SpecTran: Neural Network Machine Translator of Javadoc Tags to Java Specifications

Nikita Haduong, Qifan Lu, Samia Ibtasam
{qu,lqf96,samiai}@uw.edu

ABSTRACT

Program specifications define intended program behavior and can range in formality and completeness from natural language comments to mathematical formulations. They are important for writing high quality, maintainable code but are difficult to write and hence often not written together with the program. Many tools that help programmers write specifications exist for specific programming languages; however, the tools often require substantial human effort to create and are language-specific.

With recent advances in neural machine translation systems in the natural language processing community, we investigate whether such systems can be easily adapted to this domain of assisting programmers with creating program specifications. We test our system, which we call SpecTran, on the Javadoc informal specification scheme that is shipped with the Java programming language. Our system performs well with in-domain data, achieving up to 86.8% accuracy, but it is not robust enough to perform comparably with out-of-domain data. We provide suggestions for improving the system further and hope our work will motivate others to continue this line of research.

CCS CONCEPTS

• **Software Engineering**;

KEYWORDS

datasets, Java, Recurrent Neural Networks, Javadoc, text tagging, Natural Language processing

ACM Reference Format:
Nikita Haduong, Qifan Lu, Samia Ibtasam. 2018. SpecTran: Neural Network Machine Translator of Javadoc Tags to Java Specifications. In . ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Program specifications define the intended behavior of the program and can vary in formality and completeness from natural language and test cases to formal mathematical expressions [3, 16]. Their primary usefulness lies in their ability to serve as oracles in determining if programs are implemented correctly or not during software testing. They can also be used to learn program invariants. Formal specifications are expressions in some formal language and at some level of abstraction of a collection of properties some system should satisfy [16]. Even though formal specifications are clear, unambiguous, and can be automatically evaluated, they are often difficult to write and debug [12]. Many programming languages instead provide informal specification documentation schemes that are often in the form of natural language and are intended for a human audience. However, though these informal specifications provide helpful information to programmers about the nature of a program, they cannot be used directly to verify the correctness and behavior of a program. This has motivated work in developing systems to help automatically generate formal program specifications from informal specifications written in natural language.

Automatic documentation to formal specification translation systems, such as JDoctor [3], Toradocu [1] and @tComment [28], have achieved high precision in generating formal and complete program specifications. However, these systems often require a fair amount of human effort in constructing manual translation rules and evaluating their efficacy. The requisite human effort slows down the adaptation of these technologies to libraries with different informal documentation styles or other programming languages.

Our work aims to decrease the human effort required to automatically generate formal program specifications from Javadoc comments, while achieving similar performance as previous tools. We introduce SpecTran, a proof-of-concept neural machine translation system for translating informal Javadoc specifications into formal specifications, which we refer to as boolean expressions [30]. We use Javadoc over another language and its informal specification format because a high-quality dataset with human-translated samples was created by [3]. Java is a relatively popular programming language that has been adopted extensively by companies and individuals, and Java applications rely heavily on libraries such that good documentation is crucial [12]. Thus, the success of SpecTran (and other such documentation assistance tools) that work well with Javadoc can have a high impact on the software engineering ecosystem.

SpecTran shows that adapting a standard natural language machine translation system to the program specifications domain is feasible. Though we test SpecTran only on one specific type of informal specification scheme, Javadoc comments, we believe the proof-of-concept can easily be extended to other informal specification schemes.

1.1 Javadoc

Javadoc, initially released in 1995 with the first version of Java, defines a standard format for documenting Java classes and generating API documentations in HTML format from Java source code [18]. *Javadoc comments* are multi-line comments surrounded by /** ... */ delimiters that are placed before class, field or method declarations [19]. They are made up of two parts: a description of

```
/** Splits this string around matches of a
     regular expression.
 * @param  regex  the delimiting regular
      expression
 * @return  the array of strings computed by
      splitting this string
 * @throws  PatternSyntaxException  if the
      regular expression's syntax is invalid */
public String[] split(String regex) {
  return this.split(regex, 0);
}
```

Figure 1: Javadoc comments with @param, @return and @throws tags for a procedure and procedure definition

the class, field or method and a number of tags prefixed with @ that describe method details like parameters, return values [8, 20] and exceptions [23], as shown in Figure 1.

The `@param` tag describes method parameters and conditions that callers must conform to (pre-conditions), the `@return` tag describes the return value of the function as well as its characteristics (normal post-conditions), and the `@throws` tag describes what and when an exception will be thrown by the function (exceptional post-conditions).

1.2 Contributions

We show that a neural machine translation system [2, 5] is able to translate Javadoc descriptions into boolean expressions, easily generating legal boolean expressions. Particularly, the system does not require special architectural designs–classical architectures used in natural language machine translation [2, 15] can adapt well to this task. While the performance of our system has room for improvement, our work is a first step in creating systems that can reduce human effort in building similar documentation-to-specification tools. Section 9 provides suggestions for improving the system in future work.

