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

Learning to Generate Pseudo-code from Source Code using Statistical Machine Translation

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata,
Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura
Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0192, Japan
{oda.yusuke.on9, fudaba.hiroyuki.ev6, neubig, hata, ssakti, tomoki, s-nakamura}@is.naist.jp

***Abstract*—Pseudo-code written in natural language can aid the comprehension of source code in unfamiliar programming languages. However, the great majority of source code has no corresponding pseudo-code, because pseudo-code is redundant and laborious to create. If pseudo-code could be generated automatically and instantly from given source code, we could allow for on-demand production of pseudo-code without human effort. In this paper, we propose a method to automatically generate pseudo-code from source code, specifically adopting the statistical machine translation (SMT) framework. SMT, which was originally designed to translate between two natural languages, allows us to automatically learn the relationship between source code/pseudo-code pairs, making it possible to create a pseudo-code generator with less human effort. In experiments, we generated English or Japanese pseudo-code from Python statements using SMT, and find that the generated pseudo-code is largely accurate, and aids code understanding.**

Keywords: Algorithms, Education, Statistical Approach

I. Introduction

Understanding source code is an essential skill for all programmers. This is true for programmers at all levels, as it is necessary to read and understand code that others have written to, for example, work efficiently in a group or to integrate and modify open-source software. On a macro level, there are a variety of tools to aid engineers in comprehending the overall structure of programming projects. For example, DeLine et al. proposed a tool that allows programming experts to evaluate large-scale cooperative software engineering projects [1]. Rahman et al. also proposed a system that recommends methods for fixing source code when it does not work as expected [2].

On a more fine-grained level, when we try to understand the behavior of source code in detail, we usually need to read each statement in the source code carefully, and understand what each statement does. Of course, thoroughly reading and understanding source code of existing software is possible (although time consuming) for veteran programmers. In contrast, this process is much more difficult for beginner programmers or programmers who are learning a new programming language. Such inexperienced readers sometimes do not understand the grammar and style of the source code at hand, so reading source code written in such languages imposes a large burden.

On the other hand, in educational texts about programming and algorithms, it is common to use "pseudo-code," which describes the behavior of statements in the program using natural language (usually in English, or the programmers' mother tongue) or mathematical expressions. Pseudo-code aids comprehension of beginners because it explicitly describes what the program is doing, but is more readable than an unfamiliar programming language.

Fig. 1 shows an example of Python source code, and English pseudo-code that describes each corresponding statement in the source code.[1] If the reader is a beginner at Python (or a beginner at programming itself), the left side of Fig. 1 may be difficult to understand. On the other hand, the right side of the figure can be easily understood by most English speakers, and we can also learn how to write specific operations in Python (e.g. if we want to check if the type of the variable is not an integer, we can see that we write "`if not isinstance(`*something*`, int):`"). In other words, pseudo-code aids the "bottom-up comprehension" [3] of given source code.

However, in real programming environments, pseudo-code corresponding to source code rarely exists, because pseudo-code is not necessary once a programmer has a good grasp of the programming language and project. In addition, indiscriminately inserting pseudo-code into existing source code manually would impose a large burden on programmers. On the other hand, if pseudo-code could be generated automatically, this burden could be reduced, and pseudo-code could be created for actual programming projects. If we want to practically use automatically generated pseudo-code, we can say that satisfying the following 4 points is required:

- pseudo-code is accurate enough to describe the behavior of the original source code,
- pseudo-code should be provided upon the readers' request,
- pseudo-code should be automatically generated to avoid the burden on programmers, and
- the method to generate pseudo-code should be efficient, to avoid making the readers wait.

In this study, we propose a method to automatically generate pseudo-code from source code. In particular, our proposed method makes two major contributions:

[1] While there are many varieties of pseudo-code, in this paper we assume that pseudo-code is "line-to-line" translation between programming and natural languages as shown by Fig. 1. This assumption clearly defines the relationship between source code and pseudo-code and is a convenient first step towards applying machine translation to this task.

978-1-5090-0025-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ASE.2015.36

```
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.

- To our knowledge, this is the first method for generating pseudo-code that completely describes the corresponding source code. This should be contrasted with previous work on comment generation, which aims to help experienced engineers by reducing the amount of source code to be read, and is described in §II.
- We propose a framework to perform pseudo-code generation using statistical machine translation (SMT). SMT is a technology that can automatically learn how to translate between two languages, and was designed for translating between natural languages, such as English and Japanese. Our proposed method of applying this to pseudo-code generation has several benefits, the largest of which being that it is easy to extend the generator to other combinations of programming and natural languages simply by preparing data similar to that in Fig. 1.

In this paper, we first refer to related works on automatic comment generation and clarify the differences and the merits of our method (§II). Second, we describe the summary of two SMT frameworks to be used in this study (§III). Next, we explain how to apply the SMT framework to pseudo-code generation (§IV), how we gather source code/pseudo-code parallel data to train our pseudo-code generation system (§V), and the method to evaluate the generated pseudo-code using automatic and code understanding criteria (§VI). In experiments, we apply our pseudo-code generation system to the Python-to-English and Python-to-Japanese pseudo-code generation tasks, and we find that providing automatically generated pseudo-code along with the original source code makes the code easier to understand for programming beginners (§VII). We finally mention the conclusion and future directions, including applications to other software engineering tasks (§VIII).[2]

II. Related Works

There is a significant amount of previous work on automatic comment and document generation. The important difference between our work and conventional studies is the motivation. Previous works are normally based on the thought of "reducing" the amount of code to be read. This is a plausible idea for veteran engineers, because their objective is to understand a large amount of source code efficiently, and thus comments are expected to concisely summarize what the code is doing, instead of covering each statement in detail. From a technical point of view, however, pseudo-code generation is similar to automatic comment generation or automatic documentation as both generate natural language descriptions from source code, so in this section we outline the methods used in previous approaches and contrast them to our method.

There are two major paradigms for automatic comment generation: *rule-based* approaches, and *data-based* approaches. Regarding the former, for example, Sridhara et al. perform summary comment generation for Java methods by analyzing the actual method definition using manually defined heuristics [4], [5]. Buse et al. also proposed a method to generate documents that include the specification of exceptions that could be thrown by a function and cases in which exceptions occur [6]. Moreno et al. also developed a method to generate summaries that aid understanding, especially focusing on class definitions [7]. These rule-based approaches use detailed information closely related to the structure of source code, allowing for handling of complicated language-specific structures, and are able to generate accurate comments when their rules closely match the given data. However, if we need new heuristics for a specific variety of source code or project, or to create a system for a new programming language or natural language, the system creator must manually append these heuristics themselves. This causes a burden on the maintainers of comment generation systems, and is a fundamental limitation of rule-based systems.

