Semantic-based Analysis of Javadoc Comments

Arianna Blasi[♠] · Konstantin Kuznetsov[◇] · Alberto Goffi[†] · Sergio Delgado Castellanos[♠] · Alessandra Gorla[♠] · Michael D. Ernst[‡] and Mauro Pezzè[†]

[♠]IMDEA Software Institute, Madrid, Spain
[◇]Saarland University, Saarbrücken, Germany
[†]Università della Svizzera italiana (USI), Lugano, Switzerland
[‡]University of Washington, Seattle, USA

Abstract

Developers often document their code with semi-structured comments such as Javadoc. Such comments are a form of specification, and often document the intended behavior of a code unit, as well as its preconditions. The goal of our project is to analyze Javadoc comments to generate assertions aiming to verify that a software unit indeed behaves as expected. Existing works with this goal mainly rely on syntactic-based techniques to match natural language terms in comments to elements in the code under test. In this paper we show the limitations of syntax-based techniques, and we present our roadmap to semantically analyze Javadoc comments.

1 Introduction

One of the most important activities in the software development process is to verify that software behaves as expected. As software evolves, it becomes more compelling to automate the verification process, since every change may introduce new issues, and detecting problems as soon as they are introduced allows to save significant time in debugging. Many techniques can automatically generate inputs to test specific software units [1, 2]. However, these techniques do not automatically check that software behaves as expected, but rather assume that developers manually write assertions for this purpose.

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017 (sattose.org). 07-09 June 2017, Madrid, Spain.

In our project we aim to analyze natural language specifications, written in the form of Javadoc comments, to automatically generate executable specifications of Java classes. Javadoc comments are semi-structured, since they use predefined tags to describe preconditions on method parameters (@param tags), postconditions on regular executions (@return tags), and on exceptional conditions (@throws tags). There exist already a few techniques that can turn Javadoc comments into assertions to check pre and postconditions of Java methods [3, 4], however they either focus on specific patterns (e.g. checking for non null values) or rely on simple syntax-based algorithms to match terms in the natural language comment to elements in the code to verify. Pattern-based and syntax matching limit significantly the ability of these techniques to automatically generate assertions. Take for instance the example in Listing 1. When analyzing the Javadoc specification of method isInDanger(), we would aim to generate an assertion that checks that this method returns True if a threat was found. Thus, for given range and threat values, we aim to generate an executable condition such as the following:

```
if isInDanger(range,threat)
    assertNotNull(searchForDanger(range,
        threat));
```

This example shows the challenges of our goal: we want to understand that in order to check the return condition of isInDanger(), the assertion has to use searchForDanger(), which is another method of the same class. A syntax-based technique would be able to match the natural language comment "if a threat was found" to a method only if its name were close enough to the very same terms used in the comment (e.g. threatFound(range, threat)). Unfortunately, it is not the case here, since searchForDanger() is quite distant from the terms used in the comment. In order to match the right method to be used in the assertion, we need to have the semantic knowledge that threat is

a synonym of danger, and actions to search for and to find are semantically related.

```
1  /**
2   * Checks if there is a credible threatening
        unit to this unit within a range of
      moves.
3   *
4   * @param range The number of turns to
      search for a threat within.
5   * @param threat The maximum tolerable
      probability of a potentially
      threatening unit.
6   * @return True if a threat was found
7   */
8  public boolean isInDanger(int range, float
     threat) {...}

10 /**
11  * Searches for a unit that is a credible
      threatening unit to this unit within a
      range.
12  * ...
13  */
14 public PathNode searchForDanger(int range,
     float threat){...}
```

Listing 1: Sample methods from the FreeCol project (`http://www.freecol.org`).

The goal of our work is thus to enrich existing techniques to automatically generate code assertions with semantic knowledge, hoping to improve their effectiveness.

The remained of the paper is structured as follows. Section 2 presents the state of the art, Section 3 presents some initial proposals on how to address the problem, Section 4 presents the evaluation we plan to perform and illustrates the future work.

2 State of the Art

There exist several works that use natural language processing techniques to analyze Javadoc comments for different purposes. Here we broadly classify them according to whether they solely rely on syntax-based algorithms or rather rely on some semantic knowledge.

