Towards Detecting Inconsistent Comments in Java Source Code Automatically

Nataliia Stulova*, Arianna Blasi†, Alessandra Gorla‡, and Oscar Nierstrasz*
*Software Composition Group, University of Bern, Switzerland
†Faculty of Informatics, USI Università della Svizzera italiana, Lugano, Switzerland
‡IMDEA Software Institute, Madrid, Spain

***Abstract*—A number of tools are available to software developers to check consistency of source code during software evolution. However, none of these tools checks for consistency of the documentation accompanying the code. As a result, code and documentation often diverge, hindering program comprehension. This leads to errors in how developers use source code, especially in the case of APIs of reusable libraries. We propose a technique and a tool, *upDoc*, to automatically detect code-comment inconsistency during code evolution. Our technique builds a map between the code and its documentation, ensuring that changes in the code match the changes in respective documentation parts. We conduct a preliminary evaluation using inconsistency examples from an existing dataset of Java open source projects, showing that *upDoc* can successfully detect them. We present a roadmap for the further development of the technique and its evaluation.**

***Index Terms*—Documentation, Natural Language Processing, Software Quality**

I. Introduction

Program source code constantly evolves to meet new functionality requests, to fix issues or simply due to refactoring tasks. Unlike source code, whose correctness in case of changes is ensured by dedicated tools such as parsers, analyzers, compilers and linters, maintaining the correctness of its documentation is still responsibility of the programmer. This has two undesirable consequences: (a) existing documentation becomes outdated and is not necessarily fixed in a timely manner, and (b) documentation is not written in the first place to avoid the problem altogether [1], [2]. Program comments are often the only documentation format available to programmers, and they can aid significantly in program comprehension unless they have become outdated and misleading [3].

Listing 1 shows a method and its documentation from the `AdaptiveIsomorphismInspectorFactory` class of the JGraphT [4] library. We highlight with matching colors the code and the corresponding "doc comment" that relates input description, input validation and exceptional behavior. The doc comment mentions method parameters twice: implicitly in the free-form description part (line 2: "`one of the graphs`") and explicitly with the `@param` tags (lines 5-6).

```
1  /**
2   * Checks if one of the graphs is from unsupported graph type
3   * and throws IllegalArgumentException if it is. The current
4   * unsupported types are graphs with multiple-edges.
5   * @param graph1
6   * @param graph2
7   * @throws IllegalArgumentException
8   */
9  protected static void assertUnsupportedGraphTypes(
10   Graph graph1,
11   Graph graph2)
12   throws IllegalArgumentException
13 {
14   Graph [] graphArray = new Graph [] {
15    graph1, graph2
16   };
17   for (int i = 0; i < graphArray.length; i++) {
18     Graph g = graphArray[i];
19     if ((g instanceof Multigraph)
20       || (g instanceof DirectedMultigraph)
21       || (g instanceof Pseudograph)) {
22         throw new IllegalArgumentException(
23           "graph type not supported for the graph" + g);
24     }
25   }
26 }
```

Listing 1. Method body with its doc comment

We are interested in changes to this code snippet in revisions `b4805f5` and `a68071b`. The first one modifies both the signature and the body of this method, but not the respective documentation, introducing code-comment inconsistency:

```
- protected static void assertUnsupportedGraphTypes(
- Graph graph1,
- Graph graph2)
+ protected static void assertUnsupportedGraphTypes(Graph g)
    throws IllegalArgumentException {
- Graph [] graphArray = new Graph [] {
- graph1, graph2
- };
- for (int i = 0; i < graphArray.length; i++) {
- Graph g = graphArray[i];
```

The second revision updates the documentation but the "`one of the graphs`" to the old parameters remains unchanged, even though the method now has only one input:

```
- * @param graph1
- * @param graph2
+ * @param g
```

These two revisions are separated by **7.5 years**.