2 SPECTRAN

Our neural machine translation model is designed to take Javadoc descriptions as input and produce program specifications in the form of boolean expressions. We use the OpenNMT-py library [14], which is an open source Pytorch implementation of OpenNMT [13], and train on two datasets that are described in Section 3. Figure 2 describes the pipeline of our system. An example usage of the system is for a user to input a Javadoc description:

```
@throws java.lang.NullPointerException set is null
```

and receive as output a boolean expression: *set == null*

Embeddings are concatenated to form an input to the first hidden layer [17].

2.1 Architecture

We approach this task as a standard machine translation task (translating from a Javadoc description to a boolean expression (also known as a condition or specification)) and, as such, use a bi-LSTM seq2seq model with attention as proposed in [2, 29]. This architecture contains an encoder, a decoder, and an attention layer as shown in Figure 3. The encoder consists of a 500-dimension embedding layer, followed by two stacked bi-LSTM layers. Similarly, the decoder contains two 1000-dimension LSTM layers and a 500-dimension embedding layer. We use the concatenation-based dot-product attention described in [29]. To predict the next token in a sequence, we use a linear layer followed by a log softmax layer.

Some examples of the inputs (Javadoc tags) to SpecTran and the conditions (boolean expressions) as outputs are listed below. The first example shows a precondition and the corresponding output of our tool.

```
INPUT:
@param map the map to decorate. map is not null
OUTPUT: (map==null)==false

INPUT:
@throws java.lang.NullPointerException set is null
OUTPUT:  set==null

INPUT:
@throws java.lang.IllegalArgumentException n < 0 or
   ↪  k <= 0 or k > n
OUTPUT: n<0 || k<=0 || k>n
```

2.2 Embedding layers

The embedding layer in the model numericalize input tokens as a sequence of vectors. Embeddings map sparse, high-dimensional inputs (such as one-hot encoded words) into lower-dimensional vectors while still retaining semantic information in the embedding space [11]. Embeddings are concatenated to form an input to the first hidden layer [17]. We experiment with three different representations: word embeddings, character embeddings and byte-pair encoding (BPE). The weights of all embedding layers are learned on-the-fly during the training process.

Word embeddings represent every unique word in the training data as an N-dimensional vector (where $N = 500$ in our case). If the training data contains only the words "the fox jumped and the cat ate", then the resulting embedding will be a 6×500 matrix.

Character embeddings are similar to word embeddings, but every unique character receives its own N-dimensional vector. For example, if the input training data contains only characters "a" to "z", then the embedding will be a 26×500 matrix.

Byte-pair encoding [24] compresses text by replacing the most common pair of consecutive bytes with a byte that never occurs in that text. This substitution is performed multiple times until a certain number of substitutions are completed. After byte-pair encoding, frequent arbitrary-length character sequences are usually mapped to a single new byte, reducing the overall length of the original text. Compressed byte sequence is then embedded in a similar manner to character embeddings. In our experiments, we do 500 substitutions.

The intuition behind character embeddings and BPE is that they should be robust enough to handle unknown vocabulary in the testing data.

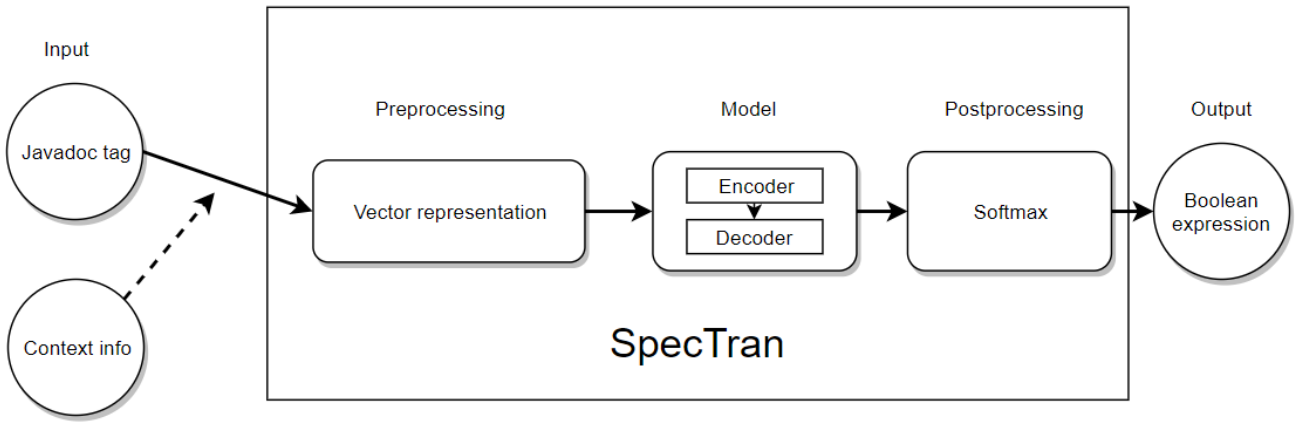

Figure 2: SpecTran takes as input a Javadoc comment with optional contextual information and outputs a boolean expression. An example of model usage is: (Input) *@throws java.lang.NullPointerException set is null.* and (Output) *set == null.*

