2017 IEEE/ACM 39th International Conference on Software Engineering

Analyzing APIs Documentation and Code to Detect Directive Defects

Yu Zhou*, Ruihang Gu*, Taolue Chen†, Zhiqiu Huang*, Sebastiano Panichella‡ and Harald Gall‡
*College of Computer Science, Nanjing University of Aeronautics and Astronautics, Nanjing, China
Email: {zhouyu,ruihang_gu,zqhuang}@nuaa.edu.cn
†Department of Computer Science, Middlesex University London, UK
Email: t.chen@mdx.ac.uk
‡Department of Informatics, University of Zurich, Switzerland
Email:{panichella,gall}@ifi.uzh.ch

***Abstract*—Application Programming Interface (API) documents represent one of the most important references for API users. However, it is frequently reported that the documentation is inconsistent with the source code and deviates from the API itself. Such inconsistencies in the documents inevitably confuse the API users hampering considerably their API comprehension and the quality of software built from such APIs. In this paper, we propose an automated approach to detect defects of API documents by leveraging techniques from program comprehension and natural language processing. Particularly, we focus on the directives of the API documents which are related to parameter constraints and exception throwing declarations. A first-order logic based constraint solver is employed to detect such defects based on the obtained analysis results. We evaluate our approach on parts of well documented JDK 1.8 APIs. Experiment results show that, out of around 2000 API usage constraints, our approach can detect 1158 defective document directives, with a precision rate of 81.6%, and a recall rate of 82.0%, which demonstrates its practical feasibility.**

***Keywords*-API documentation; static analysis; natural language processing**

I. Introduction

API (Application Programming Interface) constitutes a common reuse pattern to construct larger software systems nowadays [1]. To correctly use APIs — especially those developed by third-party vendors—clients are heavily relying on the formal documentation to seek assistance and instructions [2], [3]. Normally, the documents need to clarify the assumptions and constraints of these APIs, i.e., the usage context, so the clients can hopefully avoid pitfalls when using them by following these guidelines. However, due to the inherent evolution nature of programs and the changes of APIs [4], as well as somehow accidental overlook of the corresponding documents from developers, defective API documents are frequently encountered in practice. Sometimes, they lurk deeply in the delivered software artifact, and potentially lead to frustration, major loss of time, and even abandonment of the API [5].

As a concrete example, in the latest JDK1.8 API, the 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 who is unfamiliar with JDK, but who uses this API in the code after reading the document, it is possible that (s)he passes the method an instance of the javax.swing.JFrame type, since this argument is compatible to the Component type in Java and thus is not forbidden based on the documentation. Such kind of usage will also pass the static check easily. However, probably surprisingly, 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 one of the arguments (i.e., Component in this case) is of the Window type. The document of addImpl does clarify that, if a Window object is added to a container, the exception IllegalArgumentException will be thrown. But this important constraint is not addressed at all in the related documentation for insertTab or addTab.

As another example in JDK1.8, the 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 very far from complete. Indeed, the corresponding code actually requires that the firstEndpoint is no less than 0, and the secondEndpoint is no more than the value of the character counts; an IllegalArgumentException would be thrown otherwise.

As a third example in JDK1.8, the document for javax.swing.JTable.getDefaultEditor(class columnClass) only writes "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 of null type. If it is, the method directly returns null value without throwing an exception. But this information is not even mentioned in the document, whereas the elusive document seems to be discussing what will happen if c is *not* null.

As a fourth example in JDK1.8, the 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 found that the exceptional condition is

1558-1225/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE.2017.11

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 document is incorrect in specifying the range of the argument button.

The above four examples are simply practical and evident examples of the so called "API document defects". Indeed similar problems are frequently found in the documents. On the stackoverflow website, 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] As JDK's documentation is generally considered to be of high quality, it might not be difficult to be convinced that documents of other projects suffer from similar (or even worse) issues. Some initial research has been done, for instance, Saied et al. enumerated categories of common API usage constraints and their documentation [6]. Undoubtedly, high-quality documentation is indispensable for the usability of APIs [7], [8], and we believe that a complete and correct documentation would be highly favourable by API users.