Such situations routinely occur because comments are still treated with less care compared to source code, even when they form part of the software (*e.g.,* official API documentation). Our aim is to change this *status quo* and introduce automated analysis techniques that would help in detecting and suggesting possible fixes for outdated method documentation comments if the corresponding source code changes. We propose the upDoc tool to support our technique. In future work we plan to integrate it into standard IDEs and CI systems, to give immediate feedback to developers about code-comment inconsistencies, and to suggest fixes that developers can accept or flag as false positives. Specifically, we propose a NLP-based

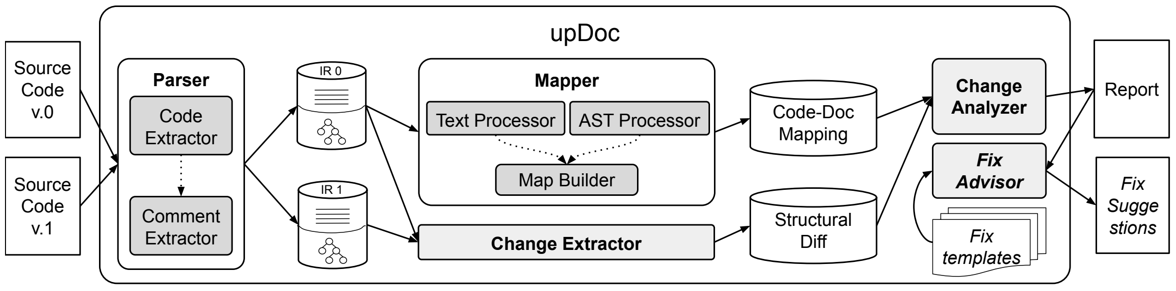

Fig. 1. upDoc workflow diagram

approach to inconsistency detection based on fine-grained *mapping between source code and documentation*.

Our motivation is supported by the results of two separate studies of co-evolution of comments and code on open source Java programs. Fluri *et al.* [1] aim to understand how changes in the source code affect the respective comments. They note that (a) comment change is triggered by the respective code change in less than half cases of code changes, and (b) for some projects, up to 10% of those comment changes happen more than 3 revisions later. Wen *et al.* [2] performe a large-scale *quantitative and qualitative* study that involved an automatic analysis of code changes in 1500 projects and a manual analysis of 500 commits. Authors confirm that *"in most of the cases, code and comments do not co-evolve"* simultaneously. Moreover, co-evolution occurs roughly in 20% of the time. These studies also stress that documentation comments have become first class citizens in code changes and refactoring, yet currently they are still treated more *syntactically* rather than *semantically*. State of the art techniques capture inconsistencies only by means of predefined patterns, and thus miss cases such as the reference to "`one of the graphs`" of the motivating example. With upDoc we aim to go one step further by semantically analyzing comments and code and thus detecting even non-obvious relations.

II. What's upDoc?

We schematically illustrate the full upDoc architecture and workflow in Figure 1. The tool consists of five principal modules combined in a *pipe and filter architecture*:

- *Parser* (Sec. III-A) is the pipeline entry point; it takes two revisions of a Java class source code and extracts tuples of method bodies and their doc comments into an intermediate representation.
- *Mapper* (Sec. III-B) creates a mapping between the structural units of the source code and the comment for each method of the code version *before* the change, *i.e.,* SourceCode v.0 in Figure 1.
- *Change Extractor* (Sec. V-B) compares the two representations produced by the *Parser* module for both source code versions comprising the change and creates a diff for both code and comment structural units for each method.
- *Change Analyzer* (Sec. V-C) inspects the changes in the two revisions of each method: if there are any detected changes in the code (as registered in the diff), it uses the mapping to check that all related comments are changed as well (and thus also registered in the diff). This module produces the final report on whether a comment should be further altered to match the code changes.
- *Fix Advisor* (Sec. V-D) combines the information about the necessary changes and a suitable fix template to automatically suggest reasonable modifications to the programmer.

III. Current Implementation

A. Source Code Parsing

We rely on the Javaparser [5] library to extract doc comments [6] and method-level AST nodes from the Java source code, and we interface with it while creating the structured intermediate representation of the code that upDoc works with. For each comment we extract both the free-text descriptive comment part and the sentences starting with tags `@param`, `@exception`, `@throws`, and `@return`. We disregard sentences with other tags as they typically contain information that is not related to the implementation. Note that each of these comment parts could be composed of multiple sentences, and a comment sentence is the smallest unit our technique works on. For each method we extract its signature AST nodes only, and we store the method name, return type name, thrown exception type names, and parameter names together with respective type names for each of them.

