2015 30th IEEE/ACM International Conference on Automated Software Engineering

Pseudogen: A Tool to Automatically Generate Pseudo-code from Source Code

Hiroyuki Fudaba, Yusuke Oda, Koichi Akabe, Graham Neubig, Hideaki Hata,
Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, NARA 630-0192 JAPAN
Email: {fudaba.hiroyuki.ev6, oda.yusuke.on9, akabe.koichi.zx8, neubig, hata}@is.naist.jp

***Abstract*—Understanding the behavior of source code written in an unfamiliar programming language is difficult. One way to aid understanding of difficult code is to add corresponding pseudo-code, which describes in detail the workings of the code in a natural language such as English. In spite of its usefulness, most source code does not have corresponding pseudo-code because it is tedious to create. This paper demonstrates a tool `Pseudogen` that makes it possible to automatically generate pseudo-code from source code using statistical machine translation (SMT).[1] `Pseudogen` currently supports generation of English or Japanese pseudo-code from Python source code, and the SMT framework makes it easy for users to create new generators for their preferred source code/pseudo-code pairs.**

I. Introduction

While the ability to understand code is important to all programmers, it is often not trivial to understand source code. In an educational situation, pseudo-code, which expresses the behavior of algorithms in plain language, is used to help learners to understand algorithms. This is helpful because pseudo-code is written in natural language, which is independent from the syntax and semantics of any specific programming language. Fig. 1 is an example of Python source code with corresponding English or Japanese pseudo-code. If Python beginners try to read this source code, they may have difficulty understanding its behavior. However, if they were able to read the pseudo-code at the same time, the source code's behavior could be more easily understood.

The difficulty of this approach is that most source code does not have corresponding pseudo-code because adding pseudo-code is tedious work for programmers. One solution to this problem is to automatically generate pseudo-code from source code. In order to use automatically generated pseudo-code in the real world, it is required that pseudo-code should be accurate enough to describe the behavior of the original source code, and that the method to generate pseudo-code is efficient to avoid making users wait.

In this paper, we propose a tool `Pseudogen`, which uses statistical machine translation (SMT) to automatically generate pseudo-code from source code, according to the method of Oda et al. [1]. SMT is a paradigm for translating from one language to another based on statistical models [2], [3], which is generally used to translate between natural languages such as English and Japanese, but has also been used for software engineering applications such as code migration [4]. SMT has the advantage of not only producing accurate results, but also that its models are trained directly from parallel sets of data expressing the input and output, without the need for manual creation of transformation rules.

Currently, `Pseudogen` natively supports generation of English pseudo-code from Python code, and users can perform this translation through a downloadable tool or a publicly available web interface. In addition, the SMT approach also indicates that pseudo-code generators can be learned for new source code/pseudo-code pairs. To support this, `Pseudogen` includes a learning functionality, which users can use to create their own pseudo-code generators.

II. Method for Pseudo-code Generation

A. Statistical Machine Translation

SMT is an application of natural language processing (NLP) that translates between two languages, generally natural languages such as English and Japanese. SMT algorithms consist of a *training* step and *translation* step. In the training step, we extract "rules" that map between fragments of the input and output languages. In the translation step, we use these rules to translate into the output language, using a statistical model to choose the best translation [3], [5], [2]. In this paper, we use $\boldsymbol{s} = [s_1, s_2, \cdots, s_{|\boldsymbol{s}|}]$ to describe the "source sentence," an array of input tokens, and $\boldsymbol{t} = [t_1, t_2, \cdots, t_{|\boldsymbol{t}|}]$ to describe the "target sentence," an array of output tokens. The notation $|\cdot|$ represents the length of a sequence. In this paper, we are considering source code to pseudo-code translation, so $\boldsymbol{s}$ represents the tokens in the input source code statements and $\boldsymbol{t}$ represents the words of a pseudo-code sentence. For example, in the small Python to English example in Fig. 2, $\boldsymbol{s}$ is described as the sequence of Python tokens ["`if`", "`x`", "`%`", "`5`", "`==`", "`0`", "`:`"], and $\boldsymbol{t}$ is described as ["if", "x", "is", "divisible", "by", "5"], with $|\boldsymbol{s}| = 7$ and $|\boldsymbol{t}| = 6$.