Given the bulk of API documents and code, it is infeasible to check/discover such problems manually in practice. Sometimes even if it is manageable on a small scale, manual examination would be tedious, inefficient, and error-prone by itself. In this paper, we propose an *automated* approach to detect the defects of API documents. The "defect" in our context encompasses two scenarios. The first scenario is that the constraint description of API usage is incomplete in the documentation (the aforementioned first three examples); the second scenario is that the description exists but semantically incorrect with respect to the code (the fourth example described above). We do not consider syntactic errors in the documents as defects, since most of such errors could be detected by textual editors with grammar checkers and may not be relevant for developers. Instead, we focus on the *semantic* aspects. By identifying and correcting these defects, the quality of API documents could be increased, which would further enhance their usability.

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 are more reliable compared to the documentation. (The assumption can be relaxed; cf. Section IV.) On the other hand, the API documents are usually a combined description of various pieces of information, such as general descriptions, function signature related descriptions, exception throwing declarations, code examples, etc. Among these, we hypothesize that statements on function signatures (i.e., related to parameter types and values) and exceptions provide the most crucial information for users during programming. In [9], such statements are defined as *directives*, which are the main focus of our work. Particularly, we limit our attention to 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%) [9]. Indeed, all of the aforementioned illustrative examples are directives within this category. We believe that automatic detection of such defects in API documents will be of great value for developers/users to better understand APIs and to avoid inappropriate use of an unfamiliar API. In Java programs, this kind of directive is generally annotated with *@param, @exception, @throws, etc.* tags. Such structured information makes it much easier to extract the document directives automatically in practice.

The main contributions of the paper are:

1) We propose an approach which can automatically detect the defects of API document directives. The approach contains 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 are then fed into an SMT solver, i.e., Z3 [10], to detect the defects in case of inconsistency.
2) The approach covers four types of document defects at the semantic level, and are evaluated on a part of the latest JDK 1.8 APIs with their documentation. The experimental results show that our approach is able to detect 1419 defects hidden in the investigated documentation. Moreover, the precision and the recall of our detection are around 81.6% and 82.0% respectively which indicate feasibility of the application.
3) We summarize more than 60 heuristics on the typical descriptions of API usage constraints in documents, which could be reused across different documentation projects. We also implement a prototype based on the proposed approach, which can facilitate the detection of API defects, especially for JDK1.8 (with a potentially wider applicability in other APIs).

The rest of the paper is organized as follows. Section II illustrates the details of our approach. The experiments with performance evaluation are given in Section III followed by a discussion in Section IV. Section V discusses the related work. Section VI concludes the paper and outlines the future research.

II. Approach

We mainly consider four cases of parameter usage constraints, based on [6]. They are *nullness not allowed*, *nullness allowed*, *range limitation*, and *type restriction*. We now give a brief explanation of these constraints.

- "Nullness not allowed" represents the case where the null value cannot be passed as an argument to a method, otherwise an exception (e.g., NullPointerException) will be thrown.
- "Nullness allowed" represents the opposite case of "Nullness not allowed", where the null value can be passed as an argument and no exception will be thrown. Usually, there is a default interpretation of the null value.
- "Type restriction" represents the case that there are some specific type requirements on the argument. Not only are

[1] cf. http://stackoverflow.com/questions/2967303/inconsistency-in-java-util-concurrent-future, posted by Mark Peters on June 03, 2010

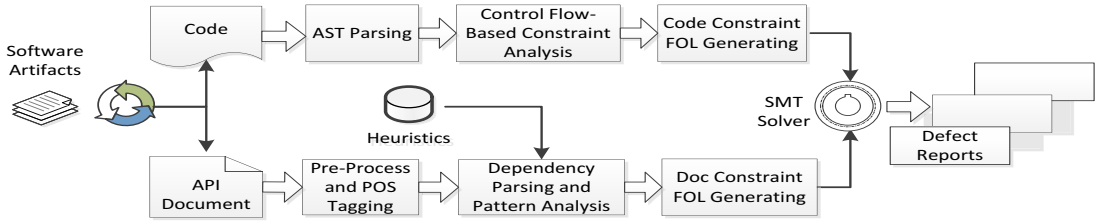

Fig. 1. Approach overview

the argument types compatible with the declared parameters, but also some additional, implicit rules should be respected. This is usually due to the features of object-oriented languages, especially the inheritance.