B. Code to Comment Mapping

We pre-process the full text of each doc comment by removing any HTML markup such as `<p>` and `<em>` tags and any Javadoc markup such as {`@code` ...}. Next, from each doc comment we extract the full period-terminated sentences using the Stanford CoreNLP toolkit [7]. Finally, we produce a bag-of-words (BoW) representation of each sentence by (a) splitting all code identifiers into constituents with regular expressions that allow for camelCase and special character splitting, (b) expanding abbreviations with a custom list (we plan to enhance it by incorporating the dataset of [8]), (c) reducing each word to its stem, and (d) filtering out stop-words with the "Short English stopwords list" [9].

We use AST-based representations of source code, just as ChangeDistiller [10] tool and the GumTree framework [11]. AST-based representation allows us to vary the granularity of the source code elements (as AST nodes at different depths)

TABLE I
SIMILARITY MEASURES SENSITIVITY

Comment	[1:illeg,1:argument,1:throw,1:one,1:except,1:check,1:unsupport,1:type,2:graph]		
		Word Mover's Distance	Cosine Similarity
Method (BC)	[1:illeg,1:argument,1:void,1:assert,1:except,1:unsupport,1:type,4:graph,1:first,1:second]	**70%**	**75%**
Method (AC)	[1:illeg,1:argument,1:void,1:assert,1:g,1:except,1:unsupport,1:type,2:graph]	**66%** (-4%)	**75%** (no score change)

mapped to the comment text. We build the bag-of-words representation of AST nodes (currently, only method signature nodes) similarly to how we do it for the comments.

We produce a many-to-many mapping between the AST nodes of source code and sentences of the comments. The mapping reflects how *similar* code is to its relative natural language documentation comment. This depends on the vocabularies of the units of these two elements. For a code and a comment unit to be related and included into the mapping the similarity score of their BoW representations must be higher than a predefined threshold. There are various metrics to assess the similarity between two texts. Some of them, like *cosine similarity*, can only assess lexical similarity. Others, such as Word Mover's Distance [12] have a *semantic* understanding of natural language words. It is possible to employ them thanks to the existence of many pre-trained models [13][14].

To illustrate the advantage of a semantic understanding of text over a more naive one, consider once again the code example of the Listing 1. In Table I we report similarity scores of the BoW representations of the comment text (which did not change), and the method signature before and after the change (BC and AC respectively). Using Word Mover's Distance upDoc can relate the method parameters `graph1` and `graph2` to the independent clause of the first sentence of the method doc comment ("`Checks if one of the graphs is from unsupported graph type`"). When `Graph graph1` and `Graph graph2` disappear to leave only `Graph g`, the semantic understanding detects a decrease of the similarity with respect to the comment. With cosine similarity upDoc does not detect this inconsistency as the vocabulary is still too syntactically similar, despite the change.

We implement both metrics in upDoc as one of our goals is to study performance baselines for code-coment inconsistencies detection using simple BoW representation. These baselines then can be used to evaluate how our technique is competitive with more elaborate approaches that rely on more advanced NLP techniques and semantic models of code and comment vocabularies.

IV. PRELIMINARY EVALUATION

Our evaluation of upDoc assesses whether our approach to code-comment mapping produces reliable links between method implementation and documentation. For this purpose we rely on the dataset produced by Wen et al. [2] of 500 commits of supposed documentation fixes for Java methods. In our experiment we inspect the differences between code-comment mappings in two source code versions. We run upDoc on the version before the commit, which is supposed to have at least one inconsistency, and on the commited version, which is supposed to fix inconsistencies. As upDoc evaluates the code-comment consistency based on similarity scores as described in Section III-B, our **research hypothesis** is that *upDoc would report higher similarity scores in a mapping for the fixed version.*

We focus on the first 50 entries of the dataset, and we manually analyze the results of upDoc runs. We first check the quality of these entries and discard 8 of them because none of the changes involved comment lines (*i.e.,* these are false positives admittedly included in the dataset [2]). We further discard entries that do not involve comments that upDoc focuses on: 4 documentation changes affecting only class-level doc comments (either general class or field descriptions), and 18 entries that involve changes only in comments within method bodies (*i.e.,* implementation comments which upDoc does not process yet). This leaves us with 20 commits, each involving one or more changes in method doc comments which we use to evaluate mappings produced by upDoc.