The objective of SMT is to generate the most probable target sentence $\hat{\boldsymbol{t}}$ given a source sentence $\boldsymbol{s}$. Specifically, we do so by defining a model specifying the conditional probability of $\boldsymbol{t}$ given $\boldsymbol{s}$, or $\Pr(\boldsymbol{t}|\boldsymbol{s})$, and find the $\hat{\boldsymbol{t}}$ that maximizes this probability

$$\hat{\boldsymbol{t}} \equiv \arg\max_{\boldsymbol{t}} \Pr(\boldsymbol{t}|\boldsymbol{s}). \quad (1)$$

This probability is estimated using a set of source/target sentence pairs called a "parallel corpus." For example, Fig. 1 is one example of the type of parallel corpus targeted in this

[1]Available at http://ahclab.naist.jp/pseudogen/

```
def fizzbuzz(n):                                     # define the function fizzbuzz with an argument n.
  if not isinstance(n, int):                         #   if n is not an integer value,
    raise TypeError('n is not an integer')           #     throw a TypeError exception with a message ...
  if n % 3 == 0:                                     #   if n is divisible by 3,
    return 'fizzbuzz' if n % 5 == 0 else 'fizz'      #     return 'fizzbuzz' if n is divisible by 5, or 'fizz' if not.
  elif n % 5 == 0:                                   #   if not, and n is divisible by 5,
    return 'buzz'                                    #     return the string 'buzz'.
  else:                                              #   otherwise,
    return str(n)                                    #     return the string representation of n.
```

Source code (Python) Pseudo-code (English)

Fig. 1. Example of source code written in Python and corresponding pseudo-code written in English.

study, in which we have one-by-one correspondences between each line in the source code and pseudo-code.

B. Tree-to-string Machine Translation

In the proposed tool, we use a method called tree-to-string machine translation (T2SMT) [6]. T2SMT works by first obtaining a "parse tree" expressing the structure of the sentence $T_{\boldsymbol{s}}$ from the source tokens $\boldsymbol{s}$. Using this parse tree allows for better handling of the hierarchical structure of the input, which allows for more accurate results both in translation of natural languages [7] and our proposed pseudo-code generation approach [1]. We show an overview of the T2SMT process in Fig. 2

First, we perform *tokenization and parsing*, obtaining the parse tree $T_{\boldsymbol{s}}$ by transforming the input statement into a token array, and converting the token array into a parse tree. In the case of pseudo-code generation we can tokenize and parse the source code by using a compiler or interpreter to analyze the structure of the code, converting it into the underlying tree structure. It should be noted that while most programming language compilers generate "abstract syntax trees" related to the execution-time semantics of the program, the tree in Fig. 2 is that $T_{\boldsymbol{s}}$ is more closely related to the actual surface form of the program, which is necessary for T2SMT. The proposed tool also includes hand-made rules to convert such abstract syntax trees into more concrete syntax trees for Python, and we plan to support more languages in the future.

Next, we perform translation by breaking up the input tree into fragments, using translation patterns to map each fragment into a string of pseudo-code (with wildcard variables) as shown in the *derivation selection* step. In this step, as there are many possible ways to convert a particular piece of source code into pseudo-code, it is necessary to pick a derivation that we will show to the user. We do so by adopting techniques from SMT, which allow us to assign scores to each derivation by calculating a number of features ("translation model" and "language model" probabilities, the number of words in the output, the number of words in the derivation, etc.), and combining them in a sum weighted by a weight vector $\boldsymbol{w}$:

$$\hat{\boldsymbol{t}} = \arg\max_{\boldsymbol{t},\boldsymbol{\phi},\boldsymbol{a}} \boldsymbol{w}^{\mathrm{T}} \boldsymbol{f}(\boldsymbol{t},\boldsymbol{\phi},\boldsymbol{a},\boldsymbol{s}), \tag{2}$$

where $\boldsymbol{f}(\boldsymbol{t},\boldsymbol{\phi},\boldsymbol{a},\boldsymbol{s})$ represents feature functions calculated during the translation process, and $\boldsymbol{w}$ is automatically learned in such a way that maximizes the accuracy on a separate set

