Statistical Learning for Inference between Implementations and Documentation

Hung Phan*, Hoan Anh Nguyen*, Tien N. Nguyen†, and Hridesh Rajan*
*Computer Science Department, Iowa State University, Email: {hungphd,hoan,hridesh}@iastate.edu
†Computer Science Department, University of Texas at Dallas, Email: tien.n.nguyen@utdallas.edu

***Abstract*—API documentation is useful for developers to better understand how to correctly use the libraries. However, not all libraries provide good documentation on API usages. To provide better documentation, existing techniques have been proposed including program analysis-based and data mining-based approaches. In this work, instead of mining, we aim to generate behavioral exception documentation for any given code. We treat the problem of automatically generating documentation from a novel perspective: statistical machine translation (SMT). We consider the documentation and source code for an API method as the two abstraction levels of the same intention. We use SMT to translate documentation from source code and vice versa. Our preliminary results show that the direction of statistical learning for inference between implementations and documentation is very promising.**

***Keywords*-API documentation generation, machine translation**

I. Introduction

Software libraries play an important role in software development. Developers use the application programming interface (API) elements including API classes and methods to access the provided functionality. *API specifications* are the conditions on the usages of those API elements that a program needs to follow for the libraries to work properly. *API preconditions* are the interface specifications on the behaviors of an API method that a client program must honor before calling it. A type of preconditions, called *behavioral exception specification*, which specifies the conditions for the cases of exceptions being thrown when a client program does not honor them. In software engineering, documentation on *behavioral exceptions* has been shown to be practically useful because they not only allow developers of a client program to understand better how to use the APIs [9], but also enable the automated tools to verify or to test different properties of a program [11].

Unfortunately, the lack of API documentation in general has been hindering the users in understanding and using the libraries in the correct manner. The reason is that manually writing and maintaining documentation over time is challenging [4]. The libraries' developers must spend additional time and effort for documenting and updating API usages.

Several approaches have been proposed to support automated API usage documentation. The methods in the *mining software repositories* area have applied data mining to derive or check documentation/specifications on API usages from existing repositories [7], [11]. For example, GrouMiner [7] considers *the usages of the APIs* at the call sites in the client programs of the APIs to derive the conditions *regarding the usage orders or temporal orders* among the API calls. A few approaches [8] combine both static analysis and source code mining. Tan *et al.* [11] analyze the implementation code and Javadoc documentation to detect inconsistencies.

```
public String substring(int begin, int end) {
  if (begin < 0) throw new StringIndexOutOfBoundsException(begin);
  if (end > count) throw new StringIndexOutOfBoundsException(endIndex);
  if (begin > end) throw new StringIndexOutOfBoundsException(end−begin);
}
```

Fig. 1: Implementation of `java.lang.String`.

Other approaches, e.g., Buse and Weimer [2], use *program analysis* on code changes to produce documentation. Sridhara *et al.* [10] use syntactic rules in a Java program to summarize the overall actions in a method. Zhai *et al.* [12] construct models for API functions by analyzing the documentation. Lei *et al.* [6] translate English specifications to C++ input parser.

Despite their successes, none of the above approaches could perform the inference in both directions between source code implementations and API documentation.

II. API Documentation Generation with Statistical Machine Translation

Formulation. In this work, to ease the documenting of API usages, we aim to generate the behavioral exception documentation of any given code. We investigate the problem of automatically generating documentation from a novel perspective. We consider the documentation and source code for an API method as the two abstraction levels of the same intention, which is the function of the API method itself. While the source code describes the behavior of the API method in a programming language that provides the instructions to perform the task, the API documentation outlines the behavioral conditions before or after the method is called (i.e., before and after the task is performed). Therefore, we treat *the API documentation generation as a machine translation problem*.