For these 20 commits, which amount to a total of 67 changes, we follow this evaluation protocol: 1) We run upDoc on the version right before the commit, and we store the similarity score for each method and corresponding comment affected by the change. 2) We run upDoc on the supposedly fixed version. 3) We compare the similarity score for each method signature and each corresponding doc comment sentence, before and after the change. We reportthe following:

- In **50 cases** the similarity **scores improve** as expected. Nearly half of these cases (24 out of 50) are trivial cases where there was no documentation and developers added some. 26 cases are instead actual fixes or significant semantic improvements to the comment text.
- In **10 cases** the similarity **scores did not change**. Manual analysis reveales that all these changes are either minor formatting fixes (*e.g.,* adding/removing white spaces and new lines) or other minor edits that do not change the semantics of the documentation. The results produced by upDoc are thus expected, since these changes do not address a real code-comment inconsistency.
- In **7 cases** upDoc reports **unexpected decreases** in the similarity scores. The manual analysis shows that these are all due to current limitations of the prototype, which at the moment analyzes only method signatures. Details included in the documentation indeed could match similar elements in the method body, but they do not always match the signature alone.

On this small dataset upDoc achieves the same results both running with Word Mover's Distance and cosine similarity. As confirmed in previous research [15], [16], developers tend to write documentation with a vocabulary that is close to the one composing code entities names. It is thus not surprising that, on a limited dataset of 67 changes, a clear advantage of Word Mover's Distance may not emerge. However, measures like cosine similarity cannot grasp inconsistencies like the one in Listing 1. Given the higher flexibility of Word Mover's Distance, we think it is worth keeping a semantic understanding of natural language to face complex code-comment inconsistencies that exist in practice.

upDoc's mapping between methods and doc comments accurately reflects inconsistencies in 90% of the cases.

V. Ongoing and Future Work

A. Improving Parsing and Mapping

We now list improvements that would allow upDoc to better exploit all the useful information conveyed by both the natural language comments and the code, thus making the mapping between them more precise and reliable.

We plan to enhance our current bag-of-words representations to include the original non-split identifiers appearing in the comment text and in the source code, increasing the mapping robustness as implemented in [15].We also plan to add a constituency parsing step to split compound sentences and construct a more fine-grained mapping. Moreover, we plan to add synonyms support both using common English synonym sets like WordNet[17] and software-specific ones like SWordNet[18]. At the same time we can benefit from the coreference resolution facilities of the CorefAnnotator component of CoreNLP [7], *e.g.,* in cases like one in Listing 1 in lines 2-3: "`it`" -> "`graph`". We would also include inline and block comments for method-level mapping and investigate the efficiency of our technique in class-level mapping. We also plan to support more kinds of AST nodes for the mapping creation, starting with those considered in [2]. Another immediate direction is the source code IR enhancement with the information from control and data flow analyses, which would allow to establish links between different AST nodes. For example, in the code example of the Listing 1 it would allow us to relate the variable `g` and the method parameters `graph1` and `graph2`, thus linking the independent clause of the first sentence of the method doc comment ("`Checks if one of the graphs is from unsupported graph type`") in the line 2 and the conditional expression node in the lines 19-24.

B. Structural Change Extraction

Working with AST-based representation of the source code allows us to filter out any purely syntactic changes (both for comments and code), such as whitespaces and formatting edits. We are primarily interested in detecting AST nodes that have been deleted and modified in the change, as they are likely to require a matching change in the existing comment text. Code addition is not strongly associated with comment changes, but rather with new comment addition if any [19], so detecting changes with added AST nodes has a lower priority for us. We will build the *Change Extractor* component of upDoc around the source code differencing functionalities of the GumTreeDiff framework [20] to benefit from the AST-based diffs of the source code under change.

