Encoding Beacon Statements for Code Comment Generation

Nan Jia (✉ 546690311@qq.com)
Hebei GEO University

Jie Chen
Guangzhou University of Traditional Chinese Medicine First Affiliated Hospital

Mingliang Li
Hebei GEO University

Research Article

Keywords: code comment, comment generation, transfer learning, code knowledge

Posted Date: May 5th, 2023

DOI: https://doi.org/10.21203/rs.3.rs-2710941/v1

License: ☇ This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License
Encoding Beacon Statements for Code Comment Generation

Nan Jia  
School of Information Engineering, Hebei GEO University, Shijiazhuang, China  
jianan_0101@163.com

Jie Chen  
The First Affiliated Hospital, Guangzhou University of Chinese Medicine, Guangzhou, China  
2287550@qq.com

Mingliang Li  
Intelligent Sensor Network Engineering Research Center of Hebei Province, Shijiazhuang, China  
417210204@qq.com

Abstract—High-quality code comment is important to help developer review and comprehend source code. Inspired by the effectiveness of deep learning techniques in the NLP field, many studies focus on using the machine translation algorithm to automatically generate comment for the source code. Most of the current studies typically involve in collecting a large number of dataset regarding the function-comment pairs, and then encode the information (e.g., AST) from the whole function for training a generation model. Most of current studies indistinguishably encode every statement in a function, which may not be the best strategy for comment generation because each statement in a function plays different roles, and the non-core code statements may have no effect or even a negative effect on the comment generation. In this paper, we propose a flexible encoding strategy that highlights the most informative statements from the function, called beacon statements, for code comment generation. Specifically, the beacon statements are detected from the function according to their roles when human understanding program, and then a pre-training model is proposed to encode the knowledge of the beacon statements. At last, the knowledge of the beacon statements is transferring to a basic model for comment generation by sharing the training parameters. The encouraging experimental results demonstrate the feasibility and effectiveness of our comment generation model.

Index Terms—code comment, comment generation, transfer learning, code knowledge

I. INTRODUCTION

Code comment is an important part of a software project that illustrates the logic and functional implementation of source code in a natural language form (Keyes, 2002; McBurney et al., 2014). Code comment has used as a standard practice in software development to increase the readability of source code (Vermeulen et al., 2000; Shido et al., 2019; “Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary”, 2020). Obviously, High-quality code comment plays an important role in code review and comprehension, which directly benefits most of software activities, such as software maintenance (Steidl et al., 2013; Storey et al., 2008; Huang et al., 2020).

Due to the importance of code comments, many studies focus on automatically generating comment for the source code to address the issues of comment mismatched, missing or outdated in many projects (Kajko-Mattsson, 2005; Hu, Li, Xia, Lo, & Jin, 2018). Essentially, code comment is the sentence written by natural language, and inspired by the effectiveness of deep learning techniques in the NLP field, previous studies (Iyer et al., 2016; Allamanis et al., 2016) have employed the machine translation algorithm to “translate” the programming language source code to natural language comment.

Most of the previous studies typically involve in collecting a large number of dataset regarding the function-comment pairs. For example, Hu et al. (Hu, Li, Xia, Lo, & Jin, 2018) collected 69,708 function-comment pairs from 9,714 open source projects, and 410,627 function-comment pairs from 22,886 projects in their another study (Hu, Li, Xia, Lo, Lu, & Jin, 2018). Then, the current studies encode the information (e.g., AST) from the whole function snippets for training a generation model (Alon et al., 2019). For example, Hu et al. (Hu, Li, Xia, Lo, & Jin, 2018) proposed to analyze the Abstract Syntax Trees (AST) of a function, and convert whole AST into sequences before they are fed into the machine translation model.

The existing studies treats the code statements in the function indistinguishable in the encoding phase, which may not be the best strategy for comment generation. Because each statement in a function snippet does not make an equal contribution to program comprehension for programmers (Storey, 2005), and the non-core code statements may have no effect or even a negative effect on the comment generation (Pascarella et al., 2019). According to the cognition process of the human understanding program, developers trend to use the statements with important information to comprehend the program (Crosby et al., 2002), we called it beacon statement in this paper. Beacon statements contains the key information for the comment generation.