On the other hand, in contrast to rule-based systems, *data-based* approaches can be found in the comment generation or code summarization fields. Wong et al. proposed a method for automatic comment generation that extracts comments from entries of programming question and answer sites using information retrieval techniques [8]. There are also methods to generate summaries for code based on automatic text summarization [9] or topic modeling [10] techniques, possibly in combination with the physical actions of expert engineers [11]. This sort of data-based approach has a major merit: if we want to improve the accuracy of the system, we need only increase the amount of "training data" used to construct it. However, existing methods are largely based on retrieving already existing comments, and thus also have significant issues with "data sparseness;" if a comment describing the existing code doesn't already exist in the training data, there is no way to generate accurate comments.

[2]Datasets used to construct and evaluate our pseudo-code generators are available at http://ahclab.naist.jp/pseudogen/

SMT, which we describe in the next section, combines the advantages of these approaches, allowing for detailed generation of text, like the rule-based approaches, while being learnable from data like the data-based approaches.

III. Statistical Machine Translation

SMT is an application of natural language processing (NLP), which discovers the lexical or grammatical relationships between two natural languages (such as English and Japanese), and converts sentences described in a natural language into another natural language. SMT algorithms used in recent years are mainly based on two ideas:

- extract the relationships (usually called "rules") between small fragments of new input and output languages, and
- use these relationships to synthesize the translation results of a new input sentence using statistical models to decide which translation is best [12], [13], [14].

SMT frameworks have quickly developed in recent years, mainly because of the large amount of data available and the increase in calculation power of computers. In this section, we describe the foundations of the SMT framework. Especially, we explain the details of the phrase-based machine translation (PBMT) and tree-to-string machine translation (T2SMT), which are major SMT frameworks, used in this work to convert source code into pseudo-code.

A. Overview of Statistical Machine Translation

At first, we introduce some notation for the formulation of SMT-based pseudo-code generation. Let $\boldsymbol{s} = [s_1, s_2, \cdots, s_{|\boldsymbol{s}|}]$ describe the "source sentence," an array of input tokens, and $\boldsymbol{t} = [t_1, t_2, \cdots, t_{|\boldsymbol{t}|}]$ 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 distribution of $\boldsymbol{t}$ given $\boldsymbol{s}$, or $\Pr(\boldsymbol{t}|\boldsymbol{s})$, and find the $\hat{\boldsymbol{t}}$ that maximizes this probability:

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

The difference of each SMT framework is guided by the method for calculating $\Pr(\boldsymbol{t}|\boldsymbol{s})$. 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 study, in which have one-by-one correspondences between each line in the source code and pseudo-code.

B. Phrase-based Machine Translation

One of the most popular SMT frameworks is phrase-based machine translation (PBMT) [15], which directly uses the phrase-to-phrase relationships between source and target language pairs. In software engineering studies, Karaivanov et al. proposed a method applying the PBMT framework to programming language conversion, which learns the relationships between parts of statements in two programming languages (e.g. "`System.out.println`" in Java to "`Console.WriteLine`" in C#) [16].

To describe PBMT modeling, we introduce a set of phrase pairs $\boldsymbol{\phi} = [\phi_1, \phi_2, \cdots \phi_{|\boldsymbol{\phi}|}]$. Each $\phi_n = \left\langle \boldsymbol{s}^{(n)} \to \boldsymbol{t}^{(n)} \right\rangle$ represents the n-th subsequence of the source sentence $\boldsymbol{s}^{(n)}$ and the target subsequence $\boldsymbol{t}^{(n)}$ associated with the corresponding source subsequence. For example, in Fig. 2, $\boldsymbol{s}$ is separated into $|\boldsymbol{\phi}| = 4$ phrases:

$$\boldsymbol{\phi} = \begin{bmatrix} \langle & \text{"}\texttt{if}\text{"} & \to & \text{"if"} & \rangle \\ \langle & \text{"}\texttt{x}\text{"} & \to & \text{"x"} & \rangle \\ \langle & \text{"}\texttt{\% 5}\text{"} & \to & \text{"by 5"} & \rangle \\ \langle & \text{"}\texttt{== 0 :}\text{"} & \to & \text{"is divisible"} & \rangle \end{bmatrix}. \tag{2}$$

$\boldsymbol{\phi}$ is generated using a "phrase table", which contains various phrase-to-phrase relationships with probabilities, and is extracted from a parallel corpus as explained in §III-D.

Given $\boldsymbol{\phi}$, we can generate the target sentence $\boldsymbol{t}$ from the source sentence by concatenating each $\boldsymbol{t}^{(n)}$. But simply concatenating $\boldsymbol{t}^{(n)}$ according to their order cannot obtain an accurate target sentence, because the grammatical characteristics (e.g. ordering of tokens) of the source and target languages are usually different. For example, if we concatenate the target side phrases of Equation (2), we obtain the target sentence "if x by 5 is divisible," which is not a fluent English sentence.

To avoid this problem, we need to perform "reordering," which chooses the proper order of phrases in the target sentence. To express reordering, we introduce the phrase alignment $\boldsymbol{a} = [a_1, a_2, \cdots, a_{|\boldsymbol{\phi}|}]$, where each a_n is an integer that represents the order of the n-th phrase pair in the source sentence. In Fig. 2, we assume that $\boldsymbol{a} = [1, 2, 4, 3]$, which means that first and second phrase pairs keep their own positions, and the third and fourth phrase pairs are swapped before the target sentence is generated.