C. Inconsistency Detection

According to previous research studies, it is more likely that a change in the code should have a corresponding change in its documentation rather than the reverse [1], [21]. To detect such inconsistencies we will combine a list of AST nodes that are marked as modified or deleted in the source code diff, and a list of related comment sentences for each AST node as present in the mapping. Iterating through the related sentences of the node under change we will check if all related sentences are present in the diff and warn the programmer if not. In case of code modification (or addition) upDoc will issue a warning if the new code does not have any relation to the comment text (*e.g.,* common identifiers or domain terms). In the case of code deletion upDoc will also report the lines in comment that are most likely affected by the change.

D. Suggesting Fixes

Our final goal is to be able not only to notify the developer about the mismatch, but also suggest a possible fix. We would take advantage of the source code mining techniques developed for automatic comment generation based on code clone analysis [22], and software changes guiding based on previous change history [23]. We would build on these previous works to produce a fix template set, and offer template filling functionality for the concrete code snippets under change.

E. Technique Scalability and Evaluation

We would like to see how much effort it takes to adapt our technique to other programming and natural languages.For this latter task we would leverage the recent advances in transfer learning. We also pursue several goals for enhancing the experimental evaluation. The primary goal is to obtain a few baseline performance rates for the different mapping granularity levels, starting with just the BoW mapping approach presented in this paper. The second one is to scale up the evaluation to more projects, and the full data set of [2] is of great practical interest.

VI. Related Work

Arnaoudova *et al.* [24] analyze *linguistic antipatterns* (LAs), i.e., poor software documentation practices that result in code-comment discrepancies. Aghajani *et al.* [3] show that LAs lead to a 29% higher chance of introducing bugs, highlighting the negative effect that outdated documentation has on code quality. The Catcher tool [25] can detect API misuses that lead to program crashes, and the study shows that such misuses are often not correctly documented. In [26] the authors confirm that *"Developers prefer documentation that is correct, complete, up to date, usable, maintainable, readable and useful."*

The above findings emphasize the need for tools that automate and assist in comment updates during software evolution.

Zhou *et al.* [27] propose a first order logic-based approach to detect code-comment inconsistencies in API documentation. It limits its attention on documentation sentences and program statements describing method exceptional behavior on variable nullness, type errors, and value range limitation. Similarly, the JavadocMiner tool [28] allows one to model code-comment named entity relations with ontologies, and describes metrics to evaluate comment quality, and its coherence with respective source code. We see this line of work extremely valuable for improving accuracy in code-comment matching, but for the broader applicability of upDoc we aim not to rely on specific heuristics or pattern-matching mechanisms.

Learning-based techniques can be used for bidirectional code-comment modeling and inference. Phan *et al.* [29] present a preliminary evaluation of an approach for bidirectional inference of behavioral exception conditions and their documentation based on statistical machine translation. In [30] Liu et al. propose a machine learning-based method for detecting outdated block and line comments in method bodies during code changes.

Several research lines exploit the bidirectional correspondence of code and comments in different software engineering activities, such as testing (@tComment [16], JDoctor [15]), clone/software similarity detection (CLCDSA [31], CroLSim [32]), code comprehension (APIBot [33], MULAPI [34]), and traceability links recovery [35]. Such techniques need the documentation to be reliable and well-maintained, thus we believe they would benefit from tools like upDoc.

VII. Conclusion

This paper outlines a techinique to automatically detect code-comment inconsistencies during code evolution based on a mapping between the source code and its comments. We have implemented a prototype tool upDoc and evaluated the accuracy of its core component, the mapping module, on a dataset of code-comment inconsistency fixes. Our results show good reliability of the mapping method to detect code-comment inconsistencies. We believe this work sets a foundation for future efforts in assisted maintenance of code-comment consistency for many software engineering tasks.

Acknowledgement

The first author would like to thank Viktor Kunčak and members of EPFL LARA lab for useful discussions. This work was partially supported by the Spanish Government through the SCUM grant RTI2018-102043-B-I00 and by the Madrid Regional projects BLOQUES and MFOC.