- "Range limitation" represents the case where some specific value ranges of the arguments are expected. Otherwise the values of the arguments are out of the scope, and usually some exceptions will be thrown.

For a better understanding of API, such usage constraints are supposed to be specified explicitly in the accompanied documents, as otherwise it would be misleading to the API users. This is, unfortunately, not the case in practice, which gives rise to numerous defects in API documentation. Our aim is to detect these defects automatically.

Our approach proceeds as the following steps. Figure 1 gives an overview.

- We first extract the annotated document out of the source code. This is a relatively simple procedure. We then have two branches (cf. Figure 1).
- In the upper branch, we exploit static code analysis techniques, i.e., to parse the code to obtain the *abstract syntax trees* (AST) and analyze the statements of control flow decisions and exception handling, as well as the call invocation relation between methods. The results are given in the form of FOL expressions. The details of this part will be elaborated in Section II-A.
- In the lower branch, we tag the POS features of the directives of the API documents, and extract the restriction and constraints related parts which are also rendered into FOL expressions. The details of this part will be elaborated in Section II-B.
- An SMT solver is employed to solve the logical equation relation between the pair of FOL formula derived from the above two procedures, and possible inconsistencies are reported if any.

For technical reasons and scalability of the approach, we make the following assumptions in the current work.

- For the code analysis: (1) we bound the depth of call graph, which is specified as a parameter of the procedure; (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 handling directives, we concentrate on the directives of the form "@tag target description". In particular, the @tag type includes "param", "exception" and "throws". "target" denotes the tagged entity and "description" is the constraint related expressions regarding the target. We hypothesize that in these expressions, API developers incline to use recurrent linguistic patterns to describe the constraints.

In the following subsections, we will articulate the two processing branches respectively.

A. Extract constraints from API source code

In this subsection, we illustrate the workflow of the upper branch in Figure 1. The input of the procedure is the API source code, and the output is an FOL formulae. The procedure goes through the following steps:

- *Step 1. Construct AST.* By parsing the API source code, for each method m we extract an AST $tree_m$. This step is usually a routine of program analysis. In addition, we generate the *static call graph* G by Eclipse's *CallHierarchy* (org.eclipse.jdt.internal.corext.callhierarchy, used in the plugin environment). From the call graph, we can easily 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 the exception information.* For each public method m, by traversing the AST $tree_m$, we locate each throw statement and collect the exception information. This is carried out for all methods in the API. The exception information of each method m is stored 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 the method,
  - t is the type of the exception,
  - 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 method invocation, is obtained.

  Algorithm 1 gives the pseudo-code of $expExtractor$. The AST parsing part of the algorithm is implemented by the aid of the Eclipse JDT toolkit, which also explains the methods *isThrowable*, *isComposite* and *isMethod* in the pseudo-code. The inputs $expExtractor$ are the statement sequence of the source code as an AST representation and the depth of the call hierarchy. First the algorithm iterates the statements in m. If

```
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}
7              /* Recursively invoke itself, in case
                  of composite statement               */
8              else if isComposite(stm) then
9                  List subList ⟵ (Block)stm.getBody();
10                 infoList ⟵
                   infoList ∪ expExtractor(subList, dep);
11             /* If the statement contains a method
                  call of n, check the invoked method
                  recursively                          */
12             else if isMethod(stm) ∧ (stm's args ∈ m's list) then
                   /* n is the callee of m in stm     */
13                 mList ⟵ n.getBody();
14                 infoList ⟵
                   infoList ∪ expExtractor(mList, dep − 1);
15
16         end
17     end
18 end
```

