get a recall of 90.3%. Similar to the observations of Experiment 1, the precision of the *range limitation* has the lowest value among the four. Overall, the performance in terms of precision and F-measure is lower than that of the first experiment, but still at an acceptable level. Based on the obtained results, we observe that, when our heuristics are applied to other APIs, although suffered at a decrease in the accuracy, the performance is still kept at an acceptable level with a precision of 59.4% and a recall of 90.3%, and thus these heuristics can be reused.

10. It contains both *java.sql* and *javax.sql*.

TABLE 7
Results of additional JDK packages (Part 1)

Category	TP	FP	FN	TN	Precision	Recall	F-measure
Nullness Not Allowed	405	158	27	842	0.719	0.938	0.814
Nullness Allowed	122	49	8	45	0.713	0.938	0.811
Range Limitation	374	422	59	338	0.470	0.864	0.609
Type Restriction	52	23	8	1	0.693	0.867	0.770
Total	953	652	102	1226	0.594	0.903	0.717

TABLE 8
Results of Android SDK API (Part 2)

Category	TP	FP	FN	TN	Precision	Recall	F-measure
Nullness Not Allowed	246	38	15	82	0.866	0.943	0.903
Nullness Allowed	89	19	3	4	0.824	0.967	0.890
Range Limitation	123	93	35	66	0.569	0.778	0.658
Type Restriction	6	7	2	2	0.462	0.750	0.571
Total	464	157	55	154	0.747	0.894	0.814

Table 8 summarizes the results of Part 2 of Experiment 2. Out of 621 detected defects, 464 are turn out to be true positives, and 157 turn out to be false positive, giving a precision of 74.7%. Again, both the illustrative defective examples from Android in Section 1 are successfully detected by *DRONE*. With 55 false negatives, we get a recall rate of 89.4%. Slightly different from the findings of the experiments for JDK, the performance of *range limitation* category is the second lowest among the four. Totally, the F-measure of this experiment is 81.4%, which further demonstrates the feasibility of *DRONE* on the selected Android APIs.

One of the purposes of carrying out these experiments, especially those for Android SDK API, is to examine the generalizability of the heuristics we formulated in the first experiment. With the heuristics developed solely based on a limited number of JDK packages, it is natural to raise the concern regarding overfitting of those heuristics to the considered packages. As one might see, regarding the generalizability of the heuristics to other JDK packages, the precision is down from 76.4% to 59.4%. This is possible because of overfitting, but the loss of the precision is acceptable. Regarding the generalizability to Android SDK API, we only observe a marginal loss of precision (76.4% to 74.7%). This is slightly surprising as the heuristics appear to generalize well to a different project. We speculate that it might be the case that Android SDK APIs we are examining are similar to the packages of the first experiment. From Experiment 2, we might conclude that the generalizability of the heuristics is reasonably well.

RQ2: ***To what extent does DRONE provide coherent repairing solutions for the detected API documentation defects?***

Experiment 3

The main purpose of this experiment is to answer **RQ2**, i.e., to demonstrate the quality of the generated API documentation recommendations by *DRONE*. As mentioned, we mainly use *accuracy, content adequacy*, and *conciseness & expressiveness* as the criteria to assess the quality of repair recommendation.

- *Accuracy* mainly concerns the correctness of generated constraints descriptions.
- *Content adequacy* only considers the content of the generated solutions: is the important information about the API implementation (related to its directives) reflected in the generated descriptions? Thus, content adequacy is mainly about whether the constraints are all included.
- *Conciseness & expressiveness* refers to whether unnecessary information is included in the rendered text: is there extraneous or irrelevant information included in the descriptions, and is the text easily understandable?

Based on the above quality aspects, we design multiple questions, shown in Table 9, for the evaluators. The first two questions address *accuracy* and *content adequacy* respectively, whereas the third and the fourth question address conciseness, and expressiveness. The answers of the questions can be among "strongly agree", "agree", "neutral", "disagree", and "strongly disagree", corresponding to five discrete levels, ranging from 5 to 1 respectively.

TABLE 9
Questions designed to evaluate repair recommendation

	Questions	Value Range
Q1	Does the repair recommendation reflect the code constraints? (Accuracy)	(5-1)
Q2	Is the repair recommendation helpful to better understand and use the API? (Content adequacy)	(5-1)
Q3	Is the repair recommendation free of other constraint-irrelevant information? (Conciseness)	(5-1)
Q4	Is the repair recommendation clear and understandable? (Expressiveness)	(5-1)

This part of the experiment includes 24 graduate students majoring in Software Engineering as subjects. Among them, eighteen were from Nanjing University of Aeronautics and Astronautics, five were from University of Zurich, and one was from University of Sannio. All subjects have experience of programming in Java and Android for at least five years. We randomly selected 160, 140, and 100 generated directives from the previous experiments respectively (in total of 400 directive samples), together with the corresponding source code and related constraints. The source code also includes the invocation chains if there is a relevant parameter constraint between the caller and callee methods.