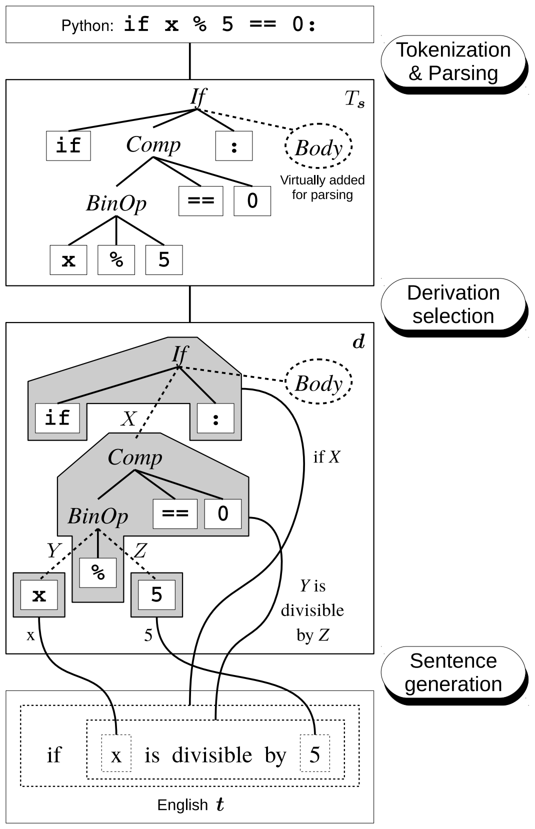


Fig. 2. Example of Python to English T2SMT pseudo-code generation.

of data. More details of this formulation can be found in Oda et al. [1].

C. Constructing an SMT Pseudo-code Generation System

The tool, in its current form, contains pre-trained models to translate basic Python code into English or Japanese. However, one of the attractive features of the proposed tool is that it allows for the easy construction of a pseudo-code generation system, learned directly from data. Thus, it is possible for a user to create a new pseudo-code generator for their preferred pair of programming language and natural language. This section describes the method for creating a new pseudo-code generator.

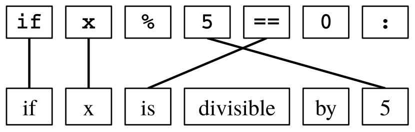


Fig. 3. Word alignment between two languages.

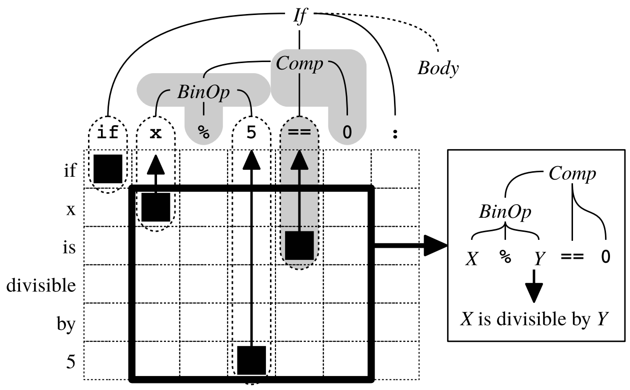


Fig. 4. Extracting T2SMT translation rules according to word alignment.

To create a new pseudo-code generator, it is necessary to extract the translation rules, which define the relationship between the parts of the source and target language sentences. These rules are created from a set of parallel data, consisting of lines of source code and the corresponding pseudo-code. To extract rules, we first obtain a "word alignment" between the tokens of the source code and pseudo-code. Word alignments represent the token-level relationships, as shown in Fig. 3. These word alignments are automatically calculated from the statistics of a parallel corpus by using a probabilistic model and unsupervised machine learning techniques [8], [9].