2.1 Syntax-based approaches

@tComment, by Tan et al., parses Javadoc comments to produce executable code conditions testing null values [3]. The tool tries to syntactically match single words in the natural language text to method parameters. Thus, a comment such as "@param item has to be null" would produce the condition item != null. This technique shares the same goal of our project, but it focuses only on null cases. Toradocu, by Goffi et al., is more advanced since it can deal with cases other than null conditions [4]. Toradocu uses part-of-speech (POS) tagging, and matches subjects to code elements that should be checked, and predicates to either predefined patterns or methods that check properties. Thus, with a comment like "@throws NullPointerException if the list is empty", it would be able to match list to a method parameter that has a similar name, and would invoke method isEmpty() on the list instance to check whether it is empty. Toradocu though, can match terms in the natural language comments to Java elements (parameters and methods) only if they are syntactically similar (i.e., up to a maximum Levenshtein distance threshold). It thus would be unable to deal with the example presented in Section 1. With a different purpose in mind, Steidl et al. also aim to match words in comments to words used in method names [5]. Their intent is to measure the cohesiveness of these elements, and they propose this as a metric for source code quality. Khamis et al. also aim to assess the consistency of code and comments [6]. They use POS-tagging to analyze comments, but use only syntax-based heuristics to match comment elements to the code.

As we already anticipated, syntax-based techniques would fail at analyzing the comment presented in Section 1. In the second part of this section we present some techniques that use semantic equivalence to analyze code.

2.2 Semantic-based approaches

Similarly to @tComment and Toradocu, Pandita et al. aim to infer method specifications from natural language API comments [7]. Beside syntax-based techniques, they employ WordNet [8] to identify synonyms. WordNet, which is the most well known project to identify whether two terms are synonyms, has been built using regular English documents, and has therefore limited abilities in software domains. Thus, in order to identify more synonyms, Pandita et al. manually defined a limited set of relevant synonyms for the domains they considered. This, however, limits the applicability and generality of the approach. Tian et al. built SEWordSim, a database of terms similarity for the software engineering domain [9]. While the goal of this work is completely traversal to ours, SEWordSim might be a useful resource for our purposes. Though, the dataset is not publicly available.

As [5] and [6], Mcburney et al. also aim to assess the quality of code documentation [10]. However, differently from the previous techniques, they rely on several semantic similarity analyses to improve the accuracy. The challenge of matching code elements to words has been faced also by Deng et al., who propose a technique to process queries in natural language to retrieve the corresponding method that implements the requested functionality [11]. The technique works using models that have been trained on two separate corpora of code comments and code ele-

ments, and therefore can also detect semantically similar concepts. Sridhara et al., instead, aim to produce natural language descriptions of methods looking at their implementation [12]. The analysis of code elements relies on naming conventions and linguistic knowledge that has been gained on a corpus of Java programs. Zhai et al. aim to do the opposite, i.e., they automatically generate code snippets given a natural language specification [13]. The technique starts from a set of pre-defined primitive models, and combines them to generate the desired implementation. Sridhara et al. also compared different semantic similarity techniques based on WordNet. Their study concludes that to apply semantic analysis to software one would either have to augment WordNet with relations that are specific to software, or would have to retrain the model with appropriate probabilities.

3 Semantic-based Javadoc analysis

The goal of our analysis is to translate Javadoc comments into code assertions considering the semantics of words. Our approach consists of three steps:

1. For each predefined Javadoc tag (e.g. `@return`) we extract the natural language description of an assertion under which the statement should hold. In our example, the method should return `True` under the condition "a threat was found". We then transform each comment into a set of propositions using natural language processing techniques;
2. We analyze the code of the target class, and extract its fields and method signatures: name, arguments, as well as return type;
3. Finally, we exploit semantic similarity to map each part of the comment predicate to a specific code element and generate assertions.

3.1 Comments processing

Given a Javadoc comment, we use the Stanford Parser[1], to 1) identify multiple sentences, 2) generate *Stanford Dependencies*(SD)—a representation of grammatical relations between words—for each sentence. Given the textual relations defined by SD, we finally 3) extract *subjects* and related *predicates*. The subject in a phrase defines the primary topic of a statement, whereas the predicate, which is the reminder of the sentence, characterizes the subject in some way. In our example, "if a threat was found" contains the subject "threat", and the predicate "was found". When the subject is represented by multiple words, we consider all of them, i.e., a compound noun and its modifiers.