Formally, the conditional probability $\Pr(\boldsymbol{t}|\boldsymbol{s})$ of the PBMT model is usually estimated using a log-linear model, that combines several probabilities calculated over the source and target sentences [17]:

$$\begin{aligned} \hat{\boldsymbol{t}} &\equiv \arg\max_{\boldsymbol{t}} \Pr(\boldsymbol{t}|\boldsymbol{s}) && (3) \\ &\simeq \arg\max_{\boldsymbol{t},\boldsymbol{\phi},\boldsymbol{a}} \Pr(\boldsymbol{t}, \boldsymbol{\phi}, \boldsymbol{a}|\boldsymbol{s}) && (4) \\ &\simeq \arg\max_{\boldsymbol{t},\boldsymbol{\phi},\boldsymbol{a}} \frac{\exp(\boldsymbol{w}^{\mathrm{T}} \boldsymbol{f}(\boldsymbol{t}, \boldsymbol{\phi}, \boldsymbol{a}, \boldsymbol{s}))}{\sum_{\boldsymbol{t}'} \exp(\boldsymbol{w}^{\mathrm{T}} \boldsymbol{f}(\boldsymbol{t}', \boldsymbol{\phi}, \boldsymbol{a}, \boldsymbol{s}))} && (5) \\ &= \arg\max_{\boldsymbol{t},\boldsymbol{\phi},\boldsymbol{a}} \boldsymbol{w}^{\mathrm{T}} \boldsymbol{f}(\boldsymbol{t}, \boldsymbol{\phi}, \boldsymbol{a}, \boldsymbol{s}), && (6) \end{aligned}$$

where $\boldsymbol{f}(\boldsymbol{t}, \boldsymbol{\phi}, \boldsymbol{a}, \boldsymbol{s})$ represents feature functions calculated during the translation process, and $\boldsymbol{w}$ represents the weight vector of the corresponding features, which defines the importance of each feature. Intuitively, Equation (6) means that the PBMT

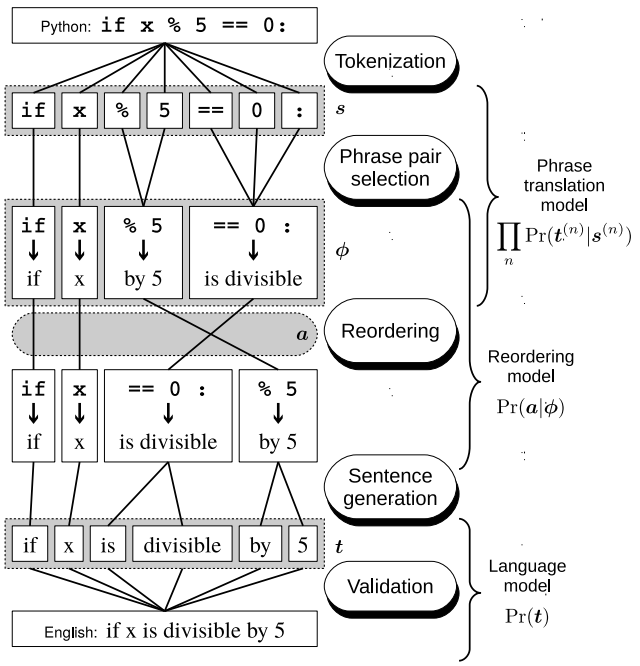

Fig. 2. Example of Python to English PBMT pseudo-code generation.

model finds the sentence which has the highest "score" calculated by the weighted sum of the features: $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{f}(\boldsymbol{t}, \boldsymbol{\phi}, \boldsymbol{a}, \boldsymbol{s})$. Some typical examples of these features include:

- **Language model** $\Pr(\boldsymbol{t})$ measures the fluency of the sentence $\boldsymbol{t}$ under the target language as described in §III-E.
- **Phrase translation model** calculates the product of the probabilities $\Pr(\boldsymbol{t}^{(n)}|\boldsymbol{s}^{(n)})$ of the individual phrases in the sentence.
- **Reordering model** $\Pr(\boldsymbol{a}|\boldsymbol{\phi})$ calculates the probability of the arranging each phrase in a particular order.

While PBMT's mechanism of translating and reordering short phrases is simple, it also lacks the expressive power to intuitively model pseudo-code generation. For example, we intuitively know that the English string including two wildcards "X is divisible by Y" corresponds to the source code "X `%` Y `==` `0:`," but PBMT is not capable of using these wildcards. Thus, $\boldsymbol{\phi}$ in the example in Fig. 2 uses obviously "wrong" correspondences such as ⟨"`==0:`" → "is divisible"⟩. In addition, source code has an inherent hierarchical structure which can not be used by explicitly when only performing phrase-to-phrase translation and reordering.

C. Tree-to-string Machine Translation

As we mentioned in the previous section, PBMT-based pseudo-code generation cannot take advantage of wildcards or the hierarchical correspondences between two languages. T2SMT uses the parse tree of source sentence $T_{\boldsymbol{s}}$ instead of the source tokens $\boldsymbol{s}$ to avoid this problem [18], as shown in Fig. 3.

T2SMT was originally conceived for translating natural languages such as English, and because natural languages include ambiguities we obtain the parse tree $T_{\boldsymbol{s}}$ using a probabilistic parser that defines the probability $T_{\boldsymbol{s}}$ given $\boldsymbol{s}$, or $\Pr(T_{\boldsymbol{s}}|\boldsymbol{s})$ [19], [20]. Thus, the formulation of T2SMT can be obtained by introducing this parsing probability into Equation (1):

$$\hat{\boldsymbol{t}} = \arg\max_{\boldsymbol{t}} \Pr(\boldsymbol{t}|\boldsymbol{s}) \tag{7}$$

$$\simeq \arg\max_{\boldsymbol{t}, T_{\boldsymbol{s}}} \Pr(\boldsymbol{t}|T_{\boldsymbol{s}}) \Pr(T_{\boldsymbol{s}}|\boldsymbol{s}). \tag{8}$$

Fortunately, the probability $\Pr(T_{\boldsymbol{s}}|\boldsymbol{s})$ can be ignored in our proposed method for pseudo-code generation, because all practical programming languages have a compiler or interpreter that can parse the corresponding source code deterministically, and thus there is only one possible parse tree. So the formulation is less complex:

$$\hat{\boldsymbol{t}} \simeq \arg\max_{\boldsymbol{t}} \Pr(\boldsymbol{t}|T_{\boldsymbol{s}}), \tag{9}$$

Fig. 3 shows the process of T2SMT-based pseudo-code generation. First, we obtain the parse tree $T_{\boldsymbol{s}}$ by transforming the input statement into a token array using tokenization, and parsing the token array into a parse tree using parsing. The important point of Fig. 3 is that $T_{\boldsymbol{s}}$ is defined by the grammar of the programming language, and is not an "abstract" syntax tree. This requirement comes from the characteristics of the inner workings of the T2SMT algorithm, and this topic is described in detail in §IV.

The probability $\Pr(\boldsymbol{t}|T_{\boldsymbol{s}})$ is defined similarly to that of PBMT (usually formulated as a log-linear model) with two major differences:

- In the T2SMT, we use the "derivation" $\boldsymbol{d} = [d_1, d_2, \cdots, d_{|\boldsymbol{d}|}]$ instead of the phrase pairs $\boldsymbol{\phi}$ in PBMT. Each $d_n = \left\langle T_s^{(n)} \rightarrow \boldsymbol{t}^{(n)} \right\rangle$ represents the relationship between between a source subtree (the gray boxes in Fig. 3) and target phrase with wildcards. All derivations are connected according to the structure of original parse tree $T_{\boldsymbol{s}}$, and the target sentence is generated by replacing wildcards with their corresponding phrases.
- The T2SMT translation process does not include explicit reordering models because the ordering of the wildcards in the target phrase naturally defines the target ordering.

