DRONE: A Tool to Detect and Repair Directive Defects in Java APIs Documentation

Yu Zhou*, Xin Yan*, Taolue Chen†, Sebastiano Panichella‡ and Harald Gall§
*College of Computer Science, Nanjing University of Aeronautics and Astronautics, Nanjing, China
Email: {zhouyu,xin_yan}@nuaa.edu.cn
†Department of Computer Science and Information Systems, Birkbeck, University of London, UK
Email: t.chen@dcs.bbk.ac.uk
‡Zurich University of Applied Science, Switzerland
Email: panc@zhaw.ch
§Department of Informatics, University of Zurich, Switzerland
Email:gall@ifi.uzh.ch

***Abstract*—Application programming interfaces (APIs) documentation is the official reference of the APIs. Defects in API documentation pose serious hurdles to their comprehension and usage. In this paper, we present *DRONE*, a tool that can automatically detect the *directive defects* in APIs documents and recommend repair solutions to fix them. Particularly, *DRONE* focuses on four defect types related to parameter usage constraints. To achieve this, *DRONE* leverages techniques from static program analysis, natural language processing and logic reasoning. The implementation is based on the Eclipse-plugin architecture, which provides an integrated user interface. Extensive experiments demonstrate the efficacy of the tool.**

Demo webpage: https://goo.gl/BmEKic
Demo video: https://youtu.be/NDPXiapxoMk

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

I. Introduction

Application programming interfaces (APIs) provide crucial support for software reuse. Developers heavily rely on their documentation to understand and make use of them. Undoubtedly, the accompanying documents are supposed to give correct and complete descriptions to inform client developers. However, in practice it is frequently reported that the APIs documentation fails to meet the developers' expectation [1]. Manually checking the correctness of documentation would be practically infeasible due to the huge amount of APIs code base and their complicated interdependency structure. Automatic document defect detection and repair recommendation are thus highly desirable—not only for client developers, but also for API providers.

Among diverse contents of APIs documentation, the usage constraints and related guidelines are of special importance. Such statements are termed as *directives* [2]. Our work focuses on method parameter usage constraints and relevant exception specifications. They fall into the category of *method call directive* which represents the largest portion of API documentation directives (43.7%) [3]. In Java APIs, this kind of directives is generally annotated by tags, such as @param, @exception, @throws, etc. The structured information is crucial to extract the document directives automatically in practice.

In this demo paper, we present DRONE[1] (**D**etect and **R**epair of d**O**cumentatio**N** d**E**fects), a tool that can automatically detect the defects of directives in Java APIs documentation and recommend repair solutions. The methodology underpinning DRONE is partially based on our previous work [4] for the defect detection. Concretely, we mainly consider the defects of four parameter usage constraint types summarized in the previous work [2], i.e., *nullness not allowed, nullness allowed, range limitation*, and *type restriction*. More recently, DRONE is also enhanced with the ability to provide corresponding repair recommendations [5].

To this end, DRONE leverages techniques from static program analysis, natural language processing (NLP), and logic reasoning. In Java source files, the tagged directives are usually mixed with code. To facilitate the processing, DRONE first extracts the target directives out of the code by pattern matching. Then, DRONE parses the source code and generates an abstract syntax tree (AST) representation. In this step, DRONE also considers parameter passing via method invocation. As a result, the call hierarchy analysis is introduced. In parallel, NLP techniques are applied to the extracted directives. In particular, domain-specific heuristics [6] are defined to help extract the parameter related constraints expressed in the directives. Afterwards, the distilled constraints from both the directives and the code are encoded into first-order formulas, and an SMT solver is adopted to find out the potential inconsistency between them. (In this step, we assume that the code is correct, since it has been extensively tested before delivered as libraries. In contrast, documentation is seldom under strict scrutiny, so any inconsistency is more likely due to a documentation defect.) Once a defect is detected, a repair solution is generated based on templates and is recommended to the API developer. The workflow of DRONE is given in Fig. 1.