Listing 1 gives a concrete example of the provided samples, taken from com.sun.javafx.robot.FXRobotImage.java. We first list the generated repair recommendation (Line 1-3), followed by the API's comments and codes (Line 4-15). The comments correspond to the related API directives annotated with specific tags (Line 4-9). The parts of parameter related constraints whose document is missing are also present inside the body of the code (Line 11-13). Each sample is organized in the same format to the participants.

Listing 1. Sample example

```
1  Check Recommendation:
2  @throws IllegalArgumentException If x or y is less than 0,
3  x is no less than width or y is no less than height.
4  /**
5  ...
6  * @param x coordinate
7  * @param y coordinate
8  ...
9  */
10 public int getArgb(int x, int y){
11 if (x < 0 || x >= width || y < 0 || y >= height) {
12 throw new IllegalArgumentException(...);
13 } ...
14 }
```

We first introduce the background to the subjects, so they can have sufficient understanding of the requirements. We then assign the 400 samples to the 24 subjects. To ensure that each repair recommendation is reviewed by at least three subjects, we classify them into 8 groups, with 3 participants each group. Accordingly, we assign 50 samples to each group, and each participant is supposed to answer all the four questions in Table 9 independently.

TABLE 10
Results of Experiment 3

Package Source	Result	Q1	Q2	Q3	Q4
Case 1	5	**337(70.2%)**	**211(44.0%)**	**342(71.3%)**	**228(47.5%)**
	4	87(18.1%)	123(25.6%)	79(16.5%)	151(31.5%)
	3	28(5.8%)	91(19.0%)	37(7.7%)	73(15.2%)
	2	10(2.1%)	40(8.3%)	4(0.8%)	9(1.9%)
	1	18(3.8%)	15(3.1%)	18(3.7%)	19(3.9%)
Case 2	5	**316(75.2%)**	**146(34.8%)**	**325(77.4%)**	**281(66.9%)**
	4	47(11.2%)	36(8.6%)	49(11.7%)	79(18.8%)
	3	13(3.1%)	82(19.5%)	14(3.3%)	25(6.0%)
	2	16(3.8%)	125(29.7%)	11(2.6%)	8(1.9%)
	1	28(6.7%)	31(7.4%)	21(5.0%)	27(6.4%)
Case 3	5	**245(81.7%)**	**201(67.0%)**	**236(78.7%)**	**224(74.7%)**
	4	19(6.3%)	24(8.0%)	30(10.0%)	31(10.3%)
	3	3(1.0%)	22(7.3%)	5(1.7%)	7(2.3%)
	2	5(1.7%)	23(7.7%)	15(5.0%)	13(4.3%)
	1	28(9.3%)	30(10.0%)	14(4.7%)	25(8.4%)

TABLE 11
Average Results of Experiment 3

Package Source	Q1	Q2	Q3	Q4
Case 1	4.49	3.99	4.51	4.17
Case 2	4.45	3.34	4.54	4.38
Case 3	4.49	4.14	4.53	4.39
Average	4.48	3.82	4.53	4.31

The review process took around 2 hours. We collected the answers and then performed basic statistics. The results are shown in Table 10 and Table 11. Table 10 gives the experiment result, and the percentage distribution is given by Figure 5. We find that, for all the studied cases, the number of results above 3 takes a majority percentage. Table 11 gives the average scores of the three cases. The scores are also visually compared in Figure 6. Across these three cases, we found that most answers of the four