D. Extracting SMT Rules

To train the PBMT and T2SMT models, we have to extract the translation rules, which define the relationship between the parts of the source and target language sentences, from the given parallel corpus. To do this, we use a "word alignment" between both sentences. Word alignments represent the word-to-word level relationships between both languages, shown in Fig. 4. In standard SMT training, word aligments are automatically calculated from the statistics of a parallel corpus by using a probabilistic model and unsupervised machine learning techniques [21], [22].

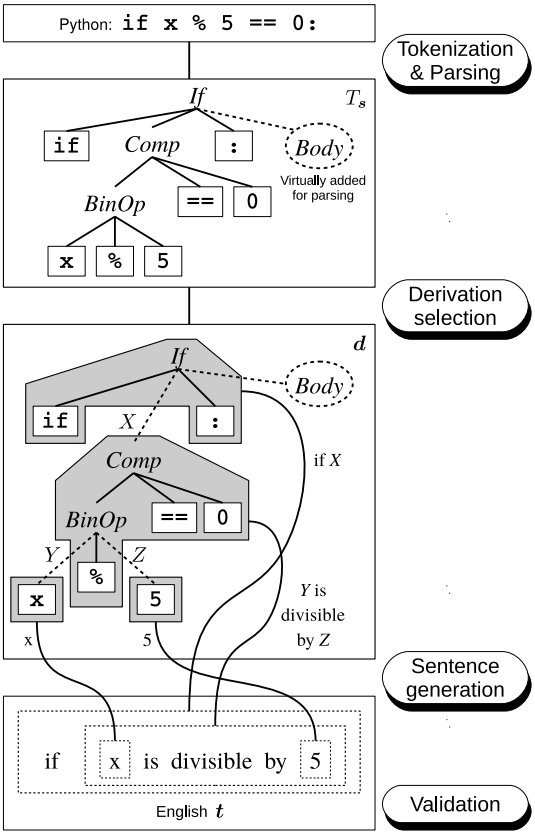

Fig. 3. Example of Python to English T2SMT pseudo-code generation.

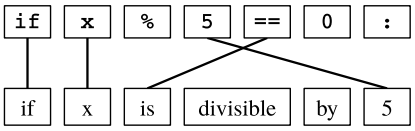

Fig. 4. Word alignment between two token strings.

After obtaining the word alignment of each sentence in the parallel corpus, we assume that the phrases that satisfy the below conditions can be extracted as a phrase pair for the PBMT framework:

- some words in both phrases are aligned, and
- no words outside of the phrases are aligned to a word in either phrase.

For example, Fig. 5 shows one phrase pair extraction $\phi = \langle$"== 0 :" $\rightarrow$ "is divisible"$\rangle$.

In the case of the T2SMT framework, we use a method known as the GHKM algorithm [23] 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,

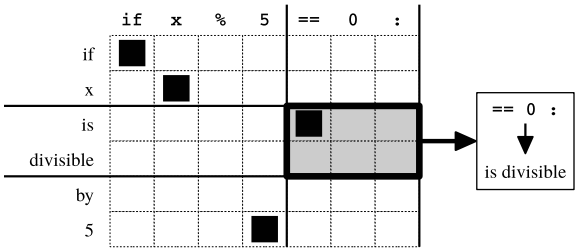

Fig. 5. Extracting PBMT translation rules according to word alignment.

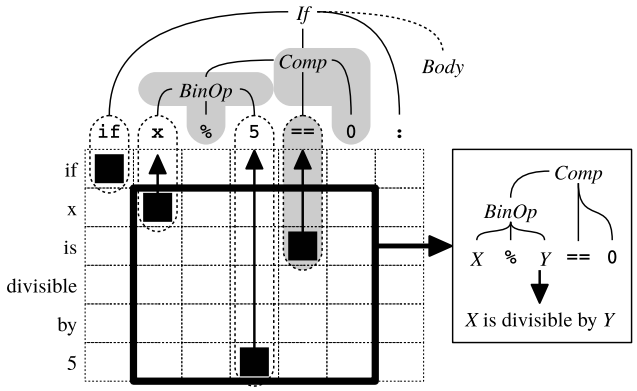

Fig. 6. Extracting T2SMT translation rules according to word alignment.

Fig. 6 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.

Extracted PBMT and T2SMT rules are evaluated using some measures and stored into the rule table for each framework along with these evaluation scores. These scores are used to calculate feature functions that are used in the calculation of probabilities $\Pr(\boldsymbol{t}|\boldsymbol{s})$ or $\Pr(\boldsymbol{t}|T_{\boldsymbol{s}})$. For example, the frequency of the rule in the parallel corpus, length of the phrase, and the complexity of the subtree are often used as features.

E. Language Model

Another important feature function of SMT is the "language model," which evaluates the fluency of the sentence in the target language. Given the target sentence $\boldsymbol{t}$, the language model is defined as the product of probabilities of each word in $\boldsymbol{t}$ given the previous words:

$$\Pr(\boldsymbol{t}) \equiv \prod_{i=1}^{|\boldsymbol{t}|} \Pr(t_i|t_1, t_2, \cdots, t_{i-1}) \quad (10)$$

$$\equiv \prod_{i=1}^{|\boldsymbol{t}|} \Pr(t_i|t_1^{i-1}), \quad (11)$$

where the notation $x_i^j = [x_i, x_{i+1}, \cdots, x_j]$ represents the subsequence of $\boldsymbol{x}$ containing the i-th to j-th elements. In addition, to save memory and prevent data sparseness, most

practical language models use "n-gram models:"

$$\Pr(t_i|t_1^{i-1}) \simeq \Pr(t_i|t_{i-n+1}^{i-1}), \tag{12}$$

where an n-gram is defined as n consecutive words. This approximation means that the next word t_i is conditioned on only on the previous $(n-1)$ words. The simplest n-gram model can be calculated simply from target language text using the count of appearances of each n-gram:

$$\Pr(t_i|t_{i-n+1}^{i-1}) \equiv \frac{\mathrm{count}(t_{i-n+1}^{i})}{\mathrm{count}(t_{i-n+1}^{i-1})} \tag{13}$$

where $\mathrm{count}(\boldsymbol{x})$ is the number of appearances of sequence $\boldsymbol{x}$ in the given corpus. For example, if the word (1-gram) "is" appears 10 times and the 2-gram "is divisible" appears 3 times in the same corpus, then we can estimate $\Pr(t_i =$ "divisible"$|t_{i-1} =$ "is"$) = 3/10$. In addition, we also use a further approximation method known as Kneser-Ney smoothing [24], which smooths probabilities for all n-grams, preventing problems due to data sparsity, allowing us to calculate the probability of any sentence accurately.