Algorithm 1: $expExtractor$ algorithm

the statement contains an exception-throw, the exception type, trigger condition, the relevant parameter, as well as the method name, will be recorded and inserted into the list (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.) If the statement itself is a composite block, we recursively go through the internal statements of the block, and extract the corresponding exception information (line 7-10). If the statement invokes another method call, and m's argument(s) is(are) passed onto the callee method n, we will use the recursion with parameters — the statement body of the callee method as the parameter, together with the depth value decreased by 1 (line 12-15). The reason why we require the parameter match in the invocation case is to track and guarantee the constraints are on the same parameter list from the caller method. This recursion continues until the depth 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 the exception information.* In *Step 2*, we have collected a list of ExcepInfo_m for each method m by directly analyzing the ASTs. We now 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 the exception information.* The cleaned exception information from the previous step is further classified into the following *categories* to formulate parameter usage constraints.
  (1) Category "Nullness not allowed". They consist of exceptions (m, P, t, c) such that c implies $p = null$ for some $p \in P$;
  (2) Category "Type restriction". They consist of exceptions (m, P, t, c) such that c contains $instanceOf$.
  (3) Category "Range limitation". They consist of exceptions (m, P, t, c) where some comparison operators exist in condition c, except that it is compared with *null*, in which case, (m, P, t, c) will 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 the technique proposed in [6] and described in Step 5 below.
- *Step 5. Constraints generation.* We formulate the collected information regarding the parameter usage constraints as an FOL formula Φ_{API}. According to the four types of the parameter usage constraints, we introduce the following *predicates*: (1) $\mathsf{NullAllow}(m, a)$, where m is the method and a is an argument of m; (2) $\mathsf{NullNotAllow}(m, a)$; (3) $\mathsf{Type}(m, a, cl)$, where m is a method, a is an argument of m and cl is a type provided by the Java language.
  Accordingly, for each method m, we generate a formula Φ_m which is a (logic) conjunction of
  - $\mathsf{NullNotAllow}(m, p)$, if p is a parameter of m and (m, P, t, c) is in the "nullness not allowed" category from Step 4.
  - $\mathsf{NullAllow}(m, p)$, if p is a parameter of m and "nullness allowed" category. For such constraints, there are no exceptions thrown. In this case, we use the control flow analysis technique similar to the one proposed in [6]. We mainly examine the parameter related conditional branches. If the branch handles the null value of the parameter without throwing the exception or the branch simply ignores

the case of null value of the parameter, we regard the parameter as "nullness allowed". In this part, we do not consider aliasing problems either.

- $\neg\mathsf{Type}(m, p, cl)$, if p is a parameter of m and there is some (m, p, t, c) such that c *implies* $p = \mathsf{instanceOf}(cl)$.
- $\bigwedge_{m,p\in P} \bigwedge_{(m,p,t,c)\in \mathrm{ExcepInfo}_m} \neg c$ which specifies the range of each parameter available from the exception information.

B. Extract constraints from directives

In this section we describe the approach we applied for the automatic extraction of constraints from directives (i.e., workflow of the lower branch in Figure 1) in API documents. The main idea underlying this approach is the observation that *constraints reported in textual descriptions of API documents have specific/recurrent grammatical structures* (depending of the constraint category) *that share some common characteristics*. Thus, such commonalities can be captured by the notion of heuristics through domain knowledge [11], [12] for enabling the automatic extraction of constraints.

The approach we proposed for this step relies on specific NLP techniques for enabling the automatic extraction of constrains of a given type in API documents. In particular, similarly to our previous work on pattern based natural language processing [11] (with necessary adaptions to the new context), the definition of the proposed approach consists of two steps:

1) *(manual) analysis of the existing linguistic patterns* of constrains described in API documents having similar (recurrent) grammatical structures;
2) *for each linguistic pattern we defined an NLP heuristic responsible for the recognition of the specific pattern.*

We performed a manual examination of 429 documents of *java.awt* and *javax.swing* 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 (the discovered patterns are available in our replication package[2]). 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.

For each extracted linguistic pattern we defined an NLP heuristic responsible for the recognition of the specific pattern. The formalization of a heuristic requires three steps: (1) discovering the relevant details that make the particular syntactic structure of the sentence recognizable; (2) generalizing some kinds of information; and (3) ignoring useless information. In the end of this process, a group of related *heuristics* constitutes the pattern for a specific constraint category.

[2]http://www.ifi.uzh.ch/en/seal/people/panichella/tools/SURF0.html

Since the API documentation is usually different from pure natural language narrations—for instance it is frequently mixed with code-level identifiers—, differently from our previous work, *we needed to pre-process such statements*. In the following, we use some examples to explain the procedure with emphasis on the pre-processing step.