After obtaining the word alignment of each sentence in the parallel corpus, we use a method known as the GHKM (Galley-Hopkins-Knight-Marcu) algorithm [10] to extract tree-to-string translation rules. The GHKM algorithm first splits the parse tree of the source sentence into several subtrees according to alignments, and extracts the pairs of the subtree and its corresponding words in the target sentence as "minimal rules." Next, the algorithm combines some minimal rules according to the original parse tree to generate larger rules. For example, Fig. 4 shows an extraction of a translation rule corresponding to the phrase with wildcards "X is divisible by Y" by combining minimal rules in the bold rectangle, with two rules corresponding to "x" and "5" being replaced by wildcards respectively.

Finally, the system calculates a number of scores for each rule, and stores them in the rule table for use during translation. These scores are used to calculate the feature functions mentioned in the previous section.

If we want to construct the SMT based pseudo-code generator described in this paper for any programming language/natural language pair, we must prepare the following data and tools.

- **Source code/pseudo-code parallel corpus** to train the SMT based pseudo-code generator.
- **Tokenizer of the target natural language**. If we consider a language that puts spaces between words (e.g. English), we can use a simpler rule-based tokenization method such as the Stanford Tokenizer[2]. If we are targeting a language with no spaces (e.g. Japanese), we must explicitly insert delimiters between each word in the sentence. Fortunately, tokenizing natural language is a well-developed area of NLP, so we can find tokenizers for major languages easily.
- **Tokenizer of the source programming language**. Usually, we can obtain a concrete definition of the programming language from its grammar, and a tokenizer module is often provided as a part of the standard library of the language itself. For example, Python provides tokenizing methods and symbol definitions in the `tokenize` module.
- **Parser of the source programming language**. Similarly to the tokenizer, a parser for the programming language, which converts source code into a parse tree describing the structure of the code, is explicitly defined in the language's grammar. Python also provides the algorithm of abstract parsing in the `ast` module.
- **Conversion rules for abstract syntax trees**. Finally, abstract syntax trees must be converted into more syntactically motivated syntax trees to improve accuracy of translation. This process is dependent on each programming language, and we describe some rules to do so for Python in [1].

III. Two Usage Scenarios

In this section we describe two usage scenarios, the first being a programming beginner using a simple and intuitive interface to generate natural language descriptions of the code he/she is faced with, and the second being an engineer who builds a custom pseudo-code generator for their project to check his/her thought process while coding.

A. Programming Education

In the field of programming education, pseudo-code is often used to describe algorithms. This is because most learners are beginners and do not have knowledge of the programming language at hand, but if they can look at source code and pseudo-code in parallel, they can make associations between the structures of the source code and the intuitively understandable explanation. However, most of the code that program learners should read does not have associated pseudo-code.

`Pseudogen` can be used to generate pseudo-code automatically in this case. As beginner programmers are not likely to want to examine the inner workings of the pseudo-code generator itself, we provide a simple and intuitive web interface available at the tool website.[3] To use the tool, users simply submit a source code fragment, and the web site outputs

[2]http://nlp.stanford.edu/software/tokenizer.shtml
[3]http://ahclab.naist.jp/pseudogen

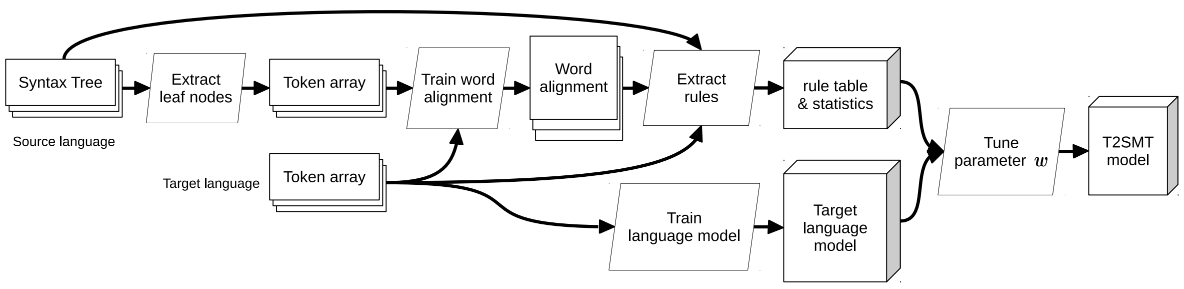


Fig. 5. Training process of the translation framework.

the generated pseudo-code. Fig. 6 shows the web interface of `Pseudogen`. Currently the web interface supports generation of pseudo-code in English or Japanese for Python programs, and it can relatively easily be expanded to other programming and natural languages, which we plan to do in the future.