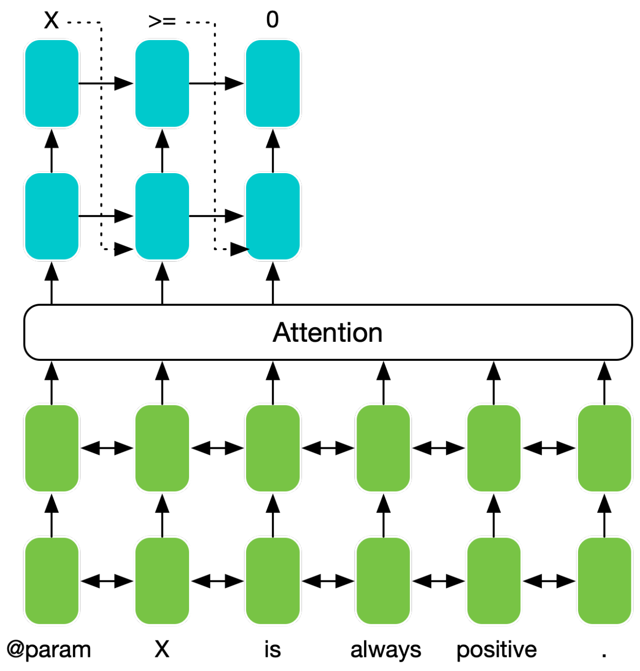

Figure 3: An illustration of the seq2seq model with attention used by SpecTran to translate a sample Javadoc comment (`@param X is always positive`) to a specification (`X >= 0`).

3 DATA

We train and evaluate SpecTran models with different input data and settings.

Data Sets: We used two datasets containing Javadoc descriptions (source input) extracted from Java libraries and their corresponding boolean expression (target output) translations. The target outputs in the **Gold Label Dataset** were created by humans and contain ground-truth program specifications while the **JDoctor Dataset** was generated by running the JDoctor tool [3] on a number of Java libraries. Normal post-conditions (**@returns** tags) in the **Gold Label Dataset** required post-processing to create a target output format suitable for SpecTran because the normal post-conditions in **Gold Label Dataset** could contain multiple possible outputs, or `(guard, property)` pairs. The `guard` is a boolean Java expression served as a predicate and `property` is a boolean Java expression describing the constraint on the return value under corresponding `guard`. Because our model output is the single most probable sequence of tokens, we merge the multiple outputs into one large pseudo-expression (Figure 4).

Data Split: We used two methods to split datasets into training, validation and testing sets: one was a combination of all the libraries with a random 80/10/10 split (**ALL-LIB**), and the other had some libraries (2 libraries for Gold Label Dataset and 6 libraries for JDoctor dataset) withheld from the training set (**TRAIN-TEST-SPLIT**). The withheld libraries were then split randomly with a 50/50 testing/validation split.

The model input had the following variations:

(1) **Textual embedding representation.** See section 2.2 for descriptions of the three representation schemes used: word embedding, character embedding and byte-pair encoding (BPE). Outputs from the model use the same representation as the input. For word embeddings, we tokenize source input with white-space and target output with an AntLR-based Java parser[1].
(2) **Contextual information.** Source input could consist of solely the Javadoc descriptions, or it could also have supplementary contextual information from the original method containing the Javadoc description. Supplementary contextual information was included in the source input by concatenating the following white-space delimited sequence to the end of the original Javadoc description: class name, method name, receiver name, return name, and parameter names. Class name is the name of the class that contains the method; method name is the name of the method; receiver name is the name of the object represented by **this** keyword