In our approach, the documents are subject to the POS tagging and the dependency parsing. We use Stanford lex parser[3] to mark all terms of the words and their dependency relation in the constraint related directives extracted from the documents. Particularly, we focus on the sentences annotated with *@param*, *@exception* and *@throws* tags.

Before dependence parsing, as mentioned before, we need to pre-process the texts. 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, but the words enclosed with such markers are recorded too, since these are either the keyword, or the corresponding variable/method/parameter names in the code.

After tagging, we perform dependency parsing and pattern analysis, aided by heuristics. For this, we largely follow the methodology of [11]. As a concrete example for heuristic based parsing, the document of java.awt.Choice.addItem(String item) states "@exception NullPointerException if the item's value is equal to <code>null</code>". We first record the exception type. Then we remove the pair of "<code>" and "</code>". Thus the sentence "if the item's value is equal to null" is finally fed into the parser.

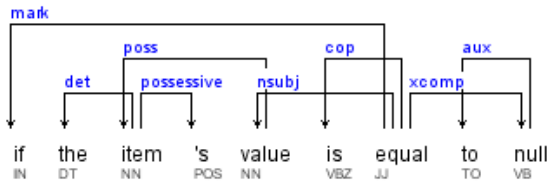

Fig. 2. POS tagging and dependency example[4]

Figure 2 illustrates the dependency parsing result of our example document description. In this sentence, we just omit 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 a heuristic during matching. In this way, the subject(subj) and object("null") of "equal to" will be extracted and be

[3]cf. http://nlp.stanford.edu/software/lex-parser.shtml

[4]The meaning of POS tags and phrasal categories can be found via http://www.cis.upenn.edu/∼treebank/

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 $item = null$.

Undoubtedly, there are other, more complicated cases than this simple example, making the extraction of heuristics a non-trivial task. They require an in-depth pre-processing. A typical situation is that there are code-level identifiers and mathematical expressions in the documents. For example, the document of java.awt.color.Color-Space.getMinValue(int component) states "@throws IllegalArgumentException if component is less than 0 or greater than numComponents -1". We first recognize the special variable names and mathematical expressions through regular expression matching. The naming convention of Java variable follows the *camelcase* style. If a upper case letter is detected in the middle of a word, the word is regarded as an identifier in the method. Similarly, if the word is followed by some mathematical operators, they are regarded as expressions. 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. We have defined 29 regular expressions and rules to detect these cases[5]. 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 recognizing the specific identifiers and expressions in the description, we create a fresh labeled word to replace them to facilitate the dependency parsing.

In the end of this process (which required approximatively 2 weeks of work) we formalized 64 heuristics (available in our replication package). A brief statistics of heuristics for each constraint type is given in Table I. Since these heuristics are different from each other, during the linguistic analysis phase, one directive will be accepted by at most one heuristic (possibly none, in the case of no constraints specified). We remark that these heuristics are interesting in their own right, and can potentially be reused and extended in other related researches.

TABLE I
HEURISTICS SUMMARIZATION

Constraints types	Heuristic number
Nullness not allowed	20
Nullness allowed	11
Type restriction	10
Range limitation	23
In total	64

We are now in a position to generate the parameter usage related constraints for the documentation, again represented by an 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 $(component < 0) \vee (component > \text{numComponent} - 1)$.

C. Identify defects

Recall that from the preceding two sections, 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 II-A. Intuitively, they represent the collected information regarding the API source code and the directives of the documents, with respect to the four types parameter usage constraints. The main task now is to detect the mismatch of these two; our approach is to check whether the two formulae Φ_{API} and Ψ_{DOC} are equivalent. If this is the case, one can be somehow confident to say that all constraints (wrt the four types of method parameter usage constraints addressed in the paper) in the API are captured in the document 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 which can point out the method and the parameter thereof, as well as the involved exception. Then we can check whether such a mistatch is a real defect of the API document.