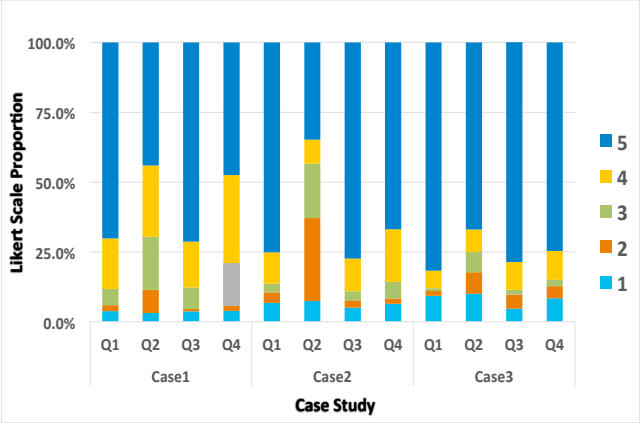


Fig. 5. Percentage Distribution in Experiment 3

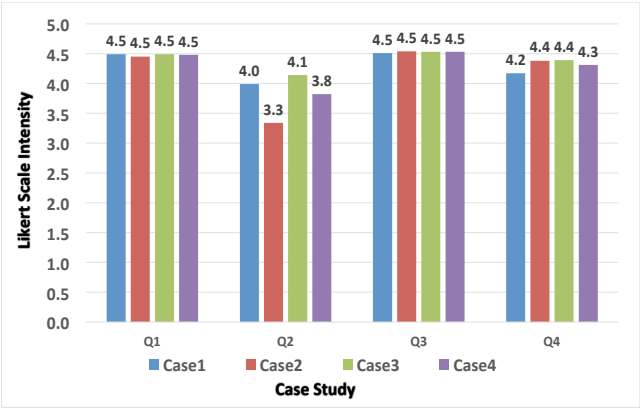


Fig. 6. Comparison in Experiment 3

questions are higher than four, demonstrating good quality of the recommendations returned by our approach. Since the samples contain both true positives and false positives, the average score of Q2 is the lowest (3.82) among the four questions, albeit it is still above 3 (neutral). Actually, during the study we found out that some false negative examples do *not* necessarily mean the bad quality of the generated directives. As an example in javaFX, the original directive of the method com.sun.javafx.geom.Arc2D.setArcType(int type) is "@throws IllegalArgumentException if type is not 0, 1, or 2." The corresponding extracted code constraint is "type $<$ OPEN || type $>$ PIE". Here, in the API, the values of constant variables OPEN and PIE are 0 and 2 respectively. Since we do not map the constant to its value, *DRONE* considers it to be a defect and generates repair recommendation "@throws IllegalArgumentException If type is greater than PIE or less than OPEN." A majority of participants think this recommendation is better than the original one.

Finally, the subjects of the experiment also provided us with some comments for improving the general quality of solutions generated by *DRONE*. Specifically, several participants proposed to generate more elaborated descriptions/templates for "if statement" with many conditions. Indeed, such cases exist, although the number is marginal. For example, in javafx.scene.chart.BarChart.BarChart(Axis xAxis, Axis yAxis, ObservableList data), the parameter constraint in the code is "if(!((xAxis instanceof ValueAxis && yAxis instanceof CategoryAxis) || (yAxis instanceof ValueAxis && xAxis instanceof CategoryAxis)))", the generated repair is: "@throws IllegalArgumentException If yAxis is not the type of CategoryAxis or xAxis is not the type of ValueAxis and xAxis is not the type of CategoryAxis or yAxis is not the type of ValueAxis". We agree that such a repair recommendation is a bit difficult to understand, and we may shorten the sentence by combining the phrases with the same subjects to increase the expressiveness.

3.3 Threats to Validity

3.3.1 *Internal Validity*

Internal validity focuses on how sure we can be that the treatment actually caused the outcome [35]. In our approach, we directly work on the API code as well as the accompanying documents of the Java libraries considered in our dataset. The exception-related constraints are therefore solely extracted from the code (via static analysis techniques) and the descriptions (via NLP techniques). Another concern is the potential bias introduced in the data set. To minimize this threat, we randomly select the packages from the latest JDK and Android libraries and exclude those of private methods. We also exclude those API descriptions with obvious grammatical mistakes. Furthermore, for the evaluation of the approach, we rely on the judgment of computer science master students because there is a certain level of subjectivity in classifying the items into true/false positives and true/false negatives. To alleviate this issue we built a truth set based on the judgment of three inspectors. Moreover, to validate the items each report is examined by three subjects independently. After the initial validation phase, all disagreements were discussed and resolved using a majority vote mechanism.

3.3.2 *External Validity*

External validity is concerned on whether the results can be generalized to the datasets other than those studied in the experiments [35]. To maximize the validity of this aspect, we include additional datasets with source code and API documentation from twelve other packages of JDK and Android libraries. However, as an inherent issue in other empirical studies, there is no theoretical guarantee that the detection strategy still enjoys high accuracy in other projects, especially for those with anti-pattern document writing styles. Nevertheless, we believe the general methodology is still valid in these cases, since our approach for the document constraints extraction is heuristic based, which means new, domain-specific styles can be handled by introducing extra heuristics to boost the performance. There is another concern with the overfitting of such heuristic-based approach. Theoretically, if we exhaustively include all the directives in the studied subjects, it will achieve a very high precision rate. But, on one hand, due to the large set of documents, it is impractical to manually analyze all such directives; on the other hand, it is not necessary due

to the naturalness of language. Therefore, we only include those recurrent linguistic patterns we found in the empirical study and leverage them as heuristics. Our goal was to observe whether our approach is capable of finding defects in well documented APIs. Indeed, all cases considered in our experiments are from the latest versions of JDK and Android. Although they are generally regarded as well-documented APIs, many defects are still present. Finally, to further reduce the threats mentioned above we plan for future work to extend our study by analyzing APIs of libraries of further domains and/or program languages.

4 Discussion

For program analysis, we just consider the explicit "throw" statements as sources of exceptions (i.e., checked exceptions). It is possible that other kinds of runtime exceptions occur during the program execution, for example, divide-by-zero. In most cases, these implicit exceptions are caused by programming errors, so it might be inappropriate to include them in the documentation [36]. As a result, we adopt a similar strategy as in [37] and suppress implicit exception extraction. For static analysis tools, we use Eclipse's *JDT* and *CallHierarchy* mainly due to their ability to parse programs with incomplete reference information. Some related work, such as [37], utilizes the *Soot* toolset [38], which requires complete type class references.