Consider the function `encrypt` in Figure 1, which contains 16 statements. The 9th statement, which implements the feature of BASE64 encrypt, the other statements in the function handle exceptions, variable initializations and processing return value. Although all the 16 statements are necessary for execution, while 9th statement is indispensable for generating the code comment, i.e., `//BASE64 encrypt`, which is the beacon statement. Obviously, the beacon statement plays more important role in generating the code comment. Then, one of the major challenges in this study is to precisely identify the beacon statements from the source code of a function.
To address this challenge, we propose a heuristic way to identify the beacon statements according to our observations and the previous study (Sridhara et al., 2010). Specifically, we firstly propose the definitions of different beacon statements, and also describe the way that detects the beacon statements from the source code. After that, a pre-training model is proposed to encode the knowledge of the beacon statements. At last, the knowledge of the beacon statements is transferred to a basic generation model that used to generate the code comments. In the evaluation, the BLEU-4 score of the proposed method is 36.56, which outperforms the baselines and other methods.

The contributions of our work are as follows:

- We propose a heuristic way to identify the beacon statements from the source code by considering their roles when human understanding program.
- We propose a transfer learning based approach to generate the code comments with combining the knowledge of the beacon statements and the knowledge of the entire source code.
- We perform careful experiments with the comment dataset to evaluate the performance of the proposed approach. The experimental results show that the proposed approach outperforms the baselines and other methods in comment generation.

Paper organization. Section 2 presents the definitions of the beacon statements, while Section 3 describes the main method of comment generation, including the pre-training model and the basic comment generation model. The setups of experiment are discussed in Section 4. Section 5 and 6 present the experimental results and discussions, while Section 7 discusses the related works regarding the code comment generation. Section 8 discusses the threats that could affect the results of our study. Section 9 summarizes our approach and outlines directions of future work.

II. BEACON STATEMENTS

In this section, we will introduce the definitions of beacon statements in detail. Meanwhile, we also provide the ways that detect beacon statements from source code. Beacon statements contain key information for developers comprehending the source code, which play important roles in code comment generation, and we analyze and summary the roles of the beacon statements from the perspective of the human understanding program.

Function Declaration Statement The first type of beacon statement is the function declaration statement. Based on common Java naming conventions (Jiang et al., 2019; Liu et al., 2016), we observe that most function names start with a verb and end with a noun. The verb usually shows the action of the function, and the noun shows the theme of the action. For example, the function name compileCode() contains a verb compile and a noun code, which indicate the action and the theme of this function. Therefore, the function declaration statement provides a very rich amount of information for generating code comments (Allamanis et al., 2014). To identify the function declaration statement, we firstly employ the JavaParser tool to obtain the Abstract Syntax Tree (i.e., AST) for each function, and then regard the first statement in the AST as the function declaration statement.

Ending Statement Previous study (Sridhara et al., 2010) shows that a function often performs a series of operations to achieve a final action by the ending statement of the function, which is the main purpose of the function. We also observe this phenomenon across many functions, which leads to the selection of ending statement as a beacon statement.

To determine the ending statement of a function, we build a Control Flow Graph (i.e., CFG) (Ammarguellat, 1992) for each function. Because each return node can lead to the program exit, then we link all the return nodes to the exit node, which leads a CFG with a single-entry and single-exit nodes. Then, an ending statement is a predecessor of the exit node of a CFG.

Non-Return Statement Non-return statement is defined as: a statement involves function call, but it does not return a value or whose return value is not assigned to a variable. As Figure 1 shows, statement 11 is a Non-return statement. We observe that a function call without return value usually carries useful information for generating code comments. Because previous study (Sridhara et al., 2010) found that a function call does not return a value, it must be invoked purely for side effects, which can be used as auxiliary information for comment generation.

To identify the non-return statement from source code, we also employ the JavaParser tool to obtain the Abstract Syntax Tree (i.e., AST) for each function, and then we traverse the whole AST and identify the statements with function call. At last, we determine whether they have return values by analyzing the leaf nodes of the statements with function call.

Same-Action Statement In a function F, if a statement has a function call cf, then F and cf have the same action, and we call the statement as same-action statement. As Figure 1 shows, statement 9 calls function encryptBASE64(), which is neither an ending statement nor a non-return state-

1 https://javaparser.org/
Fig. 2. The overall framework of code comment generation

ment, and encryptBASE64() has the same action "encrypt" with function encrypt().

Obviously, the call on statement 9 is very important towards achieving the intended functionality of "encrypt". Therefore, the same-action statement is important to generate code comment. We develop the SWUM tool to identify the same-action statement based on the previous study of Hill, et al. (Hill et al., 2009). SWUM has the ability to identify linguistic elements of an arbitrary function name. Specifically, it is critical to identify the action and theme for a given function. For example, the action of function encryptBASE64() captured by SWUM is encrypt, the theme is BASE64, and the action of function encrypt() is also encrypt. As a result, it is easy to identify the same action via applying the SWUM tool on the function name analysis.