The input of the process is the source code of a given API method *without* its documentation. The output is the behavioral exception documentation of the given API method. For example, in Java Development Kit (JDK) library, the API substring(begin,end) is used to extract a sub-string from the begin index to the end index. According to JDK documentation, the API *"Throws StringIndexOutOfBoundException if the begin index less than zero or the end index is greater than the*

978-1-5386-2675-7/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE-NIER.2017.9

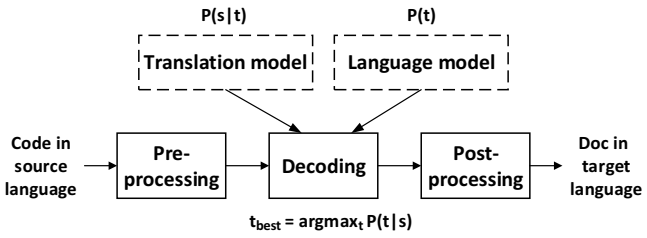

Fig. 2: Statistical Machine Translation [5]

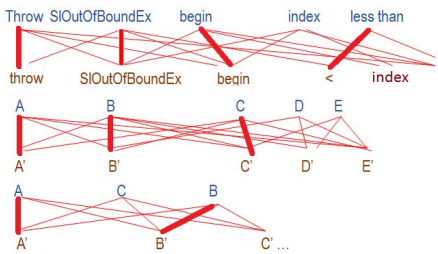

Fig. 3: Mapping Model

length of the string or the begin index is greater than the end index"(*). The implementation for substring is shown in Fig. 1. Documentation generation in this case is as follows: given the implementation of an API method as in Fig. 1, a tool will generate the exception condition documentation for the API as listed above at (*).

Approach. To be able to generate the documentation, we use a statistical machine translation (SMT) model as illustrated in Fig. 2. SMT [5] is a method for automated translation of natural languages. It uses statistical models whose parameters are trained from a corpus of source and translation texts [5]. In SMT, translation is the process to produce a sequence t in the target language T from a sequence s in the source language S, where t is the most likely sequence according to the translation and language models. The language model specifies how likely sequence t occurs in the target language while the translation model specifies how likely two sentences s and t co-occur in the corresponding texts of the two languages in the training corpus. Those two models in SMT maintain the probabilities $P(t)$ (probability of the sentence t occurring in the target language) and $P(s|t)$ for all possible sequences s and t (probability of the mapping between t and s). Those probabilities are estimated from the existing source and translated texts via a training process. Once trained, the SMT model can be used for translation. Details on SMT can be found in [5].

To apply SMT to documentation generation problem, for the language model on the target side, depending on the chosen documentation or specification language, one could use a proper language model. For example, in the case of Javadoc for JDK APIs, one could use a language model for a natural language such as n-gram, deep learning language model, etc. In the case of more formal specifications, a logic-based language model could be used. In our experiment, we used the n-gram language model for the behavioral exceptions in Javadoc documentation of the APIs in JDK.

For the mapping model, one could use a sequence-to-tree mapping model in which a sequence represents a phrase in the Javadoc documentation and an abstract syntax tree (AST) represents the respective source code. Currently, in our experiment, we use phrase-based mapping with IBM Model [1] and the Expectation-Maximization (EM) algorithm. Let us summarize how it works.

The goal of the mapping model is to derive the mappings between the sequences in documentation and the corresponding sequences in source code with the associated likelihoods. Let us use L_S and L_T to denote two sets of sequences in the documentation and in the source code, respectively. They are often called a parallel corpus. Assume that s and t are two corresponding sequences, and $s = s_1 s_2 ... s_m$ in L_S and $t = t_1 t_2 ... t_l$ in L_T. The model takes those two sets L_S and L_T to compute the likelihoods of mappings between a sequence in the documentation and a sequence in the source code.

The model proceeds in two phases. The goal of the first phase, called *single element alignment*, is to derive the mappings between pairs of words and code tokens. The second one, *sequence alignment*, is to derive the mappings between pairs of sequences of words in the documentation and code tokens.

Single element alignment. In the first phase, we use the EM algorithm to iteratively find the optimum alignment between the words and the code tokens. At first, the model attempts all possible alignments. For each pair (s,t), the model initially assigns a small weight for each pair of word in s and code token in t. For example, the pairs of (*"Throw"*, throw), (*"StringOutOfBoundException"*, StringOutOfBoundException), (*"less than"*, <), etc. are assigned with some initial weights. The same treatment is applied to all pairs of (s,t) in L_S and L_T.