In the analysis of API documents, we only consider the directive statements which are preceded by *@param*, *@throws* and *@exception* tags. Moreover, we only consider a subset of the whole API directives, i.e., the tags whose related codes have detected constraints. In some exceptional cases, the constraints are instead given in the general description part of the methods; these constraints cannot be extracted by our approach. The inclusion of additional parts of documents is left as future work. Moreover, in the document descriptions, very rarely grammatical errors exist which would potentially interfere with the dependency parsing. For example, in javax.swing.plaf.basic.BasicToolBarUI.paintDragWindow (Graphics g), the document directive states "@throws NullPointerException is g is null." Obviously, the first "is" in the sentence is a typo (should be "if"). Another example is that, in the construction method of java.awt.event.MouseEvent, "greater than" is mistakenly written as "greater then". For APIs with such grammatical mistakes, they are removed from the analysis once found.

There are cases in which a few extracted constraints are composite and cover more than one category. For example, as to java.awt.Dialog.Dialog(Window owner, String title, ModalityType modalityType), the extracted constraint of owner is "(owner!=null)&&!(owner instanceof Frame)&&!(owner instanceof Dialog)", which is related to both the *nullness* and the *type*. We classify the composite constraints into more than one category.

One of the goals of our study is to demonstrate the wide existence of API document defects, even in those generally believed well-documented APIs. However, although we have come up with heuristics for JDK libraries which have proven to be effective, there is no formal guarantee that the same heuristics will work equally well with other libraries. Nevertheless, the approach presented in this paper is able to find documentation defects and propose repair solutions for both JDK and Android libraries. More importantly, *DRONE* is essentially open to incorporate other heuristics to facilitate the NLP process. Our heuristics for JDK are valuable for at least two reasons: (1) JDK and Android have a huge user base and (2) our work, as the first work of this kind, sheds light on how developing heuristics for other libraries of (also) further program languages.

We also note that the use of new annotations can help reduce the occurrence of null-related directive mistakes, i.e., @NonNull and @Nullable. Some IDEs, such as IntelliJ IDEA[11] and Android Studio (which rides on IntelliJ), can help detect such violations. For example, if a null value is passed as an argument to an API whose corresponding parameter is annotated with @NonNull, the IDE will issue a warning. Indeed, in our exploration of the Android case study, such tags faithfully reflect the null value constraints of the API code. However, there are at least three reasons for our approach to be useful in such scenarios. First, in many cases, developers tend to forget to add such annotations to the relevant parameters. In our case study on the selected Android APIs, out of 495 null related parameters only 69 are annotated (67 *@NonNull* and 2 *@Nullable* respectively). Second, even these annotations were added, there might still be inconsistencies. For instance, in Android, the parameter transition of android.graphics.drawable.AnimatedStateListDrawable.addTransition(int fromId, int toId, @NonNull T transition, boolean reversible) is annotated with @NonNull, specifying that the value should not be null. However, its documentation states that "transition, ..., may not be null", which is obviously a defect. Last but not the least, even appropriately annotated initially, the caller method which passes the parameter to the API might be forgotten to annotate and therefore such null related information is lost along the invocation chain.

The concept of document defect in our research is based on the assumption that the API code is reliable. This assumption can be—and should be—relaxed in situations when the code quality is relatively low. Clearly, our approach can be adapted to report the *inconsistency* between code and documentation. Still, our approach targets reliable API code as such.

Current approaches based on linguistic analysis patterns are mostly conceived for analyzing few traditional sources of information (e.g., email and APIs documents). However, when performing development, maintenance and testing tasks developers access to various types of heterogeneous data. Thus, future research approaches should be designed with advanced mechanisms able to analyze relevant knowledge present in different sources of information depending on the specific task the developers are performing. For instance, to find *poor* quality documentation in both production and test code. Therefore, future research in SE can be devoted to use linguistic analysis patterns to handle further research challenges, this to make the novel techniques proposed in this paper applicable in many industrial and open source organizations.

11. http://www.jetbrains.com/idea/

5 Related Work

A majority of work on defect detection has been done at the code level whereas fewer studies focus on the document level. In view of the significant role of documentation in program understanding, we believe high quality accompanying documents contribute greatly to the success of software projects and thus deserve more considerations. In this section, we mainly review some representative related efforts in improving the quality of API documentation.

Analysis and Evolution of Directives in API documentation. Directives of API documentation and the evolution of API documentation are studied in [20], [39] and [40], [41] respectively. The authors identified the importance of directives of API documentation and gave a taxonomy of 23 different kinds [20]. We concentrate on a subset of these, i.e., those related to parameter constraints. The co-evolution characteristics of API documentation across versions with the source code was studied quantitatively [40]. Based on the study, most evolution revisions occur in annotations, which also motivates us to concentrate on the parameter-related annotations in our study. Moreover the evolution history of 19 documents was qualitatively studied by analyzing more than 1'500 document revisions [41]. It was observed that updating documentation with every version change could improve the code quality, which, in turn, suggests the positive correlation between documentation and code. [42] investigated the developers' perception of *Linguistic Anti-patterns*, i.e., the poor practices in the naming, documentation, and choice of identifers in the implementation of an entity. The results indicate that developers perceive as more serious those instances where the inconsistency involves both method signature and comments (documentation), and thus should be removed. In a recent survey of API documentation quality conducted by Uddin and Robillard [15], three kinds of problems were regarded as severest, i.e., ambiguity, incompleteness and incorrectness, two of which are considered in our approach. However, all these works adopted an empirical methodology to investigate the problem and no automated techniques were applied.