It should also be noted that n-gram models are easily applied to any type of sequence data and frequently used for software engineering, typically to measure the naturalness of the source code [25], [26] or distinguish the characteristics of source code [27], [28].

IV. Applying SMT to Pseudo-code Generation

In the previous section, we described two SMT frameworks: PBMT and T2SMT. The important advantage of using these frameworks is that we need not describe new translation rules explicitly when we update the pseudo-code generator, because the SMT framework is a statistical approach and translation rules from programming language to natural language can be automatically obtained from training data. This fact greatly reduces the burden on engineers to create or maintain pseudo-code generators. If we want to cover a new case in our generator, we simply search for or create pseudo-code corresponding to the desired statements, instead of creating specialized rules for each new case. For example, if we want to create a rule "if X is an even number" for the source code "`if` X `% 2 == 0:`" instead of "if X is divisible by 2," we only need to append a sentence including this example to our corpus, for example, "if something is an even number" and "`if something % 2 == 0:`." This work is obviously easier than describing rules explicitly, as this sort of data can be created by programmers even if they are not familiar with our particular pseudo-code generating system, and could also potentially be harvested from existing data.

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[3]. 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 studies, 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 a way of performing abstract parsing in the `ast` module. However, the output of this module cannot be used directly, as described in the next section.

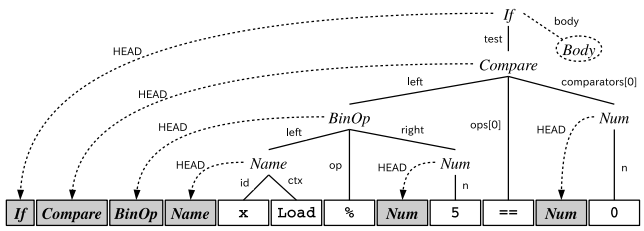

Fig. 7. The head insertion process.

A. Surface Modification of Abstract Syntax Tree

As with the Python library `ast`, parsing methods provided by standard libraries of programming languages are often "abstract" parsers, which are defined more by the execution-time semantics of the language than its written form. As we mentioned in the previous section, the GHKM heuristics, which extract tree-to-string translation rules, use word alignments, which are defined over the leaves of the syntax tree. Abstract syntax trees often use keywords and operators existing in the source code as internal nodes rather than leaves, but these surface words could be strongly related to specific words in the target language (e.g. the token "`if`" in Python corresponds to the word "if" in English).

Of course, we could always develop a new parser from grammar definitions for specific programming languages, but this is not always easy (e.g. C++ grammar is defined using hundreds of pages in the official specification document). In this paper, we apply a more reasonable way to generate a parse-like tree from an abstract syntax tree using two processes described below.

1) Head Insertion: First, we append a new edge called "HEAD" into the abstract syntax tree, which include the label of inner nodes as their leaves. Fig. 7 shows an example of

[3]http://nlp.stanford.edu/software/tokenizer.shtml

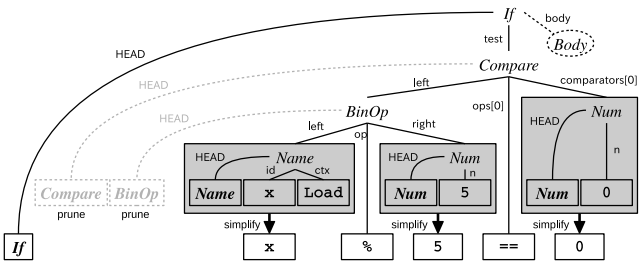

Fig. 8. Pruning and simplification process.

the head insertion process for a tree generated from the source code "`if x % 5 == 0:`" using the `ast` module in Python. Applying this process, words that have disappeared in the process of making the abstract syntax tree such as "`if`" can be treated as candidates for word alignment. This process is simple and easy to be applied to abstract syntax trees of any programming language if we can generate their trees using an abstract parser. On the other hand, there is one option of the head insertion process: where to position the HEAD nodes in the tree. In this paper, we put all HEAD nodes at the leftmost child of the parent for English and rightmost child for Japanese, corresponding to the order of the head word in the target language.

2) Pruning and Simplification: In expanded trees the number of their leaves after head insertion becomes significantly larger than the number of words in the target sentence, and this could cause noise in automatic word alignment. To fix this, we apply some pruning and simplification heuristics to reduce the complexity of trees. We developed 20 hand-written rules to prune and simplify the head-inserted tree by removing nodes that don't seem related to the surface form of the language. Fig. 8 shows an example after this process was applied to the result of Fig. 7.

In our method for pseudo-code generation, only these transformation methods require human engineering. However, developing these methods is easier than developing a rule-based system, because they are relatively simple and largely independent of the language of the pseudo-code.

B. Training Process of Pseudo-code Generator

Up until this section, we described details of each part of our pseudo-code generation system.

Fig. 9 shows the whole process of our pseudo-code generator, and Fig. 10 shows the training process of the PBMT and T2SMT frameworks. In this paper, we compare 4 types of methods for pseudo-code generation. *PBMT* is a method directly using the tokens of programming and natural languages in the PBMT framework. *Raw-T2SMT* is a T2SMT-based method trained by raw abstract syntax trees of the programming language without the modifications described in the previous section. *Head-T2SMT* is also a T2SMT-based method trained using modified trees with only the head insertion process. *Reduced-T2SMT* is the last T2SMT-based method, using the head insertion, pruning, and simplification processes.

The algorithms for training SMT systems from a parallel corpus are too complex to develop ourselves. Fortunately, we can use open-source tools to assist in constructing the system. We use following tools in this study: MeCab to tokenize Japanese sentences [29], pialign to train word alignments [22], KenLM to train Kneser-Ney smoothed language model [30], Moses to train and generate target sentences using the PBMT model [31], and Travatar to train and generate target sentences using T2SMT models [32].

V. Gathering a Source Code to Pseudo-code Parallel Corpus

As we described in §III, we need a source code/pseudo-code parallel corpus to train SMT-based pseudo-code generators. In this study, we created Python-to-English and Python-to-Japanese parallel corpora by hiring programmers to add pseudo-code to existing code.

For the Python-to-English corpus, we contracted one engineer to create pseudo-code for Django (a Web application framework), and obtained a corpus containing 18,805 pairs of Python statements and corresponding English pseudo-code. For Python-to-Japanese, we first hired one engineer to create Python code for solutions to problems from Project Euler[4]. a site with arithmetic problems designed for programming practice. We obtained 722 Python statements, which include 177 function definitions related to solving arithmetic problems. These are used to perform human evaluation experiments described in §VI-B and §VI-C. This code is shown to another Japanese engineer, who created Japanese pseudo-code corresponding to each statement. It should be noted that in both cases, this pseudo-code was created by a separate engineer not involved with this work, demonstrating that no special knowledge of the proposed system is required to create training data for it.