[1] `https://nlp.stanford.edu/software/lex-parser.shtml`

Due to the richness of natural language, it is hardly feasible to correctly identify the meaning of each predicate. To ease this step, we follow the pattern-matching approach presented in Toradocu [4]. The authors defined a set of patterns that commonly appear in Javadoc comments to which a proposition can be matched. This set includes patterns for sentences in active and passive form, as well as copula. The predicate can thus be comprised of verb, copular verb, complement, negation modifier, and also passive auxiliary. Finally, we may apply *lemmatization* to reduce the inflectional forms of a word to a common base form using a vocabulary and morphological analysis. Thus, "found" would be converted to the base lemma "find".

3.2 Code processing

Once we have propositions represented by pairs of <`subject`, `predicate`> we aim to find an association between each tuple and a Java code element of the class under test. Thus, we collect code elements that can be possible candidates for the mapping. The match for a subject may happen with: a parameter of the method the comment is referring to; a method of the class under test; or the instance of the class itself. A predicate, in turn, can correspond to the public fields and methods of the subject's declared type. Predicates containing arithmetic or null comparisons typically do not have a proper matching method call, and we thus match them to predefined patterns. Finally, we split each compound camel-cased term into separate words. Hence, the `searchForDanger` method name would be converted to "search for danger".

3.3 Semantic mapping

To generate assertions we need to associate propositions to elements in the Java code. The basic approach would utilize a direct syntactic comparison of words. However, it cannot cover more complex cases. In our example it would work if there were a method `findThreat()`. More advanced techniques would apply lexical matching supplied with Levenshtein distance. Nevertheless, it would fail to identify the `searchForDanger` method, since neither *threat* and *find* have syntactic elements in common with their corresponding match.

To improve the automatic matching, we propose to leverage *semantic relationships* between words. Often developers use synonyms while writing comments for their code. For instance, they could use "lower" in a comment and use "min" in a method signature. These words are semantically *equivalent* and can substitute one another. To resolve this kind of relations one can use WordNet or WordNet-like domain specific datasets [8, 7, 9].

However, it cannot capture all equivalence relations. In our example, WordNet did not list "danger" as a synonym of the term "threat". Besides, the terms "find" and "search" are not in the relation of equivalence.

Dealing with conceptual relationship is more complex. For instance, the propositions "a graph contains a vertex" and "a vertex exists in the graph" express strong *association*, though, the verb "contain" is not equivalent to "exist". Hill at al. [14] emphasize the contrast between *synonymous similarity* and *associations*. Word embedding has proved to be a powerful approach to represent word relations. It embeds words in a high-dimensional vector space such that words that appear close in the source text are close in the final vector space. One of its most popular implementation is Word2Vec [15] — a two-layer neural network model created by Google in 2013. Another one is GloVe, developed by the Stanford NLP group [16]. While they differ in algorithms used for learning the model, they can be both used for the same aim.

In our preliminary tests we found that the publicly available pre-trained word vectors of the GloVe model based on Common Crawl dataset[2] already produce good results, as they identify relations such as: "if vertex exists" $\rightsquigarrow$ `graph.containsVertex(v)` and "if the graph contains the edge" $\rightsquigarrow$ `graph.getEdge(v1,v2)`

In order to precisely capture the semantic peculiarities of the domain, we plan to train a Word2Vec model by feeding it with Javadoc comments. Of particular interest is the Phrase2Vec model [17], which learns continuous distributed vector representations for short text snippets. Along with its ability to recognize word collocations, it seems to be possible to identify domain specific common phrases like "less then zero", embed them into a vector space, and identify their relations, e.g. to the word "negative". Moreover, Mikolov at al. [18] showed that the Word2Vec model can be used to infer missing dictionary entries by learning a linear projection between vector spaces that represent each language with the help of mapping from small bilingual data. It is worth investigating whether the comments corpus can be translated to the code name space is such a way.