Data-Facilitating Statement Data-facilitating statement assigns data to variables used in the statements identified by the previous three heuristics. For example, consider the statement result = BASE64.encryptBASE64(strByte) on line 9 in Figure 1. This statement is selected by the same-action heuristic, but little information is known about the function parameter, i.e., strByte. Therefore, we select its data facilitator on line 5, i.e., Byte[] strByte = null.

We identify the function parameters in the function call by analyzing the AST. We firstly find the definition of a parameter in a function, and then identify the statements where the parameter is used. After that, we can find data-facilitating statement for these parameters using the def-use chains for the function. We go back one level from the use along the def-use chain.

Controlling Statement A controlling statement controls the execution flow of the program, such as if statement, while statement, and switch statement. Controlling statements often specify the conditions under which a program executes. Taking the controlling statement into the comment generation can conveys more information, specifically about when the major action occurs. We analyze the AST to identify the controlling statement. A controlling statement can be identified as controlling token, e.g., if statement is identified as IfStatement token in AST.

III. COMMENT GENERATION MODEL

A. Overall Framework

Figure 2 shows the overall framework of the proposed framework, which includes two parts: the pre-training model and the basic comment generation model (called BCGM). The pre-training model consists of a encoder and a decoder, which takes the beacon statements (i.e., BS in Figure 2) as input and has the ability to learn the knowledge of the beacon statements. The basic comment generation model also consists of a encoder and a decoder, which takes the AST of a function as input. To transfer the knowledge of the beacon statements to the basic comment generation model, we employ the transfer learning algorithm to fuse parameters of the pre-training model into the basic model.

Pre-training beacon statements We firstly identify the beacon statements from the source code by employing the method proposed in section 2. Then, for each beacon statement, we extract the identifiers from it. Meanwhile, we segment the identifiers according to Camel and Pascal rules (Beniamini et al., 2017) commonly used in Java program. After that, the identifiers coming from all the beacon statements form a text, and the word2vector technique (Mikolov et al., 2013) is used to encode the identifiers in the text.

After applying the word2vector technique, each identifier is represented by a vector, which can be used as the input of the pre-training model. Then, we use the pre-training model to learn the knowledge of the beacon statements. The encoder of the pre-training model inputs the vector of each identifier at each time step, and the gate mechanism controls the addition, deletion and update of historical information in code sequences. The output of the last time step is used as the intermediate input vector for decoder, and decoder calculates the context vector at each time step. At last, the cross-entropy between the generated comment and the real comment in the decoder is calculated as loss and the back-propagation
algorithm is used to update the optimization parameters to obtain the pre-training model that has the knowledge of the beacon statements.

**Encoding the source code of functions** The main idea of this study is to transfer the knowledge of the beacon statements to a basic comment generation model, hence we need to build a basic model that learn the knowledge of the source code of a function for comment generation. Therefore, we need to encode the source code of functions for building a generation model. We employ the method proposed by Huang et al. (Huang et al., 2020), and they proposed a statement-based AST traversal algorithm to generate the code token sequence with preserving the semantic, syntactic and structural information in the source code. After obtaining the code token sequence of each function, we can get a number of token sequence-comment pairs from the dataset, which can be used as the input for training a basic comment generation model.

**Transferring beacon statement knowledge** The comment generation model in this paper employs multiple encoders to realize knowledge transfer, and two encoders are used to encode the beacon statements and the whole source code of a function, respectively. The knowledge of beacon statements is transferred by means of sharing parameter, and the parameters of the encoder that contains the knowledge of beacon statements in the pre-training model are shared with the basic comment generation model.

In the decoding stage, we use the decoder adding attention mechanism to decode the intermediate vector containing all the knowledge to produce output, and calculate the cross-information entropy of the generated comments and the real comments, and the model parameters are optimized by BP algorithm and back-propagation algorithm.

**B. Pre-training Model**

After identifying the beacon statements from functions, we need extract the implicit knowledge from them for comment generation. We build a pre-training model to extract and preserve the implicit knowledge. The pre-training model is based on the dual-encoder seq2seq model with adding attention mechanism (Iyer et al., 2016).