Next, we divided this data into separate sets for our experiments. We split the Python-to-English corpus into 3 parts that include 16,000, 1,000, and 1,805 statements. The 16,000 element set is the "training" data used to extract SMT rules and train the language model. The next 1,000 is the "development" data used to optimize the weight vector $\boldsymbol{w}$ of the log-linear model. The last 1,805 is the "test" data used to evaluate the trained SMT models. The Python-to-Japanese corpus is smaller than the Python-to-English corpus, so we perform 10-fold cross-validation, in which we use 90% of the data as training data, no development data ($\boldsymbol{w}$ is set as the default of each SMT framework), and 10% as test data.

VI. Evaluating Pseudo-code Generation

Once we have generated pseudo-code, we would like to evaluate its accuracy and usefulness. To do so, we use an automatic measure of translation accuracy adopted from the SMT literature, a manual evaluation of the accuracy of code generation, and a measure of how much the pseudo-code can contribute to code understanding. As described in the following sections, we calculate automatic evaluation measures for English and Japanese pseudo-code and manual accuracy and code understanding for Japanese pseudo-code.

[4]https://projecteuler.net/

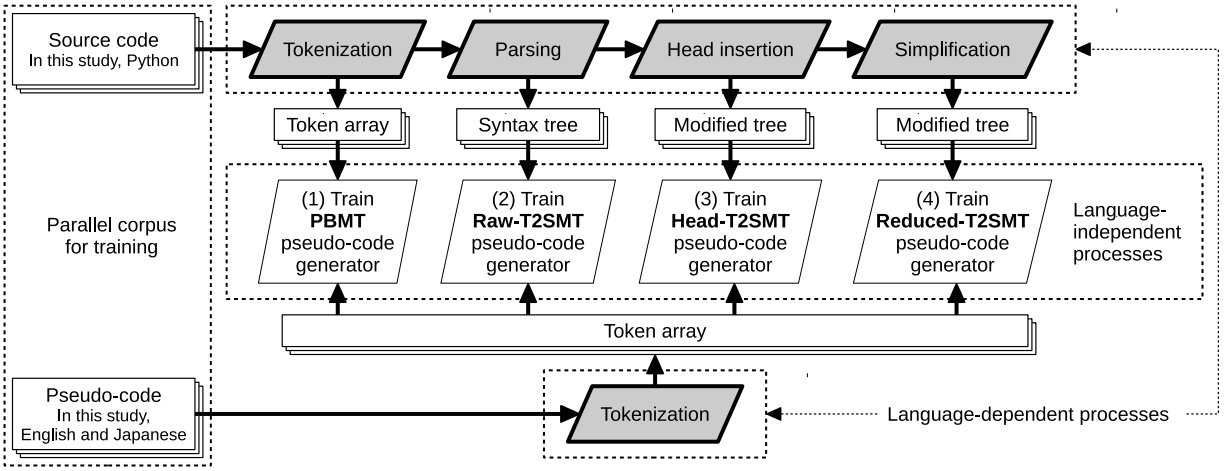

Fig. 9. Whole training process of each proposed method (a bold border indicates language-dependent processes).

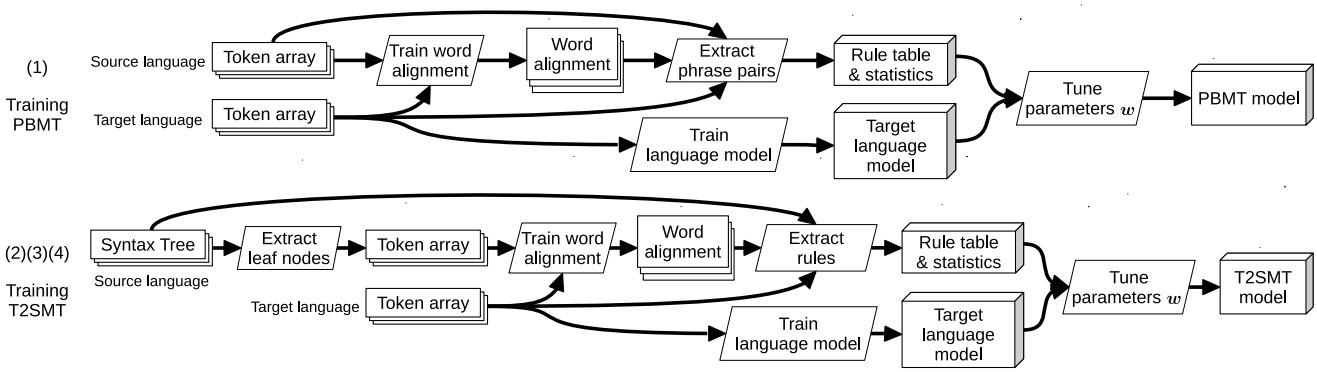

Fig. 10. Training process of PBMT and T2SMT frameworks.

A. Automatic Evaluation – BLEU

First, to automatically measure the accuracy of pseudo-code generation, we use BLEU (Bilingual Evaluation Understudy) [33], an automatic evaluation measure of the generated translations widely used in machine translation studies. BLEU automatically calculates the similarity of generated translations and human-created reference translations. BLEU is defined as the product of "n-gram precision" and a "brevity penalty." n-gram precision measures the ratio of length n word sequences generated by the system that are also created in the human reference, and the brevity penalty is a penalty that prevents the system from creating overly short hypotheses (that may have higher n-gram precision). BLEU gives a specific real value with range [0,1] and is usually expressed as a percentage. If the translation results are completely equal to the references, the BLEU score becomes 1.

In this study, we evaluated the quality of generated English and Japanese pseudo-code using BLEU, using the human-described pseudo-code of each statement as a reference.

B. Human Evaluation (1) – Acceptability

BLEU can automatically and quickly calculate the accuracy of generated translations based on references. However, it does not entirely guarantee that each translation result is semantically correct for the source sentence. To more accurately measure the quality of the translations, we also perform an evaluation using human annotators according to the acceptability criterion [34] shown in TABLE I. We employed 5 Japanese expert Python programmers to evaluate statement-wise acceptabilities of each generated Japanese pseudo-code.

TABLE I. DEFINITION OF ACCEPTABILITY

Level	Meaning
AA (5)	Grammatically correct, and fluent
A (4)	Grammatically correct, and not fluent
B (3)	Grammatically incorrect, and easy to understand
C (2)	Grammatically incorrect, and difficult to understand
E (1)	Not understandable or some important words are lacking

C. Human Evaluation (2) – Code Understanding

Especially for beginner programmers, pseudo-code may aid comprehension of the corresponding source code. To examine this effect, we perform a human evaluation of code understanding using experienced and unexperienced Japanese Python programmers through a Web interface.