[1]https://github.com/antlr/grammars-v4/blob/master/java9/Java9.g4

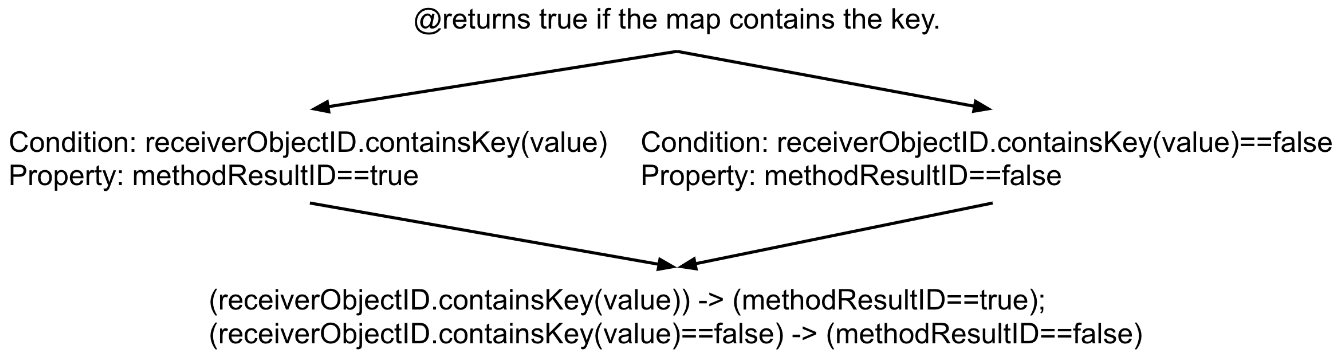
@returns true if the map contains the key.
Condition: receiverObjectID.containsKey(value)
Property: methodResultID==true
Condition: receiverObjectID.containsKey(value)==false
Property: methodResultID==false
(receiverObjectID.containsKey(value)) -> (methodResultID==true);
(receiverObjectID.containsKey(value)==false) -> (methodResultID==false)

Figure 4: Normal post-conditions handling for Gold Label Data with multiple `(guard, property)` pairs merged into one pseudo-expression. This step was not needed for the JDoctor Dataset because JDoctor generates a single boolean expression for normal post-conditions (usually using the if-then-else ternary operator).

in the method; return name is the variable name for the return value of the method; parameter names are the one or more input parameters to the method. See Figure 5 for an illustration of the input format.

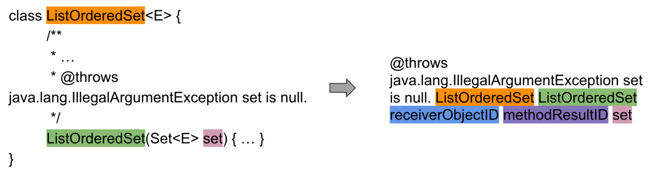
class ListOrderedSet<E> {
/**
* ...
* @throws
java.lang.IllegalArgumentException set is null.
*/
ListOrderedSet(Set<E> set) { ... }
}
@throws
java.lang.IllegalArgumentException set
is null. ListOrderedSet ListOrderedSet
receiverObjectID methodResultID set

Figure 5: Example of appending context words to the description of an @throws tag for a constructor. Note that receiver and return names are added in gold label data or by JDoctor and are not present in the original source code.

4 METRICS

We evaluate model performance by calculating the accuracy of its predicted output. Accuracy is calculated by counting the number of predicted translations that are an exact match with the target output. Additionally, as a translation task may result in output that is semantically different but syntactically the same, we perform two error analyses: identifier name mismatch (**ID-MISMATCH**) and incompleteness (**INC**). Error analyses are calculated as a percentage of the total number of errors.

(1) **ID-MISMATCH** occurs when the only difference between the predicted output and actual output is an identifier name at corresponding locations in the two outputs. Other characteristics of the predicted output, such as the length of the sequence, the order, type and content of Java tokens remain the same. The following example shows an **ID-MISMATCH** with *list* and *level* on the left-hand-side of the embedded expression.

```
Target Output: (list == null ) == false
Model Output: (level == null ) == false
```

(2) **INC** occurs when the predicted output is partially correct but is incomplete. For example:

```
GOLD: a == null || b == null || c == null
PRED: a == null || b == null
```

Higher values for the two error analyses metrics are better because a higher **ID-MISMATCH** error indicates syntactically correct but semantically different predictions, and **INC** indicates that the model came close to the correct prediction in some capacity as opposed to being completely incorrect.

5 EXPERIMENTS

We split our experiment setups by dataset and vary the input settings as described in Section 3.

5.1 Experiment 1: Gold Label Dataset

This experiment setup uses the **Gold Label Dataset**, which contained 6236 *(Javadoc description, specification) pairs* from seven Java libraries from the JDoctor evaluation repository [3]. After splitting the data into training, validation and testing sets, **ALL-LIB** contained 4988 training samples and 624 test/validation samples, and **TRAIN-TEST-SPLIT** contained 4631 training samples and 802 test/validation samples.

5.2 Experiment 2: JDoctor Dataset

This experiment setup uses the **Jdoctor Dataset** where target output was generated by using the JDoctor tool. Source input and target output are in the same format as the **Gold Label Dataset**. We first run JDoctor on six Java libraries from JDoctor evaluation repository (excluding the Java standard library) and nine additional Java libraries[2]. JDoctor can produce empty output if it does not find a condition in the Javadoc tag, so we discard such samples from the JDoctor dataset. After cleaning JDoctor output, the final dataset contained 8037 samples. Splitting the data resulted in the **ALL-LIB**

[2]Additional libraries are: assertj, commons-codec, commons-io, commons-lang3, commons-text, commons-vfs2, log4j, netty, RxJava.