Zhong and Su [43] proposed an approach that combines NLP and code analysis techniques to detect API documents errors. The errors in their work differ significantly from ours, in that they focus on two types of errors, i.e., grammatical errors (such as an erroneous spelling of words), and broken code names (which are referred to in the documents but could not be found in the source code). In contrast, we target at the incomplete and incorrect descriptions about the usage constraints of the documentation. Thus the emphasis of our work is more at the semantic level. The work by Goffi *et al.* [44] proposed a technique that automatically creates test oracles for exceptional behaviors from Javadoc comments. Similarly to *DRONE*, the technique proposed uses NLP techniques. Different from our work, Goffi *et al.* use run-time instrumentation. Indeed, we exploit static code analysis to extract the statements of control flow decisions and exception handling, as well as the call invocation relation between methods. Moreover, our approach is able to detect a wider set (or types) of documentation defects (the inconsistencies reported by the SMT solver), providing repair recommendations.

Automatic Generation of API Documentation and other Artifacts. There is another thread of relevant research on applying the NLP techniques to documents or even discussions in natural language [28] to infer properties [45], [46] such as resource specifications [47], method specifications [48], code-document traceability [49], document evolution/reference recommendation [50], [51], API type information [52], problematic API features [53], or change requests based on user reviews [54], [55], [56], [57], [58]. In this context, the more close research to our work is the one related to the automatic inferring API documentation from source code [59], [60], [61], [62], [63], [64], [65]. Indeed, inferring API documentation from the code can help to directly compare the existing documentation and finding defects. Specifically, Reiss and Renieris *et al.* [59] proposed a research approach to understand and visualize the dynamic behavior of large complex systems. This work represents a first step towards the generation of documentation from source code or other software artifacts [29], [10], [13], [28]. Following the same line of research, Lo *et al.* [61], [60] proposed a technique based on scenario-based slicing, which is able to extract expressive specifications from real programs. A more recent work has shown that, compared to other strategies, *synoptic graphs* improve developer confidence in the correctness of their systems, which is useful also for finding bugs [62], [63].

Ghezzi *et al.* [64] presented an approach that can infer a set of probabilistic Markov models, concerning data related to users' navigational behaviors, to help understand whether a Web application satisfies the interaction requirements of thousands if not millions of users, which can be hardly fully understood at design time. Finally, a recent work by Ohmann *et al.* [65] proposed Perfume, an approach to capture key system properties and improve system comprehension (by inferring behavioral, resource-aware models of software systems from logs of their executions), thus highlighting the differences between what developers think systems do and what they actually do. Buse and Weimer [37] proposed an automated API documentation generation technique for exceptions. The work leveraged static analysis to extract the call graphs and exception throwing conditions automatically, which overlaps somehow with ours in the aspect of program analysis on exceptions. However, the authors did not consider the extant documents. Instead, they generated new documents based on the program analysis results.

All the work demonstrated the feasibility of applying NLP techniques to documentation, but did not deal with the defect detection. Finally, it is important to mention that other recent work in literature proposed the use of NLP templates to document undocumented part of source or test code [28], [29], [30], [31], [32]. However, our scenario is different as we provide recommendations, based on NLP templates, for replacing and correcting API document defects with appropriate repairing solutions thus, complementing the available human written documentation. Moreover, *DRONE* generates such templates by analyzing both extracted code constraints (in terms of an FOL formula) and document

patterns.

Constraints and Directives in API Documentation. Saied et al. [17] conducted an observational study on API usage constraints and their documentation. They selected four types of constraints, which are the same as ours. But for the automated extraction of the constraints, they did not consider the inter-procedure relation. In our work, we leverage the call hierarchy to retrieve the constraint propagation information. An essential difference is that they had to manually compare the corresponding API document directives with the extracted constraints.

Tan et al. [66] proposed an approach to automatically extract program rules and then use these rules to detect inconsistencies between comments and the code. This work differs to ours in certain aspects: First, the analysis input of this work is inline comments. Second, the target is limited within the area of lock-related topics. Their subsequent work on the comment level detection includes [67], [68]. Similar work on comment quality analysis and use case documents analysis were presented in [69] and [70] respectively.

Treude et al. [71] proposed an approach to augment API documentation with insights sentences from Stack Overflow, so that dispersed information sources could be integrated and provide more comprehensive support for developers. Similarly, an approach to link source code examples from online sites such as Stack Overflow and Github Gists to API documentation was presented in [72]. Compared with all these work, we target at different research questions although some similar analysis techniques are used.