Formally, we then make a query to check whether

$$\Phi_{\text{API}} \Leftrightarrow \Psi_{\text{DOC}} \tag{1}$$

holds. If (1) holds, we can conclude that the API source code and the related documents are matched. Otherwise, usually a counterexample can be returned, indicating where the mismatching happens. As a simple example, for method $f(x)$ with a single argument x, from the API source code, one finds that x must be strictly positive, i.e., $\Phi_{\text{API}} = x > 0$. However, in the document, it is only stated that x must be nonnegative, i.e., $\Psi_{\text{DOC}} = x \geq 0$. In this case, (1) is instantiated by $x > 0 \Leftrightarrow x \geq 0$, which clearly fails. By tracing the relevant predicate (in this case $x \geq 0$), this particular defect of the document can be revealed. Note that, when one counterexample is returned, in principle we can only detect one inconsistency. To detect all inconsistencies, one has to update the formulae Φ_{API} and Ψ_{DOC} by removing the relevant part of the detected counterexample (defect), 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

[5]http://www.ifi.uzh.ch/en/seal/people/panichella/tools/SURF0.html

often suggests multiple sources of inconsistences. Hence, only a small amount of rounds is needed.

Satisfiability modulo theories (SMT) generalizes boolean satisfiability by adding useful first-order theories such as equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, etc [13]. To perform the check in (1), we exploit an SMT solver. Clearly, (1) is equivalent to checking whether

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

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, 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.

III. Experiments

To better support the detection process, we implement a prototype which takes the API code and the document directives as inputs. The outputs of the prototype are the generated FOL expressions in the SMT-LIB 2.0 standard[6]. These expressions are then fed into the SMT solver Z3.

A. Settings

We conduct two experiments. In the first one, we limit the evaluation within the scope of the packages *java.awt* and *javax.swing*; in the second one, we reuse the heuristics defined in the first one and evaluate the performance for another six packages. In both experiments, the evaluations are conducted on a laptop with an Intel i7-4710MQ 2.5GHz processor and 12.0GB RAM, running Windows 7 64-bit operating system. The versions of Java and Eclipse are 1.8 and Luna-SR2 respectively. The depth of call hierarchy is set to be 4.

The metrics used in the experiments are *precision, recall* and *F-measure*. Precision is used to measure the exactness of the prediction set; recall measures the completeness. Precision and recall are calculated as follows[7].

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

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

[6]cf. http://smtlib.cs.uiowa.edu/

[7]TP: true positive; FP: false positive; FN: false negative.

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

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

Replication Package. We make publicly available in a replication package[8] with (i) the material and working data sets of our study (e.g., including input data, and output defective reports); (ii) NLP patterns and the defined NLP heuristics.

B. Results

a) Experiment 1: We first evaluate the performance of our approach applied to the target packages and their documents (i.e., *java.awt* and *javax.swing*). The packages parsed by our prototype contain around 0.5 million lines of code (LoC)[9] and 16379 Javadoc tags in total. The details are summarized in Table II. Over these dataset, the program analysis process takes around two hours, while the document analysis takes around one hour. Finally, our approach outputs 1975 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 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 (16379 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 a subset of APIs with the constraints detected by our tool as the sample. We also apply 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, if any. Each report is examined by three subjects independently. A majority vote mechanism is used to resolve possible conflicts. The manual classification process takes around four and one-half days.

TABLE II
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
Total	551.6	13914	1409	956

The results of Experiment 1 are summarized in Table III. Overall, out of these reported 1419 defects (TP+FP), 1158 turn out to be real defects, giving rise to a precision of 81.6%. Combined with 255 false negatives, we get a recall of 82.0%. The average F-measure is 81.8%. In particular, all of the four defective document examples in *Introduction* are detected successfully. Our approach performs well on the

[8]http://www.ifi.uzh.ch/en/seal/people/panichella/tools/SURF0.html

[9]The statistics includes comments and space.

selected API packages where the heuristics are summarized. Moreover, Table III 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 III
RESULTS OF EXPERIMENT 1

Category	TP	FP	FN	Precision	Recall	F-measure
Nullness Not Allowed	116	34	33	0.773	0.779	0.776
Nullness Allowed	400	34	38	0.922	0.913	0.917
Range Limitation	485	176	97	0.734	0.833	0.780
Type Restriction	157	17	87	0.902	0.643	0.751
Total	1158	261	255	0.816	0.820	0.818