DRONE can be used by various users and can be employed in large-scale projects to detect potential defects. For instance, DRONE was successfully used to check some packages of the

[1]DRONE is open sourced at: https://github.com/DRONE-Proj/DRONE

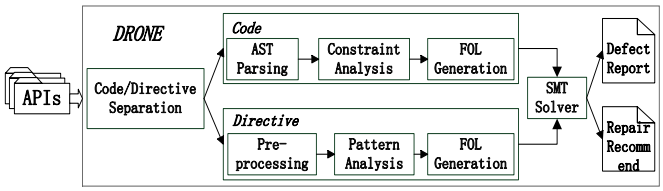

Fig. 1. Workflow of DRONE

JDK API documentation (cf. Section III). For future work, we plan to send our findings to the JDK development team, which would potentially have a positive impact on the Java ecosystem. It is worthwhile emphasizing that, although DRONE is currently focusing on Java, we expect its methodology can be applied in a wide range of APIs written in other programming languages. For this purpose, we have shared materials which are possibly useful in similar projects (e.g., heuristics for NLP, repair recommendation templates) and which are more for "academic" users in Section II. Moreover, for open-source projects, developers could apply DRONE to improve the quality of their documentation. For example, API providers could directly leverage DRONE to detect the potential defects of directives, and repair them once they were found. Software maintainers could also use DRONE to check the co-evolution between directives and code, since in many cases developers tend to overlook updating documentation after the evolution of code.

II. The DRONE Tool

DRONE is implemented as an Eclipse plug-in, thus exploiting many features provided by Eclipse IDE. The current implementation mainly supports Windows platform. To perform the code analysis, DRONE employs JDT[2] (mainly AST and CallHierarchy relevant classes) library in Eclipse. For the directive constraints extraction, it relies on the Stanford Parser[3] to conduct part-of-speech (POS) tagging and dependency parsing. The SMT solver Z3[4] is integrated to check the consistency between the formulated constraints from the two artifacts. When an inconsistency occurs, a template based directive repair recommendation is generated. The overall architecture of DRONE is illustrated in Fig. 2.

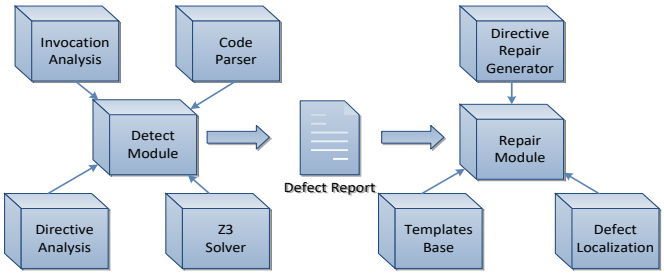

Fig. 2. Architecture of DRONE

DRONE's functionalities could be divided into two parts, i.e., *detection* and *repair*, which are elaborated as follows.

[2]https://www.eclipse.org/jdt/
[3]https://nlp.stanford.edu/software/lex-parser.shtml
[4]https://github.com/Z3Prover/z3

A. Defects Detection

For the detection, DRONE extracts the constraints from both the code and the directives which are transformed to the first-order formulae written in the SMT lib 2.0 format. DRONE integrates the Z3 solver to reason about the equality of two constraints, based on which DRONE outputs the directive defect reports. The details of our approach can be found in the previous work [4], [5].

In this demo paper, we shall focus on how the tool is used. For academic users who might reuse/extend/adapt our approaches to their project, we highlight some derivatives of our research, in particular, the NLP part. We believe that they are of independent interests. Compared with the code, directives are much less structured, since it is written in natural languages. However, we observed that some linguistic patterns recurrently appeared in these directives. For instance, in the "nullness not allowed" category, many directives state that the [parameter] could not be null after "@param" tag, or "if [parameter] is null" after "@throws" tag. We therefore considered a technique based on linguistic patterns which are formulated via heuristics approaches. To define the heuristics, we manually examined more than 400 documents of JDK packages (mainly in *java.awt, javax.swing*, and *javaFX* packages), and identified 67 heuristics. Table I presents some illustrative heuristics and the number of heuristics of each category. DRONE also pre-processes the raw directives since they are usually mixed with code elements or other tags. For that purpose, we defined 29 regular expressions and rules to recognize and remove these unwanted elements.

TABLE I
Example Heuristics

Constraints category	Heuristics
Nullness not allowed (22)	[something] be/equals null [something] be equal/equivalent to null [soemthing1] or [something2] be/equals null ...
Nullness allowed (12)	[something] can/could be null [something] may/may not be null [something] can/could be equivalent/equal to null ...
Type restriction (10)	[something] be {not} [SpecType] [something1] or/and [something2] be {not} [SpecType] ...
Range limitation (23)	[something] $>$ / $<$ / $=$ [vale] [something] be not less/greater/larger/equal/equivalent than/to [value] ...