6 Conclusion and Future Work

A vast body of research has presented approaches to detect defects of programs, but largely overlooked the correctness of the associated documents, in particular API documentation. In this paper, we investigated the detection of API document defects and their repair. We presented *DRONE*, which can automatically detect API document defects at the semantic level and recommend repairs. To the best of our knowledge, this is the first work of this kind. Our experiments demonstrated the effectiveness of our approach: in the first experiment for selected JDK API defect detection, the F-measure of our approach achieved 79.9%, 71.7%, and 81.4%, respectively, indicating a practical feasibility. In our second experiment, we extended the applicability on many more API packages including some other JDK APIs and Android libraries, and reused the heuristics from the first experiment. Although being slightly less accurate, the overall performance is still good. Finally, in a third experiment, we showed that the quality of the generated documentation repair recommendations for the detected defects was satisfactory for most of our study participants. We achieved average scores 4.48, 3.82, 4.53, 4.31 (out of 5) in terms of accuracy, content adequacy, and conciseness & expressiveness, respectively.

With our approach, we could expose various API directive defects in JDK and Android API documents, in contrast to what is believed that widely used and well-documented APIs would not exhibit such document defects. This suggests that even more serious defects are existing in other, less mature, projects.

To demonstrate a wider applicability of our approach, additional case studies with various types of APIs and extra coverage of documents are required, which are planned in our future work. We also plan to overcome the limitations identified in the experiments to further boost the accuracy of the approach. In addition, although we provide a prototype in our current paper, a monolithic, full-fledged tool is still under development. Other possible future research includes the integration of an automated constraint description generator for incomplete documents, so we are able to work properly with documentation and code of libraries implemented in different programming languages.

Acknowledgements

We are grateful to the anonymous reviewers for their many constructive comments. This work was partially supported by the National Key R&D Program of China (No. 2018YFB1003900), and the Collaborative Innovation Center of Novel Software Technology in China. T. Chen is partially supported by UK EPSRC grant (EP/P00430X/1), ARC Discovery Project (DP160101652, DP180100691), and NSFC grant (No. 61872340). We also acknowledge the Swiss National Science Foundation's support for the project SURF-MobileAppsData (SNF Project No. 200021-166275).

References

[1] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of api usability," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 2013, pp. 5–14.

[2] B. Iyer and M. Subramaniam, "The strategic value of apis," *Harvard Business Review*, p. 1, 2015.

[3] B. A. Myers, "Human-centered methods for improving api usability," in *Proceedings of the 1st International Workshop on API Usage and Evolution*, ser. WAPI '17. IEEE Press, 2017, pp. 2–2.

[4] J. Stylos, B. A. Myers, and Z. Yang, "Jadeite: Improving api documentation using usage information," in *CHI '09 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '09. New York, NY, USA: ACM, 2009, pp. 4429–4434.

[5] R. H. Earle, M. A. Rosso, and K. E. Alexander, "User preferences of software documentation genres," in *Proceedings of the 33rd Annual International Conference on the Design of Communication*. ACM, 2015, p. 46.

[6] H. Li, Z. Xing, X. Peng, and W. Zhao, "What help do developers seek, when and how?" in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 142–151.

[7] D. G. Novick and K. Ward, "What users say they want in documentation," in *Proceedings of the 24th Annual ACM International Conference on Design of Communication*, ser. SIGDOC '06. New York, NY, USA: ACM, 2006, pp. 84–91.

[8] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How do api changes trigger stack overflow discussions? a study on the android sdk," in *proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 83–94.

[9] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. C. Gall, "Analyzing apis documentation and code to detect directive defects," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 27–37.

[10] S. Panichella, J. Aponte, M. D. Penta, A. Marcus, and G. Canfora, "Mining source code descriptions from developer communications," in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*. IEEE, 2012, pp. 63–72.

[11] G. Petrosyan, M. P. Robillard, and R. De Mori, "Discovering information explaining api types using text classification," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 869–879.

[12] P. Chatterjee, M. A. Nishi, K. Damevski, V. Augustine, L. Pollock, and N. A. Kraft, "What information about code snippets is available in different software-related documents? an exploratory study," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 382–386.

[13] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora, "Codes: Mining source code descriptions from developers discussions," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 106–109.

[14] C. Shold, "The importance of documentation," 2014. [Online]. Available: https://www.carsonshold.com/2014/03/the-importance-of-documentation

[15] G. Uddin and M. P. Robillard, "How api documentation fails," *Software, IEEE*, vol. 32, no. 4, pp. 68–75, 2015.

[16] A. DuVander, "Api consumers want reliability, documentation and community," 2013. [Online]. Available: https://www.programmableweb.com/news/api-consumers-want-reliability-documentation-and-community/2013/01/07

[17] M. A. Saied, H. Sahraoui, and B. Dufour, "An observational study on api usage constraints and their documentation," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 33–42.

[18] J. Bloch, "How to design a good api and why it matters," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006, pp. 506–507.

[19] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik, "How do api documentation and static typing affect api usability?" in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 632–642.

[20] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, "What should developers be aware of? an empirical study on the directives of api documentation," *Empirical Software Engineering*, vol. 17, no. 6, pp. 703–737, 2012.

[21] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary, 2008. Proceedings*, 2008, pp. 337–340.

[22] A. D. Sorbo, S. Panichella, C. A. Visaggio, M. D. Penta, G. Canfora, and H. C. Gall, "Development emails content analyzer: Intention mining in developer discussions," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 12–23.

