CSE 503 Project Proposal

Qifan Lu (lqf96@uw.edu) Nikita Haduong (qu@uw.edu) Samia Ibtasam (samiai@uw.edu)

Motivation

The JDoctor paper [1] proposes generating program specifications from semi-structured JavaDoc comments. Program specifications express intended program behaviors and are useful in determining if the program is implemented correctly or not. While rarely practiced, semi-structured documentation with source code is supposed to convey the program's intention. JDoctor reads documentation accompanying Java classes and functions and constructs executable program specifications in the form of boolean Java expressions.

JavaDoc comments are used to annotate code with informal specification and usually include a description of member function followed by a series of directives. These directives or tags informally define program behavior and can be used to derive program specifications. For example, the '@param' directive describes method parameters and conditions that callers must conform to (called pre-conditions), the *'@return'* directive describes the return value of the function as well as its characteristics (called normal post-conditions); and the *'@throws'* directive describes what and when an exception will be thrown by the function (called exceptional post-conditions).

JDoctor translates JavaDoc comments into program specifications with four steps: text normalization, proposition identification, proposition translation, and specification creation. During the text normalization step, JDoctor preprocesses JavaDoc directives into grammatical English sentences to accommodate current NLP parsers. Then, it performs POS tagging and identifies the subject and predicate of conditions in the proposition identification step, based on the observation that nouns tend to correspond to variables or expressions while predicates tend to correspond to function calls. Finally, JDoctor makes use of pattern, lexical and semantic matching to map natural language utterances to corresponding expressions and identifier names and generate program specifications.

The translation steps that JDoctor performs are effectively semantic parsing, with a series of carefully-designed steps and manually-determined rules (such as all the text normalization rules and well-known pattern matching (e.g. "is positive" to `>0`)). ***The problem of this approach*** *is that manually-determined rules may fail if the writing style of JavaDoc changes (e.g. an author always use "greater than zero" instead of "is positive" for `>0`), or if the JavaDoc is written using a different language (e.g. documentation is written in Chinese). Furthermore, JDoctor is based on manually-designed steps, which may not generalize well to other programming languages (such as JavaScript's JSDoc or Python's DocString).*

The semantic matching process in the translation step uses word embeddings to match variable names, which inspires us about the possibility of applying deep learning to semantic parsing in order to generate program specifications. By using deep learning, we hope to discover more generalizable or automatically-discovered translation steps. As JDoctor is designed specifically for analyzing JavaDoc, with many handcrafted procedures, it is able to achieve impressive results on JavaDoc specifically at the cost of being expensive to create (the time and human labor of creating many rules and exceptions).

Finding a generalizable system would provide developers in other languages support for writing more secure code without the cost of creating language-specific versions of JDoctor.

Goal

The project’s goal is to develop a neural network-based translator that converts JavaDoc directives (or other program documentation) into program specifications with minimum manual processing steps. We will use library source code that carries program specifications (such as Java Modeling Language (JML), Contracts for Java (C4J) or JUnit test cases that utilize parameterized test cases and theories) as the training data for the neural network-based model. We may also consider alternative ways of obtaining program specifications, such as detecting program invariants using Daikon.

Proposed Design

The initial design will focus solely on translating JavaDoc directives into program specifications. If the approach is feasible and time permits, we will try to extend our approach to documentation for other programming languages. In either case, the input will be the program documentation directives that are related to method parameters, return values and exceptions. The output will be program specifications in the form of language-dependent boolean-typed expressions. In the case of JavaDoc, both the input and the output should be identical to [1].

Our neural network (NN) based model will have informal Javadoc code comments and the associated procedural specifications (generated by the JDoctor). The net will then learn the relationships between the code comments and the specifications. The NNs might also inform us about the possible weights (or influence) of a few tags or comments over others and over the program specification. This can be useful in understanding the effects or presence (or absence) of a tag or the weakness or strongness of a tag and its impact on the program specifications.

Evaluations

1. **Accuracy -** We can look at the original comments and the associated specifications produced by our system, to verify the accuracy of the produced specifications. Within accuracy, we will look at:
    a. **Recall and Precision:** False positives and true negatives based on the definition of the JDoctor paper.
    b. **Missing and Incomplete Specifications**: Specifications which are not covered by the mapping (missing) as well as incomplete (partial) specifications.
    c. **Invalid Specifications**: Specifications that fail to compile due to type mismatch (For statically-typed languages).
2. **Time and Effort -** We will look at the output of the NNs in the form of specifications and compare that with the output of JDoctors’ specification output. We will compare the improvements achieved in time and overall effort, both in the initial training (short term benefit) as well as the overall improvement with the addition of new inputs (long term benefit). **Adaptation to new Inputs:** Since NNs can quickly adapt to new inputs, we hope that the overall benefit will be higher.
3. **Output Quality Standard:** By program specifications, we mean either the logical forms or language-specific expressions. These will be for the machines because random test generation

tools such as Randoop will make use of it. Thus, we will ensure the format matching and standardization of the output as per the expected specifications guidelines.

4. **Adaptation to other languages (time permitting):** Once the specifications are trained for particular language-specific expressions, we can also try to evaluate its portability and easy adaptability to other languages by evaluating on a new training set for a different language/domain. This evaluation can tell us the scalability of this technique to other languages/docs/specifications.

Questions

- How to design the structure of NN so that it minimizes the possibility of producing illegal program specifications? Specifically, how to design NN-based models that are aware of the type systems of static-typed programming languages?
- Will the NN be more generalizable (to other programming/natural language/structures) than all the handcrafted rules of JDoctor and achieve just as high (or higher) results?
- Will the NN help us discover new rules and patterns?
- Are there some easy-to-craft Javadoc examples where the NN fails while JDoctor does well?

Contribution

The contributions of this project will be:

- To evaluate the suitability of using Neural Networks to translate language-specific descriptors to generate program specifications and its applicability to other languages.
- To measure the effectiveness (see evaluation section) of our approach and compare it with JDoctor, Toradocu (before JDoctor) [2] and @tComment [3].
- To demonstrate the consistency of our approach on libraries with comments of different writing styles or languages, and its ability to generalize to different programming languages.
- Our approach resolves a number of questions mentioned in the previous section.

References

[1] M. Balcer, W. Hasling, and T. Ostrand, “Automatic Generation of Test Scripts from Formal Test Specifications,” in *Proceedings of the ACM SIGSOFT ’89 Third Symposium on Software Testing, Analysis, and Verification*, New York, NY, USA, 1989, pp. 210–218.

[2] “Toradocu-Randoop (Manual) Evaluation. Feb-2018,” *GitLab*, Feb-2018. [Online]. Available: https://gitlab.cs.washington.edu/randoop/toradocu-manual-evaluation-feb-2018. [Accessed: 12-Apr-2019].

[3] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016). ACM, New York, NY, USA, 213-224. DOI: https://doi.org/10.1145/2931037.2931061

[4] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, “@tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies,” in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, Washington, DC, USA, 2012, pp. 260–269.