References

[1] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *SQJ*, vol. 17, no. 4, 2009.

[2] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *ICPC*, 2019, pp. 53–64.

[3] E. Aghajani, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on linguistic antipatterns affecting apis," in *ICSME*, 2018.

[4] "Jgrapht," 2020, https://github.com/jgrapht/jgrapht.

[5] "Javaparser," May 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2667379

[6] "Javadoc Oracle documentation," 2020, https://www.oracle.com/java/technologies/javase/codeconventions-comments.html.

[7] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *ACL*, 2014, pp. 55–60.

[8] C. Newman, M. J. Decker, R. S. AlSuhaibani, A. Peruma, D. Kaushik, and E. Hill, "An open dataset of abbreviations and expansions," in *ICSME*, 2019, pp. 280–280.

[9] "Default english stopwords list," 2020, https://www.ranks.nl/stopwords.

[10] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE TSE*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[11] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ASE*, 2014.

[12] M. J. Kusner, Y. Sun, N. I. Kolkin, and K. Q. Weinberger, "From word embeddings to document distances," in *ICML*, 2015, pp. 957–966.

[13] "Word2vec," 2020, https://code.google.com/archive/p/word2vec/.

[14] "Glove," 2020, https://nlp.stanford.edu/projects/glove/.

[15] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. Delgado Castellanos, "Translating code comments to procedure specifications," in *ISSTA*, 2018, pp. 242–253.

[16] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing Javadoc comments to detect comment-code inconsistencies," in *ICST*, 2012, pp. 260–269.

[17] G. A. Miller, "Wordnet: A lexical database for English," *CACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995.

[18] J. Yang and L. Tan, "SWordNet: Inferring semantically related words from software context," *EMSE*, vol. 19, no. 6, Dec. 2014.

[19] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *WCRE*, 2007, pp. 70–79.

[20] "Gumtreediff library," 2020, https://github.com/GumTreeDiff/gumtree.

[21] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *ICPC*, 2006, pp. 35–45.

[22] E. Wong, Taiyue Liu, and L. Tan, "CloCom: Mining existing source code for automatic comment generation," in *SANER*, 2015, pp. 380–389.

[23] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *ICSE*, 2004, pp. 563–572.

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

[25] M. Kechagia, X. Devroey, A. Panichella, G. Gousios, and A. van Deursen, "Effective and efficient API misuse detection via exception propagation and search-based testing," in *ISSTA*, 2019, pp. 192–203.

[26] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software Documentation Issues Unveiled," in *ICSE*, 2019, pp. 1199–1210.

[27] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing APIs documentation and code to detect directive defects," in *ICSE*, 2017, pp. 27–37.

[28] N. Khamis, R. Witte, and J. Rilling, "Automatic quality assessment of source code comments: the JavadocMiner," in *NLDB*, 2010, pp. 68–79.

[29] H. Phan, H. A. Nguyen, T. N. Nguyen, and H. Rajan, "Statistical learning for inference between implementations and documentation," in *ICSE NIER*, 2017, pp. 27–30.

[30] Z. Liu, H. Chen, X. Chen, X. Luo, and F. Zhou, "Automatic detection of outdated comments during code changes," in *COMPSAC*, 2018.

[31] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, and K. A. Schneider, "CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation," in *ASE*, 2019, pp. 1026–1037.

[32] K. W. Nafi, B. Roy, C. K. Roy, and K. A. Schneider, "CroLSim: Cross Language Software Similarity Detector Using API Documentation," in *SCAM*, 2018, pp. 139–148.

[33] Y. Tian, F. Thung, A. Sharma, and D. Lo, "APIBot: Question answering bot for API documentation," in *ASE*, 2017, pp. 153–158.

[34] C. Xu, X. Sun, B. Li, X. Lu, and H. Guo, "MULAPI: Improving API method recommendation with API usage location," *JSS*, vol. 142, pp. 195–205, Aug. 2018.

[35] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE TSE*, vol. 28, no. 10, pp. 970–983, 2002.