At each iteration, the model adjusts the weights of the mappings/alignments based on the occurrence frequencies of those mappings appearing in the parallel corpus. It continues to optimize the alignments in all of the corpus, until all alignments are exhaustively found or no more improvement between iterations. The final alignments between the words and the code tokens are denoted by the lines in Fig. 3. The thicker the line, the higher score of the alignment. Details are in [1].

Sequence alignment. In the second phase, the model expands the surrounding words of the aligned words/tokens to get larger aligned sequences. The steps for computing the sequence-to-sequence alignments are as follows. The model adds the pairs of symbols that were aligned by the single element alignment model into a *sequence mapping table* with their mapping probabilities. Second, the model collects all sequence pairs that are consistent with the single alignment, i.e., the sequence alignment has to contain all alignments for all covered symbols. A sequence pair is required to include at least one alignment point. Then, the model iterates over all target sequences to find the ones closest to source sequences, and adds those sequence pairs and mapping probabilities to the sequence mapping table.

Finally, the pairs of mappings in the mapping table are ranked. The model uses the table for the translation.

|Gen|	|Ref|	Pre.	Rec.	BLEU	Matched	Same	Close	Incorrect
10	9	76%	80%	56%	31%	23%	22%	24%

Fig. 4: Code-to-Documentation Inference Result.

Component					Complexity	
Global	Receiver	Para-type	Para-value	Para-method	Simple	Complex
62%	72%	94%	54%	42%	58%	54%

Fig. 5: Code-to-Documentation Inference Result Analysis.

III. Preliminary Results

We conducted experiments to preliminarily evaluate our approach. We focus on the following research questions:

RQ1. How accurately does SMT generate behavioral exception documentation for given source code?

RQ2. How accurately does SMT generate source code from given behavioral exception documentation?

A. Data Collection and Settings

To train the SMT model, we need to have a parallel corpus of pairs of source code and the respective documentation. We built that corpus from the implementations and Javadoc documentation of the methods from Java Development Kit (JDK) library. We parsed 1,869 JDK classes with 6,802 methods. For the Javadoc documentation of each method, we performed NLP processing on the @throws and @exception description. For the code of each method, we extracted the exceptional flow graph (EFG), which is a slice of the Program Dependence Graph (PDG) of the method leading to all exception exit points. We consider all throw statements as exception exit points. Then, we collected all code tokens along the slice, forming the EFG. The pairs of such documentation sentence and their sequence of the code tokens along the slice for a method were collected into a parallel corpus of documentation and source code. The methods that do not have exceptional behavior were discarded. In total, we have 1,524 pairs in the parallel corpus. We used 1,424 pairs for training the *n*-gram language model and the mapping model, and 100 pairs for testing. We use Phrasal [3], a phrase-based SMT tool (n=7) in our experiment. The data and results of our experiments are available at https://drive.google.com/open?id=0B9-QRtybJfCITC1BVEFpWTZQQlE.

B. Experiment 1. Inferring documentation from code

After training the model with the parallel corpus of 1,424 pairs, we ran the trained model on the source code sequences of 100 testing pairs and compared the resulting sequences against the corresponding Javadoc sequences, called *references*.

To measure accuracy, we computed precision and recall of our generated documentation sequences. We computed the longest common subsequence (LCS) of a generated sequence and its reference sequence. Precision and recall values are computed as: $Precision = \frac{|LCS|}{|Result|}$, $Recall = \frac{|LCS|}{|Reference|}$.

The higher *Recall*, the higher the coverage of the generated sequences. *Recall*=1 means that the generated sequences cover all the words in the reference sequences in the right order. The higher *Precision*, the more correct the generated sequences. *Precision*=1 means that all words in the generated sequences are correct in the right order. We also measured BLEU scores. This evaluation criterion measures translation accuracy for *all possible phrases*. The higher it is from 0–1, the more likely all phrases are correctly translated. We also manually compared the semantics of the generated documentation with the references and classified the results into four categories: (1) the ones that are textually **matched** with the references; (2) the ones that are not exactly matched but have the **same** semantics; (3) the ones that are very **close** with the reference and might need slight modification to make it correct; and (4) the ones that have totally different semantics or are just **incorrect**.