Figure 3 shows the structure of the pre-training model. Suppose we collect a dataset \( W \) that contains beacon statements from \( i \) functions, i.e., \( W = \{W^i\} \). \( W^i = \{w_1, ..., w_i\} \) represents the beacon statements extracted from a function. For each \( \{W^i\} \in W \), there is a corresponding code comment \( D^i = \{d_1, ..., d_m\} \), and \( d_n \) is the word in comment \( D^i \). \( D \) is the set of \( D^i \), and \( D^i \in D \). The goal of pre-training model is to build the mappings between \( W \) and \( D \), i.e., \( W \rightarrow D \).

In order to explore potential mappings between code and comments, we use an attention mechanism, and the hidden state in the node unit of each time step in the encoder is calculated by the input of the current time step and the hidden state of the previous time step:

\[
h_{\text{encoder}_{w,t}} = GRU_{\text{encoder}_w}(w_t, h_{\text{encoder}_{w,t-1}}) \quad (1)
\]

The decoder computes the conditional probability of predicting the current word given the context vector \( c_t \) and the previous predicted word:

\[
p(d_{t'}|d_1, ..., d_{t'-1}, w) = g(d_{t'-1}, h_{\text{encoder}_{w,t'}}, c_t) \quad (2)
\]

\( h_{\text{encoder}_{w,t'}} \) refers to the hidden state of the decoder in time step \( t' \), and its formula is:

\[
h_{\text{decoder}_{w,t'}} = GRU_{\text{decoder}_w}(d_{t'-1}, h_{\text{decoder}_{w,t'-1}}, c_t) \quad (3)
\]

The context vector \( c_i \) is the weighted average value of all hidden states in the encoder, and the formula is:

\[
c_i = \sum_{j=1}^{n} a_{i,j} h_{\text{encoder}_{w,j}} \quad (4)
\]

The weight calculation formula is:

\[
a_{i,j} = \frac{\exp(e_{i,j})}{\sum_{k=1}^{n} \exp(e_{i,k})} \quad (5)
\]

\( e_{i,j} \) is an aligned model, which is used to calculate the match degree between the hidden state \( h_{\text{encoder}_{w,i}} \) of encoder and the hidden state \( h_{\text{decoder}_{w,j}} \) of decoder.

\[
e_{i,j} = \text{score}(h_{\text{encoder}_{w,i}}, h_{\text{decoder}_{w,j}}) \quad (6)
\]

At each time step, the output of the decoder is connected to the softmax layer, and gets the probabilities of all the words, including the probabilities \( p_{t,y_t} \) of the words in the code comment. After that, calculating the cross entropy as the loss function:

\[
p_t = \text{softmax}(U \times h_{\text{decoder}_{w,t}}) \quad (7)
\]

\[
\ell = \sum_{t=1}^{m} \log(p_{t,y_t}) \quad (8)
\]

\( U \) are the parameters, and \( \ell \) is the loss function. Then, the back propagation is carried out by iteration and updating the parameters in the model with the gradient descent method. Algorithm 1 shows the pseudocode of the training process of the pre-training model.

**C. Basic Comment Generation Model**

The basic comment generation model shares the parameters coming from the pre-training model. Meanwhile, the basic comment generation model employs a pair of encoder and decoder to learn the knowledge of source code from the AST of the function. The basic model combines the knowledge of beacon statements and the AST of the function to generate the code comment.

1) **Encoder**: Suppose \( h_{\text{encoder}_{w,t}} \) and \( h_{\text{encoder}_{w,t-1}} \) represent the hidden state of each encoder at the current time step, as Figure 4 shows, then their values are determined by the hidden
state of the current input and the historical information. The calculation formulas are as follows:

$$h_{encoder_{s,t}} = f(s_t, h_{encoder_{s,t-1}})$$ (9)

$$h_{encoder_{w,t}} = f(w_t, h_{encoder_{w,t-1}})$$ (10)

where $f$ refers to the nonlinear function, the encoder is used to encode AST knowledge and the beacon statement knowledge in formulas 9 and 10 is implemented by the sequence encoder LSTM (Hochreiter & Schmidhuber, 1997).