[23] A. D. Sorbo, S. Panichella, C. Aaron Visaggio, M. Di Penta, G. Canfora, and H. C. Gall, "DECA: development emails content analyzer," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, 2016, pp. 641–644.

[24] T. Brants, "Tnt: a statistical part-of-speech tagger," in *Proceedings of the sixth conference on Applied natural language processing*. Association for Computational Linguistics, 2000, pp. 224–231.

[25] F. M. Hasan, N. UzZaman, and M. Khan, "Comparison of different pos tagging techniques (n-gram, hmm and brill's tagger) for bangla," in *Advances and Innovations in Systems, Computing Sciences and Software Engineering*. Springer, 2007, pp. 121–126.

[26] D. J.Asmussen, "Survey of pos taggers-approaches to making words tell who they are," *DK-CLARIN WP 2.1 Technical Report*, 2015.

[27] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, Feb. 2009, vol. 185, ch. 26, pp. 825–885.

[28] L. Moreno and A. Marcus, "Automatic software summarization: the state of the art," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, 2017, pp. 511–512.

[29] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*, 2013, pp. 23–32.

[30] L. Moreno, A. Marcus, L. L. Pollock, and K. Vijay-Shanker, "Jsummarizer: An automatic generator of natural language summaries for java classes," in *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*, 2013, pp. 230–232.

[31] M. Kamimura and G. C. Murphy, "Towards generating human-oriented summaries of unit test cases," in *2013 21st International Conference on Program Comprehension (ICPC)*, 2013, pp. 215–218.

[32] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: an empirical investigation," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 547–558.

[33] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 43–52.

[34] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 279–290.

[35] R. Feldt and A. Magazinius, "Validity threats in empirical software engineering research-an initial survey." in *SEKE*, 2010, pp. 374–379.

[36] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: a survey," in *Proceedings of the 2002 ACM symposium on Document engineering*. ACM, 2002, pp. 26–33.

[37] R. P. Buse and W. R. Weimer, "Automatic documentation inference for exceptions," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 273–282.

[38] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[39] S. Inzunza, R. Juárez-Ramírez, and S. Jiménez, "Api documentation," in *Trends and Advances in Information Systems and Technologies*, Á. Rocha, H. Adeli, L. P. Reis, and S. Costanzo, Eds. Cham: Springer International Publishing, 2018, pp. 229–239.

[40] L. Shi, H. Zhong, T. Xie, and M. Li, "An empirical study on evolution of api documentation." in *FASE*, vol. 6603. Springer, 2011, pp. 416–431.

[41] B. Dagenais and M. P. Robillard, "Creating and evolving developer documentation: understanding the decisions of open source contributors," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 127–136.

[42] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: what they are and how developers perceive them," *Empirical Software Engineering*, vol. 21, no. 1, pp. 104–158, 2016.

[43] H. Zhong and Z. Su, "Detecting api documentation errors," in *ACM SIGPLAN Notices*, vol. 48, no. 10. ACM, 2013, pp. 803–816.

[44] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 213–224.

[45] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated api property inference techniques," *Software Engineering, IEEE Transactions on*, vol. 39, no. 5, pp. 613–637, 2013.

[46] F. N. A. Al Omran and C. Treude, "Choosing an nlp library for analyzing software documentation: a systematic literature review and a series of experiments," in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 187–197.

[47] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language api documentation," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 307–318.

[48] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language api descriptions," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 815–825.

[49] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 832–841.

[50] B. Dagenais and M. P. Robillard, "Using traceability links to recommend adaptive changes for documentation evolution," *Software Engineering, IEEE Transactions on*, vol. 40, no. 11, pp. 1126–1146, 2014.

[51] M. P. Robillard and Y. B. Chhetri, "Recommending reference api documentation," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1558–1586, 2015.

[52] G. Petrosyan, M. P. Robillard, and R. De Mori, "Discovering information explaining api types using text classification," in *Proceedings of the 37th International Conference on Software Engineering*. IEEE Press, 2015, pp. 869–879.

[53] Y. Zhang and D. Hou, "Extracting problematic api features from forum discussions," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013, pp. 142–151.

[54] A. Di Sorbo, S. Panichella, C. Alexandru, J. Shimagaki, C. Visaggio, G. Canfora, and H. Gall, "What would users change in my app? summarizing app reviews for recommending software changes," in *Foundations of Software Engineering (FSE), 2016 ACM SIGSOFT International Symposium on the*, 2016, pp. 499–510.

[55] A. Di Sorbo, S. Panichella, C. V. Alexandru, C. A. Visaggio, and G. Canfora, "Surf: Summarizer of user reviews feedback," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 55–58.

[56] A. Ciurumelea, A. Schaufelbhl, S. Panichella, and H. Gall, "Analyzing reviews and code of mobile apps for better release planning." in *2017 IEEE 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2017, pp. 91–102.

[57] S. Panichella, A. Di Sorbo, E. Guzman, C. Visaggio, G. Canfora, and H. Gall, "How can i improve my app? classifying user reviews for software maintenance and evolution," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, Sept 2015, pp. 281–290.