First, we show a sample of a function definition and

TABLE II. DEFINITION OF UNDERSTANDABILITY

Level	Meaning
5	Very easy to understand
4	Easy to understand
3	Not either easy or difficult to understand
2	Difficult to understand
1	Very difficult to understand
0	Do not understand

TABLE III. BLEU% AND MEAN ACCEPTABILITIES OF EACH PSEUDO-CODE GENERATOR.

Pseudo-code Generator	BLEU% (English)	BLEU% (Japanese)	Mean Acceptability (Japanese)
PBMT	25.71	51.67	3.627
Raw-T2SMT	49.74	55.66	3.812
Head-T2SMT	47.69	59.41	4.039
Reduced-T2SMT	**54.08**	**62.88**	**4.155**

corresponding pseudo-code as in Fig. 1 to unexperienced programmers, who read the code. These programmers then assign a 6-level score indicating their impression of how well they understood the code for each sample. This impression is similar to Likert scale described in TABLE II. The evaluation interface records these scores and elapsed time from proposing a sample to the programmer submitting the score. This elapsed time can be assumed to be the time required by the programmer to understand the sample. Next, we have the programmers describe the behavior of functions they read in their mother tongue. We obtained these results from 14 students, in which 6 students had less than 1 year of Python experience (including no experience), perform this experiment task.

We use 117 function definitions in the Python-to-Japanese corpus that were split into 3 settings randomly (each split was different for each subject), which were respectively shown with different varieties of pseudo-code:

- *Code* setting that shows only source code itself, with no pseudo-code.
- *Reference* setting that shows human-created pseudo-code (i.e. training data of the pseudo-code generator).
- *Automated* setting that shows code automatically generated by our *Reduced-T2SMT* method.

VII. EXPERIMENTAL RESULTS AND DISCUSSION

First, TABLE III shows the BLEU scores for each of the proposed methods for Python-to-English and Python-to-Japanese datasets, and the mean acceptability score of each method for the Python-to-Japanese dataset.

From these results, we can see that the BLEU scores are relatively high (except for PBMT on the Python-to-English data set), suggesting that the proposed method is generating relatively accurate results for both data sets. For reference, a current state-of-the-art SMT system achieves a BLEU score of about 48 in the relatively easy French-to-English pair [35], suggesting that translation from source code to pseudo-code is easier than translating between natural languages. This result can be expected, because SMT systems targeting natural language pairs always include the noise caused by ambiguities of tokenization or parsing for source natural language sentences, while our pseudo-code generator has no such input ambiguity.

In addition, we can see that BLEU scores for the Python-to-Japanese dataset are higher than the Python-to-English dataset. This can be attributed to the characteristics of each dataset. The Python-to-Japanese dataset's original source code was generated by one engineer from arithmetic problems for programming practice, so all the input code in the set shares a similar programming style. In contrast, the source code of the Python-to-English dataset is extracted from Django, which is developing many engineers for a variety of objectives, and thus there is larger variance. The results of *PBMT* and *Head-T2SMT* for Python-to-English reflect these characteristics. As we mentioned in §III-B, PBMT-based pseudo-code generators cannot adequately handle grammatically complicated sentences, likely the reason for their reduced accuracy. Each T2SMT system achieves significantly higher BLEU than the *PBMT* system, indicating that properly analyzing the program structure before generating comments is essential to accurately generating pseudo-code.

We can also note that the *Head-T2SMT* system has a lower score than *Raw-T2SMT* for the Python-to-English dataset. This result is likely caused by the variance of human-created pseudo-code and word alignment. The head insertion process introduces many new tokens into the syntax tree and some of these tokens have no information to express the relationship between programming and natural language, causing problems in automatic word alignment. However, this problem did not rise to the surface for the Python-to-Japanese dataset, because it has less variance in the data. The *Reduced-T2SMT* system achieves the highest BLEU score for all settings in both languages, indicating that this method can avoid this problem by reducing redundant structures in head-inserted syntax trees.

To evaluate statistical significance, we performed a pairwise bootstrap test [36] for 10,000 sets of evaluation sentences randomly extracted from these results. Based on this we obtained a statistical significance under $p < 0.001$ for all T2SMT systems against the *PBMT* system, and $p < 0.001$ for the *Reduced-T2SMT* system against all other systems.

We can also see in TABLE III that the acceptability scores of the pseudo-code for the Python-to-Japanese dataset is also improved in correlation to the BLEU improvement. Especially, *Reduced-T2SMT* achieves a mean acceptability over 4, which means that a large amount of the pseudo-code generated by the most advanced method *Reduced-T2SMT* was judged as grammatically correct by evaluators.

Fig. 11 shows the acceptability distribution of each system. We can see that all systems can generate grammatically correct and fluent pseudo-code for 50% of the statements. Further, we can see that each T2SMT-based system generates less pseudo-code with "intermediate acceptability." This is an interesting result, but it is intuitively understandable that T2SMT-based systems can generate accurate pseudo-code if their rule tables cover the input statements, because these systems explicitly use the grammatical information through syntax trees of the programming language.

Finally, TABLE IV shows the results of the code understanding experiment. In this table, we show 3 results calculated by different evaluator groups. The *Experienced* group includes 8 of 14 evaluators who have more than 1 year of experience

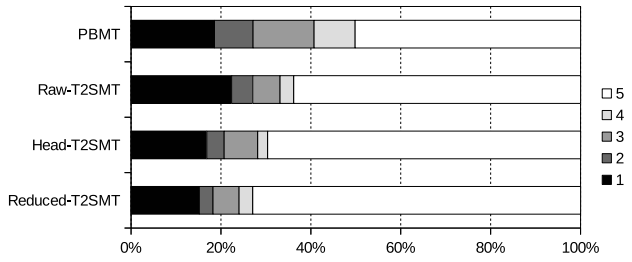

Fig. 11. Acceptability distribution of each system.

TABLE IV. MEAN UNDERSTANDABILITY IMPRESSIONS AND MEAN TIME TO UNDERSTAND.