In experiments [1], we examined the usefulness of automatically generated pseudo-code in helping beginner programmers understand source code. During the experiments, subjects who were beginners replied that being shown pairs of source code and pseudo-code at the same time helped them understand the content of programs in unfamiliar programming languages.

B. Debugging

This tool could also be used by more experienced programmers, with the automatically generated pseudo-code used to help find bugs. Bugs can be roughly divided into two categories: either the algorithm itself is correct but the implementation is incorrect, or the implemented algorithm itself is incorrect. If `Pseudogen` translates incorrect code to pseudo-code, the pseudo-code will explain a process that does not match the intended functioning of the program. This has the potential to help find bugs that occurred due to incorrect implementation, as bugs may be found more easily if the programmer steps back from the implementation in a specific programming language and reads their implementation in natural language. In addition, if a programmer wants to implement an algorithm that already has pseudo-code, they can convert their code into pseudo-code using `Pseudogen`, and check that the results do indeed match the original pseudo-code.

In the case of more advanced users, it will be likely that they want to adapt `Pseudogen` to take input in their own favorite programming language, or generate pseudo-code in their native tongue. In the following two sentences, we explain the training process for those who want to build new pseudo-code generation models or improve the performance of `Pseudogen` on their own. Fig. 5 shows the training process.

In general, the algorithms for training SMT systems from a parallel corpus are too complex to develop from scratch, so `Pseudogen` is based on a combination of a number of open-source tools that are readily available. Specifically, we use the following tools in this study: MeCab to tokenize Japanese sentences [11], pialign to train word alignments [9], KenLM to train language models [12], and Travatar to train and generate target sentences using T2SMT models [13]. The `Pseudogen` site explains how to download and install all of these tools.

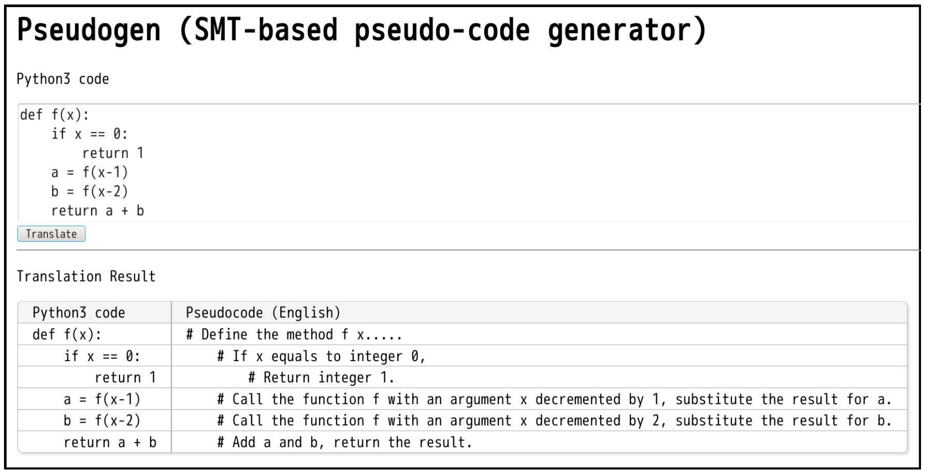
Next, we need to create parallel training data that has source code with corresponding pseudo-code. In the case of the Python-to-English and Python-to-Japanese models provided with `Pseudogen`, we hired two engineers to create this code for us. We obtained 18,805 pairs of Python statements and corresponding English pseudo-code, and for Python-to-Japanese, we obtained 722 pairs.