Among these four constraint categories, we found the precision for the *range limitation* type and the *nullness not allowed* type is lower than 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 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 II-C). However, unfortunately, the document directive for 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 costs a false negative. In general, we strive to achieve a trade-off between false positive and false negative, but preciser program analysis would be needed which is subject to further investigation.

b) Experiment 2: In this experiment, we extend the exploration scale to cover more API libraries. Particularly, we incorporate six other JDK packages, i.e.,*javax.xml, javax.management, java.util, java.security, java.lang* and *java(x).sql*[10] into our study, but still reuse the heuristic of the first experiment. The information of these packages is given in Table IV. In this part, our approach generates 2057 FOL constraints for the APIs methods, and 1188 (TP+FP) are reported as defected.

TABLE IV
DATA OVERVIEW IN EXPERIMENT 2

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
Total	520.9	12778	5401	3347

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 V summarizes the performance details for each constraint category.

TABLE V
RESULTS OF EXPERIMENT 2

Category	TP	FP	FN	Precision	Recall	F-measure
Nullness Not Allowed	294	154	32	0.656	0.902	0.760
Nullness Allowed	45	26	14	0.634	0.763	0.692
Range Limitation	289	334	62	0.464	0.823	0.593
Type Restriction	31	15	5	0.674	0.861	0.756
Total	659	529	113	0.555	0.854	0.672

Out of these 1188 detected defects, 659 turn out to be true positives, and 529 false positives, giving a precision rate of

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

55.5%. Taking the 113 false negatives, we get a recall of 85.4%. Similar to the observations of the Experiment 1, the precision of *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 55.5% and a recall of 85.4%, and thus these heuristics can be reused.

C. Threats to Validity

c) Internal Validity: Internal validity focuses on how sure we can be that the treatment actually caused the outcome [14]. In our approach, we directly work on the API code, as well as the accompanying documents. The exception related constraints are therefore solely extracted from the code (via static analysis techniques) and the descriptions (via NLP techniques). Another concern of this aspect is the potential bias introduced in the data set. To minimize this threat, we randomly select the packages from the latest JDK, 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 judgement of some computer science master students, because there is a 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 judgement of three inspectors. Moreover, to validate the items each report is examined by three subjects independently. Then, after an initial validation items, all disagreements were discussed and resolved using a majority vote mechanism.

d) External Validity: External validity is concerned on whether the results can be generalized to the datasets other than those studied in the experiments [14]. To maximize the validity of this aspect, we include additional dataset from six other packages of JDK API documentation. However, as an inherent problem 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. Our goal was to observe whether our approach is capable to find defects in well documented APIs. Indeed, all cases considered in our experiments are from the latest version of JDK. Although they are generally regarded as well-documented APIs, many defects are still detected. Finally, for better reducing the threats mentioned above we plan for future work to extend our study by analyzing APIs of libraries of different domains.

IV. Discussion

For program analysis, we only 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 execution, for example, divide-by-zero. In most cases, such implicit exceptions are caused by programming errors, so it might be inappropriate to include them in the documentation [15]. Therefore, we adopt a similar strategy as in [16] and omit 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 [16], utilizes the *Soot* toolset [17], 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. 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. 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 some other cases, where 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 from the code is "(owner!=null)&&!(owner instanceof Frame)&&!(owner instanceof Dialog)". For such composite one, it relates with the *nullness* as well as the *type*, and we classify the composite ones to the both categories.

One of the goals of our approach is to demonstrate wide existence of API document defects, even in those generally believed well-documented APIs. We have come up with heuristics for JDK libraries, which prove to be effective. However, there is no formal guarantee that the same heuristics will work equally well to other libraries. Nevertheless, we note that the approach presented here is essentially open to incorporate other heuristics to facilitate the NLP. We argue that this by no means devaluates our heuristics for JDK for at least two reasons: (1) JDK has many users and (2) our work, as the first work of this kind, may shed light on developing heuristics for other libraries.

Last, 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 less reliable. Our approach can be adapted to report the *inconsistency* between the code and the

documentation.