For users who simply want to use our tool, Fig. 3 displays the user interface of DRONE for the detection functionality. Once the target API libraries are imported, and the save directory is set, users could click the first 'analyze' button to enable invocation analysis. It will conduct AST and call hierarchy parsing, and the generated parsing result would be stored in the same directory (marked as A in Fig. 3). Code parsing and doc analysis could be enabled by clicking the next 'analyze' button. The constraints of both codes and directives would be analyzed. During this step, invocation parsing results in the previous step are reused and constraints in FOL format are produced. The file containing the constraints is generated and stored in the same directory (marked as B in Fig. 3). The last step is to enable the logic solver Z3 to deduce the potential inconsistency between the generated FOLs (marked

as C in Fig. 3). The right part of the view is the console, which displays the corresponding execution traces and logs of the individual analysis steps (marked as D in Fig. 3).

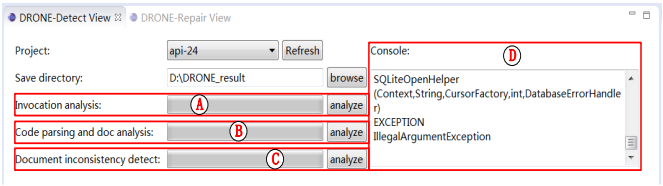

Fig. 3. DRONE UI—defect detection view

B. Defects Repair

Based on the defect report generated from the previous part, DRONE gives repair recommendations based on the predefined templates. Again for potential benefits of academic users, we collect some sample templates which are presented in Table II.

TABLE II
REPAIR RECOMMENDATION TEMPLATES

Constraints category	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]

Fig. 4 displays the user interface of DRONE for the repair recommendation related functionality. Users can load the defect reports generated in the previous phase, and enable the repair recommendation by clicking the 'start' button. Then the list of such defects could be browsed (marked as E in Fig. 3). The buttons in the left of the view panel could help navigate the list items. Once an item is left-clicked, the corresponding API method and its directives would be located and displayed in the code editor. Particularly, the first line of the API method will be highlighted (marked as G in Fig. 4). The repair recommendation will be given in the right part of the panel view (marked as F in Fig. 4).

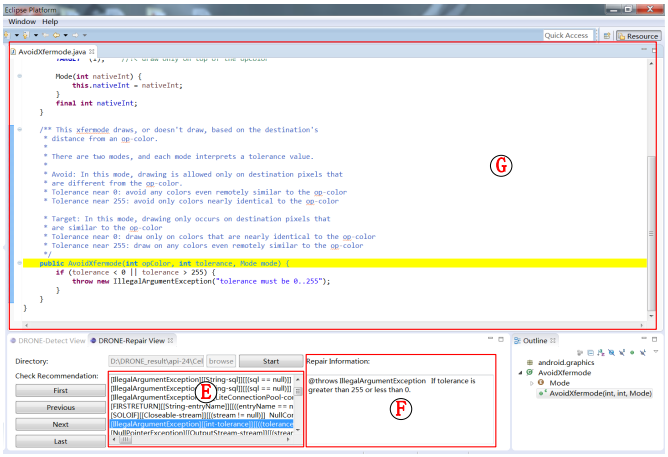

Fig. 4. DRONE UI—defect repair view

III. EVALUATION

To demonstrate the feasibility of DRONE, we have evaluated DRONE by extensive experiments with real-life API libraries. In this section, we mainly outline two case studies on DRONE's performance evaluation. The results are partially reproduced from [4], [5], to which we also refer interested readers for the details of the entire experiment sets. The first case study evaluates the directive defect detection ability. We use a subset of JDK APIs, mainly java.awt, javax.swing, and javaFX, from which the heuristics are extracted. In total, these three packages sum up to more than 1,100 kLoC. The metrics we consider for this part are mainly precision, recall and F-measure. In total, DRONE reported 1,689 defects, out of which 1,291 turn out to be true positives (TP), giving a precision rate of 76.4%. Meanwhile, the recall rate of the experiment is 83.8%, and accordingly, the F-measure is 79.9%. Table III shows the result.