Next, it is necessary to process this data using the above mentioned tools. Within `Pseudogen`, we include scripts and training programs to perform the whole training process, but we describe it briefly below for completeness: First, we split the data into tokens: for Japanese we used MeCab, for English we used Stanford Tokenizer, and for Python we used the `tokenize` function. Next, we use KenLM to calculate the language model from English and Japanese sentences. Once we have tokenized words and the language model calculated with it, we then train word alignment with pialign. We next generate a syntax tree from source code by first using the Python `ast` parser, followed by the hand-created rules described in the previous section to convert the abstract syntax tree to a more literal format. As engineers will need to implement these rules if they are interested in creating a pseudo-code generator for a language other than Python, we provide the reference implementation for Python with the `Pseudogen` code. Finally, we train the model using learned word alignments, parsed source code and tokenized words.

Once the model is trained, pseudo-code can easily be generated by connecting together these tools, or by starting a web UI included in the package. When given a snippet of source code, `Pseudogen` will parse source code into a syntax tree, and apply the trained translation model to the syntax tree, outputting the pseudo-code.

IV. Related Research/Tools

The aim of `Pseudogen` is to generate natural language pseudo-code that explicitly states the operations performed by the program to aid code understanding. In contrast to this, the great majority of research work on generating natural language from programs focuses on automatic generation of *summary comments*, which concisely describe the behavior of whole functions, or large fragments of code. These comments are generated using hand-made rules [14], [15], [16], [17], or data-driven methods using automatic text summarization [18], topic modeling [19], or information retrieval techniques [20]. In contrast to these methods, `Pseudogen` is able to generate



```
def f(x):
    if x == 0:
        return 1
    a = f(x-1)
    b = f(x-2)
    return a + b
```

Python3 code	Pseudocode (English)
def f(x):	# Define the method f x.....
if x == 0:	# If x equals to integer 0,
return 1	# Return integer 1.
a = f(x-1)	# Call the function f with an argument x decremented by 1, substitute the result for a.
b = f(x-2)	# Call the function f with an argument x decremented by 2, substitute the result for b.
return a + b	# Add a and b, return the result.

Fig. 6. Web interface of Pseudogen

pseudo-code at a much finer granularity, allowing readers to understand the program instructions in detail.

While scarce, there is also some work that focuses on generation of more detailed pseudo-code. For example, the Agile Pseudocode Development Tool[4] allows developers to write "pseudocode diagrams," which can automatically be converted into source code to compile or pseudo-code for non-experts to read. This, however requires creating these diagrams, which is not a part of the standard programming workflow.

Finally, there is work related to *natural language programming*. In contrast to our work, which generates natural language pseudocode from programs, natural language programming attempts to generate programs from natural language. These include methods to program through spoken commands corresponding to Java [21] or use an expressive subset of English that can be automatically converted into source code [22]. There is also some work on using unrestricted language, such as for generating regular expressions [23], or generating input parsers from English specifications of input file formats [24]. While these works are considering transformation in the opposite direction of Pseudogen, it is likely that there is much that these tasks could share.

V. Discussion of Potential Impact

This tool has two major areas of potential to impact the broader software engineering community.

The first is that it provides a general tool to *generate natural language descriptions of source code*. This follows much of the description up until this point, and has implications for programming education and debugging. There are also a number of other areas for which it is potentially useful, including the checking of stale comments, or ensuring that the source code meets specifications. These are not directly handled by the tool currently, but they have the potential to open new directions for future research in these areas.

More broadly, this tool provides a way to *find correspondences between natural language descriptions and source code*. As we mentioned above, if there are relationships between natural language and programming languages, we could also possibly use these to perform natural language programming, converting from pseudo-code descriptions to programming languages. These alignments could also be used for more inquiry-based studies of software, examining, for example, the various ways a particular natural language command could be realized in source code. This could be done by asking several different programmers to write a program corresponding to a particular natural language instruction, taking alignments, and comparing the various types of source code that align to each part of the natural language instruction.