	Context Info	All Datasets			Training-Test Dataset		
		Accuracy	**ID-MISMATCH**	**INC**	**Accuracy**	**ID-MISMATCH**	**INC**
Byte Pair Encoding	With Context	68.9	3.6	2.4	2.4	1.4	0
	Without Context	74.1	3.1	1.8	2.4	2.1	0
Word Embedding	With Context	**79.0**	53.4	7.4	8.1	35.0	0
	Without Context	77.5	*57.1*	7.6	**12.7**	*47.5*	2.4
Character Embedding	With Context	–	–	–	–	–	–
	Without Context	10.9	0	2.7	8.5	0	2.0

Table 1: Gold Label Dataset. ID-MISMATCH and INC error analysis results are calculated as a percentage of the total number of errors. See Section 4 for additional descriptions of the error analyses.

	Context Info	All Datasets			Training-Test Dataset		
		Accuracy	**ID-MISMATCH**	**INC**	**Accuracy**	**ID-MISMATCH**	**INC**
Byte Pair Encoding	With Context	62.2	37.5	4.3	**15.4**	24.3	0.6
	Without Context	77.4	61.5	0.5	1.7	38.0	0.8
Word Embedding	With Context	**86.8**	52.8	5.7	1.6	24.4	0
	Without Context	76.2	*70.2*	3.1	4.1	*46.0*	1.5
Character Embedding	With Context	–	–	–	–	–	–
	Without Context	0.6	0	1.0	0	19.5	0

Table 2: JDoctor Dataset. ID-MISMATCH and INC error analysis results are calculated as a percentage of the total number of errors. See Section 4 for additional descriptions of the error analyses.

set containing 6429 training samples with 804 test/validation samples, and **TRAIN-TEST-SPLIT** contained 4944 training samples with 1546 test/validation samples.

6 RESULTS

Table 1 summarizes model results from experiment 1, and Table 2 summarizes results from experiment 2. In both experiments, **ALL-LIB** performs significantly better than **TRAIN-TEST-SPLIT**, regardless of the encoding used and whether context information is included or not. Character embedding models performed extremely poor, sometimes achieving zero accuracy, compared to the other two input representation schemes, so we disregard character embedding results[3]. Word embedding models performed best in all but **TRAIN-TEST-SPLIT** in Experiment 2, where BPE greatly outperformed word embeddings. Adding contextual information to the source input only appears to be helpful in the **ALL-LIB** setting.

We show a sample of predicted failed outputs from the two models that perform best in experiment 1 and experiment 2 in Table 3. The first and third examples are complex failed outputs that are neither an **ID-MISMATCH** error nor an**INC** error. These two examples show a model preference for generating outputs of the form `[Identifier] [Binary Operator] [Constant]`. The second and fourth examples are interesting to compare because the output in the second example is incomplete on the left side of the expression (order does not necessarily matter to the model) while the fourth example shows the model adding an additional expression and disjunction.

[3]Character embedding models with contextual information could not be run, as the model calculated the input data to contain only four distinct vocabulary characters.

7 DISCUSSION

High performance in the **ALL-LIB** setting compared with the **TRAIN-TEST-SPLIT** setting indicate that the model overfits to the data. The vast difference in performance is a surprising result because manual inspection of the data shows that similar samples exist in the testing/validation data for both settings. It is possible that due to duplicate samples within the training data, the degree to which the model overfits is more extreme than expected. With too many duplicate training samples, the model learns to translate only very specific types of descriptions, and should it encounter a description that is the same except for a different parameter name, it will completely fail in its prediction. If both **ALL-LIB** and **TRAIN-TEST-SPLIT** contained the same fraction of duplicate training samples, then the large difference in performance could be partially explained. Unfortunately, we were unable to perform this analysis, discovering the duplicate samples too late.

With further inspection of the data, we find many of the target output samples to be needlessly complicated. For example, the target output might be *(put == null) == false*, and a simpler equivalent would be *put != null*. If the training data contained many variations of possible target output for the same source input, the model might become more robust and perform better on out-of-domain data. Additionally, these variations would be able to augment the quantity of training data.