V. Related Work

Directives of API documentation and the evolution of API documentation are studied in [9] and [18], [19] respectively. The authors identified the importance of directives of API documentation and gave a taxonomy of 23 kinds [9]. We concentrate on a subset of them, i.e, those related with parameter constraints. [20] investigated the developers' perception of *Linguistic Anti-patterns*. The results indicate that developers perceive as more serious ones the instances where the inconsistency involves both method signature and comments (documentation), and thus should be removed. In a latest survey of API documentation quality conducted by Uddin and Robillard [5], three kinds of problems are regarded as severest, i.e., ambiguity, incompleteness and incorrectness, two of which are considered in our approach. However, all these work, adopted an empirical methodology to investigate the problem and no automated techniques were applied.

Zhong and Su [21] proposed an approach combining 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 certain words), and broken code names (which are referred 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. Buse and Weimer [16] proposed an automated API documentation generation technique for exceptions. However, the authors did not consider the extant documents. Instead, they generated new documents based on the program analysis results. Therefore, the work could not help to identify the document defects.

Saied et al. [6] conducted an observational study on the 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. Tan et al. [22] proposed an approach to automatically extract program rules and use these rules to detect inconsistencies between comments and the code. This work differs to ours on certain aspects: Firstly, the analysis input of this work is inline comments. Secondly, the target is limited within the area of lock-related topics. Their subsequent work on the comment level detection includes [23], [24]. Similar work on comment quality analysis and use case documents analysis were presented in [25] and [26] respectively. Compared with all these work, we target at different research questions although some similar analysis techniques are used.

There is another thread of relevant research on applying the NLP techniques to documents or even discussions in natural languages to infer interested properties [27], such as resource specifications [28], method specifications [29], code-document traceability [30], document evolution/reference recommendation [31], [32], API type information [33], source code descriptions [34], [35], problematic API features [36], change requests based on user reviews [37], [38], [39], [40], [41]. They demonstrated the feasibility of applying NLP techniques to documentation, but did not deal with the defect detections.

VI. Conclusion and Future Work

Computer software, by definition, consists of both programs and documents. A majority of work has been conducted to detect the defects of programs, whereas the correctness of documents is also crucial for the success of a software project. In this paper, we shift the focus and investigate the problems of document defect detection. To the best of our knowledge, this is the first work that automatically detects API document defects at the semantic level. In our first experiment on the latest JDK 1.8 API library, out of 1975 API usage constraints, our approach detects 1419 defects with a precision of 81.6% and recall of 82.0%, indicating a practical feasibility. In our second experiment, we consider the applicability on more API packages and reuse the heuristic from the first experiment. Although suffered a decrease of accuracy, the overall performance is still kept at an acceptable level, with an average precision of 55.5%, and recall of 85.4% respectively.

We have exposed various directive defects in JDK's API documentation, which is widely believed to be well documented APIs. This implies that probably even more serious defects do exist in other less robust projects' documents. To demonstrate a higher applicability of our approach, additional case studies from 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 our approach.

Acknowledgements

We would like to thank the reviewers for the valuable comments. This work was partially supported by the Natural Science Foundation of Jiangsu Province under grant No. BK20151476, the National Basic Research Program of China (973 Program) under grant No. 2014CB744903, the National High-Tech Research and Development Program of China (863 Program) under grant No. 2015AA015303, the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Fundamental Research Funds for the Central Universities under grant No. NS2016093. T. Chen is partially supported by UK EPSRC grant (EP/P00430X/1), European CHIST-ERA project SUCCESS, NSFC grant (No. 61662035), and an overseas grant from the State Key Laboratory of Novel Software Technology, Nanjing Unviersity (KFKT2014A14). We also acknowledge the Swiss National Science Foundation's support for the project Essentials (SNF Project No. 200020-153129).

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] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78800-3_24

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597799

[36] 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.

[37] 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, p. to appear.

[38] A. D. Sorbo, S. Panichella, C. Alexandru, C. Visaggio, G. Canfora, and H. Gall, "Surf: Summarizer of user reviews feedback," in *39th IEEE International Conference on Software Engineering (ICSE 2017), Buenos Aires, Argentina*, 2017, p. To Appear.

[39] 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, p. To Appear.

[40] 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.

[41] 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, p. to appear.