The average precision, recall and BLEU score of this experiment is shown in Fig. 4. We achieve the high precision of 76%, high recall of 80% and high BLEU score of 56%. The lengths of the generated documentation sequences are almost the same as those of the references. In terms of semantic accuracy, 54 out of 100 generated documentations are correct, 31 of which are exactly matched with the expected. In 23 cases, the generated documentation contains phrases having the same semantics as in references even though they are not lexically matched. For example, we generated the phrase **less than 0** while the one in the reference is **negative** as shown below.

```
Generated: if horizon be less than 0 throw IllegalArgumentException
Reference: if horizon be negative throw IllegalArgumentException
```

There are 22 generated sentences that are semantically close to the references. For those cases, deleting or adding a few words (as shown below), or renaming an identifier to match with an argument name of the API would make them correct.

```
Generated: if key or value be null throw NullPointerException
Reference: if value be null throw NullPointerException
```

Analysis. We first study how certain aspects of API documentation affect the inference accuracy. We looked at two aspects: components of APIs involved in the conditions and the complexity of the conditions. The components of APIs include global states, receiver state and parameter states. They are conditions related to global variables, receiver object, and parameters, respectively. For parameters, we further divided into three sub-categories: para-type for conditions checking on the types of parameters, para-value for conditions checking on parameters against literal values, and para-method for conditions of calling predicate methods on parameters. For complexity, we considered a documentation simple if its conditions do not contain any conjunction or disjunction, and complex otherwise. Fig. 5 shows the BLEU scores for different categories. As seen, the inferring conditions on object types achieves highest accuracy and that on predicate methods of parameters has the lowest. The inference accuracy for simple documentation sentences is slightly better than that for complex ones.

To study further the knowledge that an SMT model learns, we investigated the mapping table produced as the by-product after the SMT model was trained. We focused on the mappings

|Gen|	|Ref|	Pre.	Rec.	BLEU	Matched	Same	Close	Incorrect
19	15	74%	86%	66%	23%	4%	38%	35%

Fig. 6: Documentation-to-Code Inference Result.

of operators and methods' argument names that are used in the conditions of exceptional behaviors. First, we examined 9 operators, which include conditional, unary, equality, and relational operators in Java language. We manually checked the corresponding mapping phrases for those operators in the phrase table produced by the tool. If a phrase is a correct English description of a given operator, we mark the rank of that correct one in the resulting ranked list. The SMT model can produce the correct descriptions of 8 out of 9 operators at the first result, which gives the top-1 accuracy of 89%. All correct results of these operators are found in top-4 mappings.

Second, we also randomly examined 100 identifiers that were used as the argument names of API method calls and were used in the conditions leading to exceptions. The SMT model achieves 76% top-1 accuracy, meaning that in 3 out of 4 cases, the model maps the identifiers in the code to the correct names in Javadoc at the top rank. It maps almost all the names correctly at top-3 and maps them all correctly at top-7.

C. Experiment 2. inferring implementation from documentation

In this experiment, we evaluated the generation capability in the other direction, i.e., from documentation to implementation. This is also a useful application of our approach because the resulting implementations will provide developers the idea on possible implementations that satisfy the given text.

We trained a SMT model in the direction from documentation to code. Then, we used the Javadoc sequences of the 100 test pairs as the inputs of the trained model to generate implementation sequences. The corresponding code sequences of the 100 pairs are used as references. We used the same metrics as in the previous experiment (precision, recall, and BLEU score). We also manually checked the semantics of the generated code.