2) **Decoder**: To achieve better performance of comment generation, the decoder of BCGM adds attention mechanism to automatically mine the implicit relationship between the current output and all hidden states in the encoder. The encoder finally obtains two vectors, which are AST sequence vector $h_{encoder_{s,t}}$ and beacon statement vector $h_{encoder_{w,m}}$, and the decoder splices them into their initial hidden state. The formula is as follows:

$$h_{encoder,0} = h_{encoder_{s,t}} \oplus h_{encoder_{w,m}}$$ (11)
Algorithm 1: Pre-training Model

Input: \( D \): the set of beacon statements; \( \text{size} \): patch size; 
Initialize the model parameters: \( \text{GRU}_{\text{encoder}} \), \( \text{GRU}_{\text{decoder}}, W \) 
batches \( \rightarrow \text{set\_batches}(D, \text{size}) \) /build batches for training set 

\[
1: \text{while True do:} \\
2: \text{batches \( = \text{next\_batches() \text{/get data for the next batch} \) /} \\
3: \ell \rightarrow \text{init\_loss() /} \text{initialize error} \) \\
4: \text{for(} w_1, w_2, ..., w_n; d_1, d_2, ..., d_2 \text{) in batches do} \\
5: (x_1, x_2, ..., x_n) \rightarrow \{ w_1, w_2, ..., w_n \} \\
6: (y_1, y_2, ..., y_m) \rightarrow \{ d_1, d_2, ..., d_2 \} \\
7: \text{for} \ k \in (1, n) \text{ do /} \text{Encoder computes hidden vectors} \\
8: h_{\text{encoder}_{w,k}} \rightarrow \text{GRU}_{\text{encoder}}(x_k, h_{\text{encoder}_{w,k-1}}) \\
9: \text{end for} \\
10: h_{\text{decoder}} \text{ is intermediate vector} \\
11: h_{\text{encoder}_w,0} = h_{\text{encoder}_w,n} \\
12: \text{for} \ t \in (1, m) \text{ do} \\
13: c_t = \sum_{j=1}^{m} a_{tj} h_{\text{encoder}_{w,j}} \\
14: h_{\text{decoder}_{w,t}} = \text{GRU}_{\text{decoder}}(y_t-1, h_{\text{encoder}_{w,t-1}}, c_t) \\
15: p = \text{softmax}(W \times h_t) \\
16: t+ = -\log(p_{y_t}) \\
17: \text{end for} \\
18: \text{end for} \\
19: \ell.\text{backward() /} \text{back propagating for update gradient} \\
20: \text{end while} \\
\]

\[
h_{\text{encoder}_{,t}} = f(d_{t-1}, h_{\text{encoder}_{,t-1}}, A_t)
\]

(12)

where \( A_t \) is the context vector at time \( t \) in the attention mechanism, and its formula is as follows:

\[
A_t = \sum_{i=1}^{1} a_{t1} h_{\text{encoder}_{,i}} + \sum_{j=1}^{m} a_{tj} h_{\text{encoder}_{,j}}
\]

(13)

\( a_{t1} \) refers to the correlation coefficient between the hidden vector \( h_{\text{decoder}_{,t-1}} \) in the decoder time step \( t-1 \) and the hidden state \( h_{\text{encoder}_{,t}} \) in the encoder unit \( i \). After the decoder obtains the current hidden state, it will be mapped to the word table, and uses softmax function to do normalization calculation, and get the probability distribution function \( p(y_t) \) for each time step \( t \):

\[
p(y_t) = \text{softmax}(U \times h_{\text{decoder}_{,t}} + b)
\]

(14)

\( U \) and \( b \) are the parameters of the softmax function, and then the cross-information entropy between real tags is calculated as the objective function for optimization:

\[
\ell = -\sum_{t=1}^{M} \sum_{r=1}^{r} q(y_t) \log(p(y_t))
\]

(15)

where \( q(y_t) \) is the probability distribution of \( y_t \), and \( M \) is the batch size, and \( r \) is the target sequence length. In this way, the decoder of the basic model will decode the intermediate vectors that combine the source code knowledge with the beacon statement knowledge into machine-generated code comments. Algorithm 2 shows the pseudocode of training the BCGM model.

IV. Evaluation

In this section, we conduct comprehensive experiments to evaluate the effectiveness of the proposed model, and we aim at answering the following research questions:

- **RQ1**: What is the performance of only applying the beacon statements to generate code comments when comparing with randomly selected statements?
- **RQ2**: What is the accuracy of the comment generation when transferring the knowledge of beacon statements to the generation model?
- **RQ3**: What is the performance of comment generation model when comparing with existing comment generation models?

A. Dataset

We uses the public dataset provided by Hu et al. (Hu, Li, Xia, Lo, Lu, & Jin, 2018) for the code comment generation experiment. The dataset was collected from JAVA projects created on Github between 2009 and 2016. The function comments are extracted through the Eclipse JDT compiler, which is an integrated development environment widely used in Java development. Specifically, the functions in the source code are extracted and parsed into abstract syntax tree, and the text in Javadoc node is selected as the comment.