SpecTran learns to translate descriptions into legal (but not necessarily accurate) boolean expressions but finds it difficult to use the correct parameter names (indicated by **ID-MISMATCH** results). This is in contrast with our expectations, as we expected the reverse to occur. We hypothesized that parameter names would be a simple matter for the model to copy from input to prediction, and the boolean expression/operator would be difficult because of the

	Source input	Gold target output	Predicted output
Gold Label Dataset	@returns the not predicate. PredicateUtils notPredicate receiverObjectID methodResultID predicate	(true) -> (methodResultID.equals(predicate))	(true) -> (methodResultID >= 0)
	@param set the set to decorate. set is empty and not null. ListOrderedSet listOrderedSet receiverObjectID methodResultID set list	receiverObjectID.isEmpty( ) && (set == null)== false	(set == null) == false
JDoctor Dataset	@throws java.lang.IllegalArgumentException exceptions is empty. CompositeException CompositeException receiverObjectID methodResultID exceptions	exceptions != null && exceptions.length == 0	expectedSize <0
	@returns the source or its wrapper Completable. Completable wrap receiverObjectID methodResultID source	true ? methodResultID .equals(source)	true ? (methodResultID.equals (source) || methodResultID == null)

Table 3: Examples of bad predictions. All examples taken from the two best performing models: ALL-LIB+Word Embeddings+Context information.

possible many-to-one translation mapping (e.g., "is greater than" in a description needs to map to ">").

8 RELATED WORK

We review the literature of work in program specification generation, program documentation generation, natural language machine translation, and semantic parsing.

8.1 Program Specification Generation

Our work is inspired by systems that can generate program specifications from program comments (natural language).

A tool for automatically deriving formal program specifications from interfaces of Java classes was developed by [12]. The tool probes Java classes by invoking them on dynamically generated tests and captures the information observed during their execution as algebraic axioms. More recently, Blasi et al. proposed JDoctor [3], an approach that combines pattern, lexical, and semantic matching to translate Javadoc comments into executable procedure specifications written as Java boolean expressions. JDoctor translates Javadoc tags into program specifications with four steps: text normalization, proposition identification, proposition translation and specification creation. Toradocu [10] is the foundation of JDoctor and it specifically focuses on generating test oracles for exceptional behaviors by translating `@throws` tags into exceptional post-conditions. Toradocu and JDoctor make use of similar techniques, like subject and predicate identification and pattern and lexical matching, to extract subject and predicates and perform translation. However Toradocu does not consider semantic information of natural language in Javadoc tags.

Like Toradocu, earlier works usually focus on specific program characteristics, rather than producing general program specifications. @tComment [28] uses pattern-matching to determine three kinds of precondition properties related to nullness of parameters. ALICS [21] infers formal specifications from natural language text of API documents. Though ALICS performs automatic parts-of-speech (POS) tagging on Javadoc tags, it then pattern-matches these tags against a small set of hard-coded nouns and jargon, limiting its generalizability. Phan et. al [22] used n-gram language models for behavioral exceptions (a specific type of precondition) in Javadoc documentation of the APIs in JDK and showed that using statistical learning for inference between implementations and documentation only works for exceptional comments and does not consider semantic knowledge. Zhou et al. [34] combine Javadoc tag translation with condition verification using SMT solvers. However their system only works for certain types of preconditions. Finally, iComment [26] uses part of speech tagging, phrase & clause parsing, and semantic role labeling as features in machine learning, statistical and program analysis techniques to extract implicit program rules from comments. aComment [27] which built on iComment [26] detected code-comment inconsistencies and concurrency bugs. These two tools still focus on special use cases, although they process unstructred data, unlike Javadoc tags.

A common theme among these previously developed tools is the requirement of human effort in creating special rules and pipelines for specific languages. Our work aims to reduce the amount of human effort required to develop such tools.

8.2 Documentation Generation

Many tools, languages and frameworks have been introduced as documentation standards. Programmers following best practices in documenting their code will often use such frameworks, and our system makes use of these informal documentation practices to produce formal specifications.

Javadoc [18], a documentation generator for Java, generates documentation from programming objects that follow a framework structure which includes data attributes, attribute manipulation methods etc. The programmer must provide the objects in the correct structure in order for Javadoc to perform correctly. JML (Java Modeling Language), started by Gary Leavens et al. at Iowa State University, was also introduced as a means for documenting Java programs. It can be used to specify the detailed design of Java classes and interfaces by adding annotations to Java source files [4].

In addition to code written by developers, third-party libraries form an important part of programs. However, library source code can be unavailable, a mix of other languages, and be highly complex, optimized implementations. Thus, [32] built a tool that took Javadoc as input, then used the Javadoc parser and various text processing tools to generate models for that could later be used in program analysis.

Besides JML, Eiffel was also introduced as another contract documentation language to enable programmers to include contracts (specification elements embedded in executable code) in their code [9]. Microsoft's SAL is a source code annotation language that makes explicit descriptions of how a function uses its parameters, its assumptions during execution, and its and guarantees upon exit [6]. These annotations are then used for automatic static analysis tools.