The result of this experiment is shown in Fig. 6. Our result showed that the model can achieve a high quality of 74% precision, 86% precision and 66% BLEU score. In terms of semantic accuracy, 27 out of 100 generated code fragments are correct, 23 of which are exactly matched with the references and 4 others have the same semantics as expected. Such an example is shown below where the SMT generated the sequence !(size > 0) while the one in the reference is size <= 0.

```
Generated: if ( ! ( size > 0 ) ) throw new IllegalArgumentException ( ) ;
Reference: if ( size <= 0 ) throw new IllegalArgumentException ( ) ;
```

38 generated code fragments have semantics close to the references. All of them have syntax errors. Deleting or adding a punctuation or a few tokens, or renaming an identifier to match with an API's argument name would make them correct. In the example below, one needs to rename index to fromIndex and replace the redundant tokens at the end with a semi-colon.

```
Gen: if ( index < 0 ) throw new IndexOutOfBoundsException ( ) != null ) if (
Ref: if ( fromIndex < 0 ) throw new IndexOutOfBoundsException ( ) ;
```

Component					Complexity	
Global	Receiver	Para-type	Para-value	Para-method	Simple	Complex
71%	57%	56%	68%	55%	70%	56%

Fig. 7: Documentation-to-Code Inference Result Analysis.

1) Analysis: Fig. 7 shows the accuracy analysis for inference in this direction. Different from inferring from code to documentation, in this case, the highest accuracy is achieved for global conditions. The accuracy for para-method is still the lowest but is not significantly different from the accuracy for receiver and para-type. For complexity, the accuracy for simple conditions is much better than that for complex ones.

IV. Conclusion

In conclusion, our preliminary results show that the direction of statistical learning for inference between documentation and implementations is very promising. While our currently used phrase-based SMT achieves promising results, more customization on the mapping and language models are needed toward more formal specifications and source code. Language models specific to formal specifications are desired. A combination of a model for formal languages and a statistical language model is a good direction. Regarding mapping, a structure-based mapping model that is tailored toward structures in source code such as tree-to-tree mappings could improve inference accuracy.

V. Acknowledgments

This work was supported in part by the US National Science Foundation (NSF) grants CCF-1518897, CNS-1513263, CCF-1723215, CCF-1723432, and TWC-1723198.

References

[1] P. F. Brown, V. J. D. Pietra, S. A. D. Pietra, and R. L. Mercer, "The mathematics of statistical machine translation: parameter estimation," *Comput. Linguist.*, vol. 19, no. 2, pp. 263–311, Jun. 1993.

[2] R. Buse and W. Weimer. "Automatically documenting program changes". In ASE'10, pages 33–42. ACM, 2010.

[3] D. Cer, M. Galley, D. Jurafsky, and C. D. Manning, "Phrasal: A statistical machine translation toolkit for exploring new model features," in *Proceedings of the NAACL HLT 2010 Demonstration*, pp 9–12.

[4] B. Dagenais and M. P. Robillard. Creating and evolving developer documentation: Understanding the decisions of open-source contributors. In FSE'10, pages 127–136. ACM, 2010.

[5] P. Koehn, *Statistical Machine Translation*, 1st ed. New York, NY, USA: Cambridge University Press, 2010.

[6] T. Lei, F. Long, R. Barzilay, and M. Rinard. "From Natural Language Specifications to Program Input Parsers", In ACL'13, pages 1294–1303.

[7] T.T. Nguyen, H. A. Nguyen, N. Pham, J. Al-Kofahi, and T. N. Nguyen. "Graph-based mining of multiple object usage patterns". In ESEC/FSE'09, pages 383-392. ACM, 2009.

[8] H. A. Nguyen, R. Dyer, T. Nguyen, and H. Rajan, "Mining preconditions of APIs in large-scale code corpus,". In FSE'14, pages 166-177. ACM.

[9] S. Subramanian, L. Inozemtseva, and R. Holmes."Live API documentation". In ICSE'14, pages 643–652. ACM, 2014.

[10] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. "Towards automatically generating summary comments for Java methods". In ASE'10, pages 43-52. ACM, 2010.

[11] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," in ICST'12, pages 260–269. IEEE CS, 2012.

[12] 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 ICSE'16, pages 380-391. ACM, 2016.