Table I shows the details the selected comment by Hu et al. They 13,154 Java projects from Github using the search API \(^2\) during 2009 to 2014, and 9,732 projects during 2015 to 2016. We divide these data sets into three parts, namely, training set, verification set, and test set.

The second dataset (collected during 2015 to 2016) is used to train the pre-training model, while the other dataset (collected during 2009 to 2014) is used for the training of the basic comment generation model. We use the previous dataset to get the shared parameters of the pre-training model, and then transfer them to the BCGM model.

\(^2\)https://developer.github.com/v3/search/
Tables II to IV illustrate statistics of the data set. Among them, Table II is the statistics for comment lengths. As the table shows, the lengths of the comments are relatively small. Their average length is 17 words, and 86% of comments contain no more than 30 words. From the statistics for code lengths in Table III, we know that the average length of the functions is 106 after word splitting including splitting for camel case. Meanwhile, 87 percent of the functions are less than 200 words in length. After extracting beacon statements from functions, we also make statistics on them, as shown in Table IV. The average length of the token sequences is 60, and more than 93% of them are less than 150 words.

### Table I
Details of the four open source projects

<table>
<thead>
<tr>
<th>Time</th>
<th>Projects</th>
<th>Training Set</th>
<th>Validation Set</th>
<th>Test set</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-2014</td>
<td>13,154</td>
<td>272,737</td>
<td>34,092</td>
<td>34,092</td>
</tr>
<tr>
<td>2015-2016</td>
<td>9,732</td>
<td>55,766</td>
<td>6,970</td>
<td>6,970</td>
</tr>
</tbody>
</table>

B. Evaluation Criteria

In order to evaluate the effectiveness of the proposed model, BLEU (Thabet et al., 1991) and Rouge (Lin, 2004) are computed, respectively. BLEU is one of the machine translation evaluation metrics, which is used to measure the quality of the generated comment. BLEU (Bilingual Evaluation Understudy) measures the similarity between a generated comment and a reference comment. The higher the BLEU score is, the more similar the generated comment is to the reference comment, and the better the model performs. BLEU uses n-gram for matching and calculates the ratio of N groups of word similarity between generated comments and reference comments. The formula of BLEU is as following:

\[ BLEU = BP \cdot \exp(\sum_{n=1}^{m} w_n \log p_n) \]  \hspace{1cm} (16)

where \( p_n \) is the ratio of the subsequence of the generated comment with length \( n \) to the reference comment. As the value of \( n \) increases (the maximum value of \( n \) is 4), the BLEU score decreases exponentially. \( BP \) is the length penalty factor, and its formula is as follows:

\[ BP = \begin{cases} 1, & \text{if } c > r \\ e^{(1-r/c)}, & \text{if } c \leq r \end{cases} \]  \hspace{1cm} (17)

where, \( c \) represents the length of the generated comment, and \( r \) represents the length of the reference comment.

Rouge(i.e., Retractor-Oriented Understudy for Gisting Evaluation) is an evaluation mechanism used to evaluate the performance of the text summary and machine translation proposed by Chin-Yew Lin in 2004. Unlike BLEU, Rouge focuses more on recall rates, while BLEU focuses more on precision. Rouge calculates the recall by measuring the longest common sequence between a generated comment and a reference comment. The formula of Rouge is as following:

\[ R_{lcs} = \frac{\text{LCS}(X, Y)}{m} \]  \hspace{1cm} (18)

\[ P_{lcs} = \frac{\text{LCS}(X, Y)}{n} \]  \hspace{1cm} (19)

\[ F_{lsc} = \frac{(1 + \beta^2) R_{lcs} P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}} \]  \hspace{1cm} (20)

where \( \text{LCS}(X, Y) \) stands for the length between the generated comment and the reference comment. \( m \) and \( n \) are the length of the generated comment and the reference comment, that is, the number of contained words. \( R_{lcs} \) and \( P_{lcs} \) represent the recall and precision, respectively. \( \beta \) is used to balance the weight between recall and precision. In order to pay more attention to recall, \( \beta \) is usually set to a larger number.

In this study, the experiment environment is as follows:

- Software environment: Ubuntu 18.04 LTS (64bit) operating system, Python3, tensorflow-gpu.
- Hardware environment: Intel(R) Core(TM), i9-10900x CPU, 3.70GHz×20, Memory 256G, Storage capacity 4T, and GeForce RTX 2080 Ti×2.