8.3 Natural Language Processing and Machine Translation

The informal documentation we use for our system is written in natural language, and because we want to convert this natural language into formal specifications that are also written in a text-format, we approach our task as a machine translation problem. Current state-of-the-art machine translation models use deep neural networks. These networks are trained on large quantities of translation pairs (e.g. a source language sentence in English and its translation in the target language). They are composed of an encoder that reads and encodes a source sentence into a fixed-length vector and a decoder that outputs a translation from the encoded vector [5]. However, this method is limited by the length of sentences it can handle. Thus, [2] proposed encoding the input sentence into a sequence of vectors and choosing a subset of these vectors adaptively while decoding the translation to achieve improved translation performance. With respect to neural network architecture, [25] showed that neural machine translation based on RNN (Recurrent Neural Networks) with long short-term memory (LSTM) performed well on the conventional English to French phrase-based machine translation task.

8.4 Semantic Parsing

The semantic parsing task aims to translate natural language into a formal meaning representation such as logical forms or structured queries ([15], [7]). In one semantic parsing work, Zhong et al. [33] use deep neural networks to translate nature language questions into corresponding SQL queries. Their model outperformed the previous state-of-the-art neural semantic parsing model by Dong and Lapata [7] that used an attention-enhanced encoder-decoder model trained on natural language descriptions paired with meaning representations. It encoded sentences and decoded logical forms using RNN-LSTM units.

One shortcoming of semantic parsing, specifically in the SQL synthesis space, is that the standard approach of synthesizing SQL queries from natural language uses sequence-to-sequence-style modeling and requires serialization of SQL queries. However, there can be multiple equivalent serializations of the same SQL query–this is known as the *order matters* problem. SQLNet [31] solved this problem by avoiding the sequence-to-sequence structure when the order does not matter. They utilized a dependency graph such that one prediction can be done by taking into consideration only the previous predictions that it depends upon.

Though state-of-the-art, these approaches treated the logical form as an unstructured sequence and thus ignored type constraints on well-formed programs and did not address entity linking. Thus, [15] design a semantic parsing model that guarantees well-typed logical forms and incorporates an entity linking embedding module in its encoder. This module allows the model to identify question spans that should be linked to entities. Their model architecture is based on an encoder-decoder neural network trained on a question-answer task. In our work, we choose to use a similar encoder-decoder bi-LSTM network with attention as discussed in our review of the literature.

9 CONCLUSION AND FUTURE WORK

We present a neural network model for automatically translating Javadoc descriptions into boolean expressions. While model performance is high with in-domain data and very poor with out-of-domain data, our work shows the feasibility of applying standard neural machine translation techniques to this new domain. Creating better models that can reduce the amount of human effort required to develop tools for generating informal specifications to formal specifications would greatly benefit the quality of software and the software development ecosystem. We believe that this line of research is valuable and hope others will be motivated to continue it.

Results of this work show the difficulty of developing a model to be robust enough for out-of-domain data despite out-of-domain data appearing fairly similar to the training data. Additionally, while context information decreases the frequency of **ID-MISMATCH** and **INC** errors, the overall accuracy also falls. Future work should explore incorporating contextual information in a different manner. These alternative ways include using a special prefix attached to contextual information, a special delimiter that is unseen in training data, and incorporating the information as a separate layer deeper in the network and allowing the network to learn when the contextual information is helpful. While this work explores varying the input to the network, it uses a model architecture that is not necessarily suitable for a task with limited training data. Changing the architecture or fine-tuning a pretrained model to our specific task is also an avenue for future investigation. Finally, it is imperative to gather additional data or to augment the dataset in an automatic manner.

REFERENCES

[1] 2018. Toradocu-Randoop (Manual) Evaluation. Feb-2018. https://gitlab.cs.washington.edu/randoop/toradocu-manual-evaluation-feb-2018/tree/commons-math

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv:1409.0473 [cs, stat]* (Sept. 2014). http://arxiv.org/abs/1409.0473 arXiv: 1409.0473.

[3] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro PezzÃĺ, and Sergio Delgado Castellanos. 2018. Translating Code Comments to Procedure Specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 242–253. https://doi.org/10.1145/3213846.3213872 event-place: Amsterdam, Netherlands.

[4] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. 2005. An overview of JML

tools and applications. *International Journal on Software Tools for Technology Transfer* 7, 3 (June 2005), 212–232. https://doi.org/10.1007/s10009-004-0167-4

[5] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN EncoderâĂŞDecoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1724–1734. https://doi.org/10.3115/v1/D14-1179

[6] corob msft. [n. d.]. SAL Annotations. https://docs.microsoft.com/en-us/cpp/c-runtime-library/sal-annotations

[7] Li Dong and Mirella Lapata. 2016. Language to Logical Form with Neural Attention. *arXiv:1601.01280 [cs]* (Jan. 2016). http://arxiv.org/abs/1601.01280 arXiv: 1601.01280.