TABLE III
RESULTS OF CASE STUDY 1: JDK APIS

Category	TP	FP	FN	Precision	Recall	F-measure
Nullness Not Allowed	233	77	7	0.752	0.971	0.847
Nullness Allowed	599	63	27	0.905	0.957	0.930
Range Limitation	406	245	120	0.624	0.772	0.690
Type Restriction	53	13	95	0.803	0.358	0.495
Total	1291	398	249	0.764	0.838	0.799

We note that, in order to validate the generalizability of our approach—particularly the heuristics—it's necessary to test DRONE over different APIs than those from which the heuristics are extracted. Hence we apply our approach to the latest Android APIs (level-24). For this purpose, we select the largest 12 packages with more than 400 kLoC code, as well as related documents. Table IV gives DRONE's performance on this set of APIs. Based on the result, we can observe that DRONE achieves a precision rate of 74.7% and a recall rate of 89.4%, which demonstrates its generalizability and, in particular, the efficacy of various heuristics which have been used. By these experiments, one can reasonably expect to deploy them for further similar projects.

TABLE IV
RESULTS OF CASE STUDY 1: ANDROID APIS

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

The second case study targets at DRONE's repairing ability. In accordance with the relevant work on the quality evaluation of text generation, such as [7], [8], we evaluate our recommended repairs in terms of accuracy, content adequacy, and conciseness & expressiveness. In this part, we hired 24

graduate students majoring in software engineering (SE) as subjects. These subjects are from different universities and all have at least five-year programming experience in Java and Android. We design four questions and use Likert-type scale (5-1) to rate the responses. The questions and the responses are given in Table V and Table VI respectively. The results show that subjects are satisfied with the repair recommendations and rate a majority of them with the highest score.

TABLE V
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)

TABLE VI
RESULT DISTRIBUTION OF EXPERIMENT 4

Result	Q1	Q2	Q3	Q4
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%)

IV. RELATED WORK

Defect defection tools have been widely investigated at the code level, but very few studies focus on the defects at the document level. Directives of API documentation and the evolution of API documentation were empirically studied by several researchers [9]–[11], and an empirically elicited taxonomy of directives characterizing API documentation was recently presented [3], [9]. The DRONE tool was conceived around a subset of directives related with parameter constraints. In this context, Buse and Weimer proposed an approach to analyze the exception related statements in source code and generate the documentation automatically [12]. Tan et al. developed a technique to detect the inconsistencies between code and comments [13]. Zhong and Su investigated the errors in API documentation and proposed an automatic approach to detect them [14]. Recent work by Panichella *et al.* [15] discussed techniques for automating SE tasks by leveraging summarization strategies. The approach behind DRONE leverages summarization techniques for the detection of constrains in API documentation, and differs from other pieces of work, as they are mostly related to syntactic errors and no repair recommendations are provided, [15]. Close work by Blasi and Gorla [16] proposed a tool, called RepliComment, that could detect comment clones and notify them to developers.

V. CONCLUSION

In this paper, we have demonstrated DRONE, the tool we designed and implemented to automatically detect and repair directive defects in APIs documentation. We also evaluated DRONE via two case studies against different and extensive qualitative and quantitative metrics. The results indicate that DRONE is a practical and accurate tool in detecting real-life API documentation defects, confirming its usefulness in improving the quality of API documentation. Future work aims at validating DRONE by contacting the original developers of the analyzed JDK libraries, as well as extending the tool to deal with defects present in other programming languages.

ACKNOWLEDGMENTS

This work was partially supported by the National Key R&D Program of China (No. 2018YFB1003902), 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 project SURF-MobileAppsData (No. 200021-166275).

REFERENCES

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

[2] M. A. Saied, H. Sahraoui, and B. Dufour, "An observational study on api usage constraints and their documentation," in *(SANER 2015)*. IEEE, 2015, pp. 33–42.

[3] 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.

[4] 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*, 2017, pp. 27–37.

[5] Y. Zhou, C. Wang, X. Yan, T. Chen, S. Panichella, and H. C. Gall, "Automatic detection and repair of directive defects of java APIs documentation," *IEEE Transactions on Software Engineering*, To appear.

[6] 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 *ASE 2015*. IEEE, 2015, pp. 12–23.

[7] 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 *International Conference on Software Engineering, ICSE 2016*, 2016, pp. 547–558.

[8] 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.

[9] 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.

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

[11] 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 *International conference on Automated software engineering*. ACM, 2014, pp. 785–790.

[12] 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.

[13] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *(ICST 2012)*. IEEE, 2012, pp. 260–269.

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

[15] S. Panichella, "Summarization techniques for code, change, testing, and user feedback (invited paper)," in *Workshop on Validation, Analysis and Evolution of Software Tests, VST@SANER 2018*, 2018, pp. 1–5.

[16] A. Blasi and A. Gorla, "Replicomment: identifying clones in code comments," in *ICPC 2018*. ACM, 2018, pp. 320–323.