V. Results Analysis

A. RQ1: Comparing with randomly selected rule

For RQ1, we compare the performance of comment generation when applying the beacon statements and the randomly selected statements, respectively. Specifically, we have used a heuristic rules to select beacon statements for each function. To have a fair comparison, we need randomly select the same number of code statements from a function. For example, if we
select 5 beacon statements from a function, then we randomly select 5 statements for comment generation from this function.

The dataset used in this experiment is the second one (collected from 2015 to 2016), i.e., 55,766 function-comment pairs are used for training a generation model, and 6,970 function-comment pairs are used for validation set, and another 6,970 function-comment pairs are used for test set. To show the effect of the beacon statements for code generation comments, we directly employ the pre-training model in Figure 3 to generate the code comment. When building the model, 50 epochs were set for training, and the learning rate was set as 0.5, and the learning attenuation rate was set as 0.95, and the batch size was 32.

Table V shows a summary of the BLEU and Rouge on the dataset. We observe that the performances of applying beacon statements over 1-gram to 4-gram are better than that of the random selection rule, and the BLEU4 of applying beacon statements is 33.90, which is significant better than the one of the random selection rule, i.e., 27.42. In addition, the Rouge value achieved by the beacon statements is obviously superior to the one achieved by the random selection rule.

In summary, the performance of applying the beacon statements is better than that of applying the randomly selected statements, which indicates that the beacon statements play more important role in the comment generation task when comparing with the randomly selected statements. The words containing in the beacon statements are more informative than the ones in the randomly selected statements for the comment generation task.

B. RQ2: The performance of transfer learning

To applying the knowledge of the beacon statements to the basic comment generation model, we employ a transfer learning model described in Figure 4. The basic comment generation model shares the parameters from the pre-training model for the code comment generation task.

Table VI shows the shared parameters by two models. Parameters 1 to 4 are the kernel and bias for the attention mechanism, and parameters 5 to 8 are the weights and offsets for the decoder, and parameter 9 is the dimensions for word embedding, and parameters 10 to 14 are the weights and offsets for the encoder. The dimensions of all the parameters can be found in Table VI.

To evaluate the effectiveness of transferring the knowledge of the beacon statements to the basic model, we design a comparative method that transfers the knowledge of the randomly selected statements of a function to the basic comment generation model, i.e., the randomly selected statements of the commits in RQ1 are used in this experiment for knowledge transferring. In addition, we also design another comparative method that transfers the knowledge of all statements of a function to the basic comment generation model. Specifically, we firstly extract identifiers of all the statements of a function and employ the word2vector to encode each identifier, and then employ the pre-training model to encode the knowledge of the all statements of a function. After that, we transfer the knowledge of the whole statements to the basic comment generation model.

Table VII shows the evaluation results. We can observe that the way of transferring all statements achieves better performance when comparing with the way of transferring the randomly selected statements, which indicates that transferring the knowledge of the whole statements to the basic comment generation model is more effective than that of the random-selection one. This is because the randomly selected statements may not be representative, and by themselves do not explain what the function is trying to implement. On the other hand, transferring the knowledge of the whole statements to the generation model is more effective, because the all code statement joined together is syntactically and semantically coherent, although some unimportant code statements may interfere comment generation.

Meanwhile, we observe from Table VII that transferring the knowledge of the beacon statements to the basic comment generation model is more effective than the ones of other two ways. The BLEU and Rouge achieved by transferring the
knowledge of the beacon statements are better than the ones of the other two ways. This is because the basic comment generation model selectively pick the most important code statements from a function, and these statements contain enough information to generate comments without introducing statements that would interfere with comment generation.

In summary, with the help of the knowledge of the beacon statements, the basic comment generation model achieves the best performance in the comment generation task when comparing with the ones of transferring the knowledge of the random-selection statements and the whole statements.

C. RQ3: Comparing with other models

In recent studies, many researches have proposed the comment generation models, and we select another two models for comparison, i.e., CodeNN (Iyer et al., 2016) and Seq2Seq (Sutskever et al., 2014). We want to investigate whether the proposed model can still perform well when comparing with other ones. CodeNN generates code comments using the LSTM network that adds the attention mechanism. Unlike other methods, which place the attention mechanism on hidden vectors in the encoder, and adds input for each time step in the encoder in its attention mechanism. Meanwhile, we also employ the Seq2seq model to generate the code comments, which is also based on attention mechanism and takes the AST sequence of a function as input for training.