[8] DrJava.org. 2013. Documentation with Javadoc. http://www.drjava.org/docs/user/ch10.html

[9] Eiffel Studio. 2012. Eiffel Software. https://www.eiffel.com/

[10] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro PezzÃĺ. 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. https://doi.org/10.1145/2931037.2931061 event-place: SaarbrÃijcken, Germany.

[11] Google. [n. d.]. Embeddings: Translating to a Lower-Dimensional Space | Machine Learning Crash Course. https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space

[12] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. 2004. *Discovering Algebraic Specifications for Java Classes.*

[13] http://opennmt.net/. 2019. Open Source Neural Machine Translation in PyTorch. Contribute to OpenNMT/OpenNMT-py development by creating an account on GitHub. https://github.com/OpenNMT/OpenNMT-py original-date: 2017-02-22T19:01:50Z.

[14] http://opennmt.net/. 2019. Open Source Neural Machine Translation in Torch. Contribute to OpenNMT/OpenNMT development by creating an account on GitHub. https://github.com/OpenNMT/OpenNMT original-date: 2016-10-24T16:09:20Z.

[15] Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. 2017. Neural Semantic Parsing with Type Constraints for Semi-Structured Tables. 1516–1526. https://doi.org/10.18653/v1/D17-1160

[16] Axel van Lamsweerde. 2000. Formal Specification: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*. ACM, New York, NY, USA, 147–159. https://doi.org/10.1145/336512.336546 event-place: Limerick, Ireland.

[17] Thang Luong, Michael Kayser, and Christopher D. Manning. 2015. Deep Neural Language Models for Machine Translation. In *Proceedings of the Nineteenth Conference on Computational Natural Language Learning*. Association for Computational Linguistics, Beijing, China, 305–309. https://doi.org/10.18653/v1/K15-1031

[18] Oracle. 1993. javadoc-The Java API Documentation Generator. https://docs.oracle.com/javase/7/docs/technotes/tools/solaris/javadoc.html#processingofsourcefiles

[19] Oracle. 2012. How to Write Doc Comments for the Javadoc Tool. https://www.oracle.com/technetwork/articles/javase/index-137868.html

[20] Oracle. 2013. How to Write Doc Comments for the Javadoc Tool. https://www.oracle.com/technetwork/java/javase/documentation/writingdoccomments-137785.html#principles

[21] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, Zurich, 815–825. https://doi.org/10.1109/ICSE.2012.6227137

[22] Hung Phan, Hoan Anh Nguyen, Tien N. Nguyen, and Hridesh Rajan. 2017. Statistical Learning for Inference Between Implementations and Documentation. In *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track (ICSE-NIER '17)*. IEEE Press, Piscataway, NJ, USA, 27–30. https://doi.org/10.1109/ICSE-NIER.2017.9 event-place: Buenos Aires, Argentina.

[23] Daniel Schreck, Valentin Dallmeier, and Thomas Zimmermann. 2007. How documentation evolves over time. In *Ninth international workshop on Principles of software evolution in conjunction with the 6th ESEC/FSE joint meeting - IWPSE '07*. ACM Press, Dubrovnik, Croatia, 4. https://doi.org/10.1145/1294948.1294952

[24] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural Machine Translation of Rare Words with Subword Units. *CoRR* abs/1508.07909 (2015). arXiv:1508.07909 http://arxiv.org/abs/1508.07909

[25] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'14)*. MIT Press, Cambridge, MA, USA, 3104–3112. http://dl.acm.org/citation.cfm?id=2969033.2969173 event-place: Montreal, Canada.

[26] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. [n. d.]. /* iComment: Bugs or Bad Comments? */. ([n. d.]), 14.

[27] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. ACM Press, Waikiki, Honolulu, HI, USA, 11. https://doi.org/10.1145/1985793.1985796

[28] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12)*. IEEE Computer Society, Washington, DC, USA, 260–269. https://doi.org/10.1109/ICST.2012.106

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *arXiv:1706.03762 [cs]* (June 2017). http://arxiv.org/abs/1706.03762 arXiv: 1706.03762.

[30] E. Weyuker, T. Goradia, and A. Singh. 1994. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering* 20, 5 (May 1994), 353–363. https://doi.org/10.1109/32.286420

[31] Xiaojun Xu, Chang Liu, and Dawn Song. 2017. SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning. *arXiv:1711.04436 [cs]* (Nov. 2017). http://arxiv.org/abs/1711.04436 arXiv: 1711.04436.

[32] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. 2016. Automatic model generation from documentation for Java API functions. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. ACM Press, Austin, Texas, 380–391. https://doi.org/10.1145/2884781.2884881

[33] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *arXiv:1709.00103 [cs]* (Aug. 2017). http://arxiv.org/abs/1709.00103 arXiv: 1709.00103.

[34] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs Documentation and Code to Detect Directive Defects. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 27–37. https://doi.org/10.1109/ICSE.2017.11 event-place: Buenos Aires, Argentina.