In many of the cases we considered, both the predicate and the method are made by multiple words. However, neither Word2Vec nor GloVe standard models support these cases; they produce vector representations for single words and only for few common phrases. In order to compare multiple words at once, we may use Word Mover's Distance (WMD) [19]. This distance function measures the semantic distance between two text snippets as the cumulative distance that all words in the first text have to exactly match the words in the second text. Using WMD we plan to look for all possible candidates among the extracted code elements. Thus, the <`threat`, `was found`> would match `searchForDanger` method. This match would finally produce the assertion of our example:

```
this.searchForDanger(range, threat)!=null?
    result==true:result==false
```

[2] http://nlp.stanford.edu/data/glove.840B.300d.zip

4 Evaluation Plan and Future Work

In order to evaluate the effectiveness of semantic analysis we are going to analyze various Java projects that are well documented with Javadocs (e.g. Guava, JGraphT).

For every Javadoc comment referring to a method, we plan to identify code elements that are most semantically related to it. Besides, we want to run syntactic-based matching and compare the results. We want to answer two research questions. First, we want to prove that the semantic approach covers all the cases that the syntactic one can resolve. Second, we want to ensure that semantic technique can be more efficient and can produce correct matching for complex samples which cannot be handled by the syntactic approach. For this end we plan to manually build the ground truth representing the correct mapping. Here, we plan to rely on the assertions in test code written either by developers or by us directly. This dataset will allow us to evaluate the *precision* and *recall* of the matches between code elements and comments.

Finally, we are going to generate assertions and have a quantitative evaluation based on how many assertions correctly compile. We also plan to evaluate how our generated assertions improve test suites in terms of code coverage.

To improve the chances of matching comments to code elements, in the future we plan to employ static and dynamic analysis of method bodies to have more information about their behavior. Moreover, we plan to analyze natural language text that is not included in predefined tags. Such text, in fact, may contain useful information that may be implicit in the parts of comments we currently consider.

References

[1] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-oriented Software," in *In Proc. 19th ACM SIGSOFT*, ESEC/FSE '11, pp. 416–419, ACM, 2011.

[2] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed Random Testing for Java," in *22Nd ACM SIGPLAN*, OOPSLA '07, pp. 815–816, ACM, 2007.

[3] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing Javadoc comments to detect comment-code inconsistencies," in *In Proc. 5th ICST*, pp. 260–269, 2012.

[4] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic Generation of Oracles for Exceptional Behaviors," in *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '16, pp. 213–224, 2016.

[5] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *In Proc. 21st ICPC*, pp. 83–92, 2013.

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

[7] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language API descriptions," in *In Proc. 34th ICSE*, pp. 815–825, 2012.

[8] G. A. Miller, "WordNet: a lexical database for English," *ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[9] Y. Tian, D. Lo, and J. Lawall, "SEWordSim: Software-specific word similarity database," in *In Proc. 36th ICSE*, pp. 568–571, 2014.

[10] P. W. McBurney and C. McMillan, "An empirical study of the textual similarity between source code and source code summaries," *Empirical Software Engineering*, vol. 21, no. 1, pp. 17–42, 2016.

[11] H. Deng, G. Chrupala, N. Calzolari, K. Choukri, T. Declerck, H. Loftsson, B. Maegaard, J. Mariani, A. Moreno, J. Odijk, *et al.*, "Semantic approaches to software component retrieval with English queries.," in *LREC*, pp. 3248–3252, 2014.

[12] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *In Proc. 33rd ICSE*, pp. 101–110, 2011.

[13] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin, "Automatic model generation from documentation for Java API functions," in *In Proc. 38th ICSE*, pp. 380–391, 2016.

[14] F. Hill, R. Reichart, and A. Korhonen, "Simlex-999: Evaluating semantic models with (genuine) similarity estimation," *Computational Linguistics*, 2016.

[15] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[16] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global Vectors for Word Representation," in *EMNLP*, vol. 14, pp. 1532–1543, 2014.

[17] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, pp. 3111–3119, 2013.

[18] T. Mikolov, Q. V. Le, and I. Sutskever, "Exploiting similarities among languages for machine translation," *arXiv preprint arXiv:1309.4168*, 2013.

[19] M. J. Kusner, Y. Sun, N. I. Kolkin, K. Q. Weinberger, *et al.*, "From Word Embeddings To Document Distances," in *ICML*, vol. 15, pp. 957–966, 2015.