Group	Setting	Mean Impression	Mean Time
Experienced	*Code*	2.55	41.37
	Reference	3.05	35.65
	Automated	2.71	46.48
Inexperienced	*Code*	1.32	24.99
	Reference	2.10	24.97
	Automated	1.81	39.52
All	*Code*	1.95	33.35
	Reference	2.60	30.54
	Automated	2.28	43.15

in programming Python, The *Inexperienced* group includes the remaining 6 evaluators (with less than 1 year of experience of Python), and *All* includes all evaluators. From the results, we can see that the result of the *Reference* setting achieves the highest understandability and the fastest reading time of all settings. This means that proposing correct pseudo-code improves the ease and efficiency of code reading when readers try to read the whole source code in detail. The result of the *Automated* setting also achieves a better impression than that of the *Code* setting. However, the reading time becomes longer than other settings. We assume that this is the result of the few strange lines of pseudo-code generated from our generator (e.g. pseudo-code scored 1 in acceptability) that confuse the readers in their attempt to interpret source code. Reducing generation errors in the *Automated* method can reduce this time-loss, in principle, similarly to the results of *Reference*.

Fig. 12 shows 3 sets of examples from the proposed methods. We can see that each T2SMT-based systems (especially *Reduced-T2SMT*) generates more accurate sentences in English than the *PBMT* system.

Python	`for node in graph.leaf_nodes(app_name):`
PBMT	for node in graph.leaf_nodes with an argument app_name,
Raw-T2SMT	for every node in, return value is the return value of the graph.leaf_nodes app_name,
Head-T2SMT	for every node in graph.leaf_nodes app_name,
Reduced-T2SMT	for every node in return value of the graph.leaf_nodes with an argument app_name,
Python	`if self._isdst(dt):`
PBMT	if self.call the method _isdst with 2 arguments dt, if it evaluates to true,
Raw-T2SMT	self._isdst with an argument dt, if it evaluates to true,
Head-T2SMT	if self._isdst with an argument dt, return the result.
Reduced-T2SMT	call the method self._isdst with an argument dt, if it evaluates to true,
Python	`context[self.var_name] = obj`
PBMT	self.call the method self.var_name context [] = obj.
Raw-T2SMT	call the method obj, substitute it for value under the ' key of the self.var_name key of the context dictionary.
Head-T2SMT	substitute obj for value under the self.var_name key of the context dictionary.
Reduced-T2SMT	substitute obj for the value under the self.var_name key of the context dictionary.

Fig. 12. Examples of generated pseudo-code from each system.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we proposed a method for pseudo-code generation that is, to our knowledge, the first of its type. Our method is based on statistical machine translation (SMT) techniques, especially phrase-based machine translation (PBMT) and tree-to-string machine translation (T2SMT), which allow us to automatically learn statement-wise pseudo-code generators and require less human effort to create and update our generators. Experiments showed that our proposed methods generate grammatically correct pseudo-code for the Python-to-English and Python-to-Japanese programming/natural language pairs, and distinguished that proposing pseudo-code with source code improves code comprehension of programmers for unfamiliar programming languages. To use SMT frameworks, we prepared a parallel corpus, which includes sentence (or syntax tree) pairs related to each other, and described several algorithms to adjust the parallel corpus to be in a format appropriate for SMT.

In the future work, we are planning to develop a pseudo-code generator uses the proposed SMT framework to handle multiple statements, close to the standard setting of automatic comment generation. To do so, we must find or create a high-quality parallel corpus of source code and comments corresponding to multiple statements in the source code, which is a less well-formed problem than creating line-to-line comments, and is thus an interesting challenge for future work. We also plan to investigate the use of automatic pseudo-code generation by experienced programmers in large software project environments. For example, automatically generated pseudo-code could be used by programmers to confirm that the code that they wrote is actually doing what they expect it to be doing, or to help confirm when existing (single-line) comments in the source code have gone stale and need to be updated.

ACKNOWLEDGMENT

Part of this work was supported by the Program for Advancing Strategic International Networks to Accelerate the Circulation of Talented Researchers. We also thank Mr. Akinori Ihara, who also gave us useful advice regarding this work.

REFERENCES

[1] R. DeLine, G. Venolia, and K. Rowan, "Software development with code maps," *Commun. ACM*, vol. 53, no. 8, pp. 48–54, 2010.

[2] M. M. Rahman and C. K. Roy, "Surfclipse: Context-aware meta search in the ide," in *Proc. ICSME*, 2014, pp. 617–620.

[3] M.-A. Storey, "Theories, tools and research methods in program comprehension: Past, present and future," *Software Quality Journal*, vol. 14, no. 3, pp. 187–208, 2006.

[4] 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.

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

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

[7] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Proc. ICPC*, 2013, pp. 23–32.

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

[9] 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.

[10] 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.

[11] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proc. ICSE*, 2014, pp. 390–401.

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

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

[14] 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.

[15] P. Koehn, F. J. Och, and D. Marcu, "Statistical phrase-based translation," in *Proc. NAACL-HLT*, 2003, pp. 48–54.

[16] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *Proc. Onward!*, 2014, pp. 173–184.

[17] F. J. Och and H. Ney, "The alignment template approach to statistical machine translation," *Computational Linguistics*, vol. 30, no. 4, pp. 417–449, 2004.

[18] 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.

[19] D. Klein and C. D. Manning, "Accurate unlexicalized parsing," in *Proc. ACL*, 2003, pp. 423–430.

[20] S. Petrov, L. Barrett, R. Thibaux, and D. Klein, "Learning accurate, compact, and interpretable tree annotation," in *Proceedings of COLING-ACL*, 2006, pp. 433–440.

[21] 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.

[22] 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.

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

[24] R. Kneser and H. Ney, "Improved backing-off for m-gram language modeling," in *Proc. ICASSP*, 1995, pp. 181–184.

[25] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. ICSE*, 2012, pp. 837–847.

[26] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proc. FSE*, 2013, pp. 532–542.

[27] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proc. FSE*, 2014, pp. 269–280.

[28] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proc. ICSE*, 2015.

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

[30] 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.

[31] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin, and E. Herbst, "Moses: Open source toolkit for statistical machine translation," in *Proc. ACL*, 2007, pp. 177–180.

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

[33] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proc. ACL*, 2002, pp. 311–318.

[34] I. Goto, K. P. Chow, B. Lu, E. Sumita, and B. K. Tsou, "Overview of the patent machine translation task at the ntcir-10 workshop," in *NTCIR-10*, 2013.

[35] O. Bojar, C. Buck, C. Federmann, B. Haddow, P. Koehn, J. Leveling, C. Monz, P. Pecina, M. Post, H. Saint-Amand, R. Soricut, L. Specia, and A. Tamchyna, "Findings of the 2014 workshop on statistical machine translation," in *Proc. WMT*, 2014, pp. 12–58.

[36] P. Koehn, "Statistical significance tests for machine translation evaluation," in *Proc. EMNLP*, 2004, pp. 388–395.