VI. Conclusion

In this paper, we describe a tool Pseudogen, implementing the method of Oda et al. [1], which automatically generates pseudo-code from source code using the framework of statistical machine translation. This is a novel application of SMT, which only requires training data to apply to any kind of programming language and natural language.

In the near future, we plan to create default models for the tool for other programming languages, including major languages such as Java, C++, and C#. We also plan to develop a pseudo-code generator which outputs more abstract descriptions, handling larger structures than single lines of code, consisting of multiple statements. In addition, there is still a problem due to semantic ambiguities. For example, in Python, `a % b` can be evaluated as "a mod b" if a is a numeral, or % notation that replaces % to b in string a if a is a string. Using the type information can partially solve this problem. Although Pseudogen can be applied to any programming language, it does not use variables and method names to generate pseudo-code. It will be an interesting task to take this information into account to generate more appropriate pseudo-code. Finally, we plan to apply similar SMT techniques to automated programming.

[4]http://www.gnstudio.com/open-source/apdt/

Acknowledgment

Part of this works was supported by the Program for Advancing Strategic International Networks to Accelerate the Circulation of Talented Researchers.

References

[1] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation," in *Proc. ASE*, 2015.

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

[3] P. Koehn, *Statistical Machine Translation*. Cambridge University Press, 2010.

[4] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Migrating code with statistical machine translation," in *Proc. ICSE*, 2014.

[5] A. Lopez, "Statistical machine translation," *ACM Computing Surveys*, vol. 40, no. 3, pp. 8:1–8:49, 2008.

[6] L. Huang, K. Knight, and A. Joshi, "Statistical syntax-directed translation with extended domain of locality," in *Proc. AMTA*, vol. 2006, 2006, pp. 223–226.

[7] G. Neubig and K. Duh, "On the elements of an accurate tree-to-string machine translation system," in *Proc. ACL*, 2014.

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

[9] G. Neubig, T. Watanabe, E. Sumita, S. Mori, and T. Kawahara, "An unsupervised model for joint phrase alignment and extraction," in *Proc. ACL-HLT*, Portland, Oregon, USA, 6 2011, pp. 632–641.

[10] M. Galley, M. Hopkins, K. Knight, and D. Marcu, "What's in a translation rule?" in *Proc. NAACL-HLT*, 2004, pp. 273–280.

[11] T. Kudo, K. Yamamoto, and Y. Matsumoto, "Applying conditional random fields to japanese morphological analysis." in *Proc. EMNLP*, vol. 4, 2004, pp. 230–237.

[12] K. Heafield, I. Pouzyrevsky, J. H. Clark, and P. Koehn, "Scalable modified Kneser-Ney language model estimation," in *Proc. ACL*, Sofia, Bulgaria, August 2013, pp. 690–696.

[13] G. Neubig, "Travatar: A forest-to-string machine translation engine based on tree transducers," in *Proc. ACL*, Sofia, Bulgaria, August 2013, pp. 91–96.

[14] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proc. ASE*, 2010, pp. 43–52.

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

[16] R. P. Buse and W. R. Weimer, "Automatic documentation inference for exceptions," in *Proc. ISSTA*, 2008, pp. 273–282.

[17] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013, pp. 23–32.

[18] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proc. WCRE*, 2010, pp. 35–44.

[19] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver, "Evaluating source code summarization techniques: Replication and expansion," in *Proc. ICPC*, 2013, pp. 13–22.

[20] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *Proc. ASE*, 2013, pp. 562–567.

[21] A. Begel and S. Graham, "Spoken programs," in *Proc. IEEE VL/HCC*, 2005.

[22] H. Liu and H. Lieberman, "Metafor: Visualizing stories as code," in *Proc. IUI*, 2005.

[23] N. Kushman and R.Barzilay, "Using semantic unification to generate regular expressions from natural language," in *Proc. NAACL*, 2013.

[24] T. Lei, F. Long, R. Barzilay, and M. Rinard, "From natural language specifications to program input parsers," in *Proc. ACL*, 2013.