[58] S. Panichella, A. Di Sorbo, E. Guzman, C. Visaggio, G. Canfora, G. Gall, H.C., and H. Gall, "Ardoc: App reviews development oriented classifier," in *Foundations of Software Engineering (FSE), 2016 ACM SIGSOFT International Symposium on the*, 2016, pp. 1023–1027.

[59] S. P. Reiss and M. Renieris, "Encoding program executions," in *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada*, 2001, pp. 221–230.

[60] D. Lo and S. Khoo, "Smartic: towards building an accurate, robust and scalable specification miner," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, 2006, pp. 265–275.

[61] D. Lo and S. Maoz, "Scenario-based and value-based specification mining: better together," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, 2010, pp. 387–396.

[62] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, 2011, pp. 267–277.

[63] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy, "Unifying fsm-inference algorithms through declarative specification," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 252–261.

[64] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli, "Mining behavior models from user-intensive web applications," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 277–287.

[65] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun, "Behavioral resource-aware model inference," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 19–30.

[66] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* icomment: Bugs or bad comments? */," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 145–158.

[67] L. Tan, Y. Zhou, and Y. Padioleau, "acomment: mining annotations from comments and code to detect interrupt related concurrency bugs," in *Proceedings of the 33rd international conference on software engineering*. ACM, 2011, pp. 11–20.

[68] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *Software Testing, Verification and Validation, 2012 Fifth International Conference on*. IEEE, 2012, pp. 260–269.

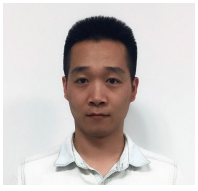
[69] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013, pp. 83–92.

[70] S. Liu, J. Sun, Y. Liu, Y. Zhang, B. Wadhwa, J. S. Dong, and X. Wang, "Automatic early defects detection in use case documents," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 785–790.

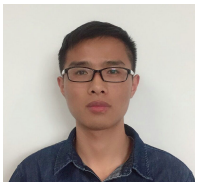
[71] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 392–403.

[72] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 643–652.

Yu Zhou is an associate professor in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics (NUAA). He received his BSc degree in 2004 and PhD degree in 2009, both in Computer Science from Nanjing University China. Before joining NUAA in 2011, he conducted PostDoc research on software engineering at Politechnico di Milano, Italy. From 2015-2016, he visited the SEAL lab at University of Zurich Switzerland, where he is also an adjunct researcher. His research interests mainly include software evolution analysis, mining software repositories, software architecture, and reliability analysis. He has been supported by several national research programs in China.



Changzhi Wang received his BSc degree in Computer Science and Technology, from University of Jinan, Shandong Province, China, in 2014. He is now studying for the MSc degree at Nanjing University of Aeronautics and Astronautics. His research interests include software evolution analysis, and mining software repositories.



Xin Yan received his BSc degree in Software Engineering, from Jiangsu University, China, in 2017. He is currently a MSc student in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics. His research interests include software evolution analysis, artificial intelligence, and mining software repositories.

Taolue Chen received the Bachelors and Masters degrees from the Nanjing University, China, both in computer science. He was a junior researcher (OiO) at the CWI and acquired the PhD degree from the Free University Amsterdam, The Netherlands. He is currently a lecturer at the Department of Computer Science and Information Systems, Birkbeck, University of London. Prior to this post, he was a (senior) lecturer at Middlesex University London, a research assistant at the University of Oxford, and a postdoctoral researcher at the University of Twente, The Netherlands. His research interests include formal verification and synthesis, program analysis, logic in computer science, and software engineering.

Sebastiano Panichella is a senior researcher at Zurich University of Applied Sciences (ZHAW), former at University of Zurich (UZH) when this work was carried out. His research interests are in software engineering (SE) and cloud computing (CC): Continuous Delivery, Continuous integration, Software maintenance and evolution (with particular focus on Cloud Applications), Code Review, Mobile Computing, Summarization Techniques for Code, Changes and Testing. His research interests also include Textual Analysis, Machine Learning and Genetic Algorithms applied to SE problems. His research is funded by one Swiss National Science Foundation Grants. He is author of several papers that appeared in International Conferences (ICSE, ASE, FSE, ICSME, etc.) and Journals (ESE, IST, etc.). These research works involved studies with industrial companies and open source projects and received best paper awards. He serves as program committee member of various international conferences (e.g., ICSE, ASE, ICPC, ICSME, SANER, MSR). He is a member of IEEE. He is Editorial Board Member of the Journal of Software: Evolution and Process (JSEP).

Harald Gall is Dean of the Faculty of Business, Economics, and Informatics at the University of Zurich, Switzerland (UZH), and professor of software engineering in the Department of Informatics at UZH. His research interests are in evidence-based software engineering with focus on quality in software products and processes. This focuses on long-term software evolution, software architectures, software quality analysis, data mining of software repositories, cloud-based software development, and empirical software engineering. He is probably best known for his work on software evolution analysis and mining software archives. Since 1997 he has worked on devising ways in which mining these repositories can help to better understand software development, to devise predictions about quality attributes, and to exploit this knowledge in software analysis tools such as Evolizer or ChangeDistiller.