Table VIII shows the BLEU and Rouge for the competing approaches on the test set. It can be seen that the BLEU of BCGM is 36.56, better than those of the rest of the approaches. It is a little surprised that Seq2Seq has the better results than CodeNN. The reason may also be that CodeNN only uses the original code sequence information, which does not take into account the structure of the entire code. Different from the CodeNN, Seq2Seq takes the and takes the AST sequence of a function as input, which considers the code structure of a function. Meanwhile, Seq2Seq is also based on attention mechanism, which further improve the accuracy of the comment generation.

In summary, the performance of the three models differ significantly on the task of header comment generation. BCGM applying the knowledge of the beacon statements to the generation model achieves the best performance, which indicates that beacon statements play important role in the comment generation, and some useful information for comment generation can be obtained by encoding the beacon statements.

VI. DISCUSSION

Figure 5 illustrates an example of a function, which contains 22 code statements and the beacon statements are marked in red boxes. In general, the function implements the feature of "exports component PDA to svg file".

![Fig. 5. Function exportToSVG()](image)

Table IX illustrates four comments generated by the competing methods. The comments generated by our model are similar to the original comments, and our method basically restores the meaning of the original comment. i.e., the original comment for the code snippet is: "export plan component PDA to svg file ". The comment generated by our model is: “export plan component to svg file”, CodeNN and Seq2Seq can also predict some important words for the generation comment, such as: “export”, “plan”, “SVG”, “component”, while the overall meaning of the comments generated by CodeNN and Seq2Seq are slightly different from the original comment. We also observe that some words are replaced by their another word. For example, the word “component” is replaced by “object” in the comment generated by CodeNN. This substitution makes the meaning of the comment no longer accurate. The result shows that our model can capture the
meaning of a function, and the other methods cannot learn effective information in the source code.

TABLE IX
GENERATED COMMENTS

<table>
<thead>
<tr>
<th>No.</th>
<th>Methods</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Original</td>
<td>export plan component PDA to svg file</td>
</tr>
<tr>
<td>2</td>
<td>CodeNN</td>
<td>export objects to a given SVG</td>
</tr>
<tr>
<td>3</td>
<td>Seq2Seq</td>
<td>sets up plan component</td>
</tr>
<tr>
<td>4</td>
<td>Ours</td>
<td>export plan component to svg file</td>
</tr>
</tbody>
</table>

VII. RELATED WORK

In recent years, with the development of information retrieval technology (Manning et al., 2008) and deep learning (LeCun et al., 2015), automatic comment generation has drawn a lot of attention. There are three kinds of methods to automatically generate code comments: template filling based, information retrieval based, and deep learning based.

The template filling based methods (Sridhara et al., 2010; Moreno et al., 2013; Haiduc et al., 2010) firstly extract the keywords from the source code via static analysis techniques, then define a number of comment templates via the predefined heuristic rules. At last, the template filling based methods fill the keywords into the selected template, and generates the code comment. The template filling based approaches are also applied in areas such as release note generation (Moreno et al., 2014, 2016) and commit comment generation (Cortés-Coy et al., 2014; Buse & Weimer, 2010; Huang et al., 2017). Due to the limited templates, most of these approaches can only mechanically outline a superficial level summary of the source code, and the real intent behind the code is lost.

The information retrieval based methods generate code comments by searching the similar code snippets from the code base. For example, Wong et al. (Wong et al., 2013) proposed a novel method to automatically generate code comments by mining large-scale Q&A data from StackOverflow. Meanwhile, Wong et al. (Wong et al., 2015) applied code clone detection techniques to discover similar code segments in software repositories and used existing comments to describe similar code segments. Although the generated comments are close to those written by programmers, it has two major disadvantages: 1) if the code base is small and there are less reusable comments; 2) if there is a large code base, it is inefficient due to the code retrieval techniques.

The deep learning based methods use deep learning to train probabilistic models for the code comment generation. Iyer et al. (Iyer et al., 2016) used LSTM model to design a code comment automatic generation method for C# code and SQL queries. Allamanis et al. (Allamanis et al., 2016) used convolutional neural network to summarize the Java code into short, name-like summaries (average 3 words). Hu et al. (Hu, Li, Xia, Lo, & Jin, 2018) took structural and semantic information from the abstract syntax tree, converted it into sequence information, and then used the machine translation model to translate the code into comments. Since then, Hu et al. (Hu, Li, Xia, Lo, Lu, & Jin, 2018) took code API information into consideration, which further improves the accuracy of code comment generation. Different from the above methods, we consider exploiting the beacon statements to assist the comment generation, and propose a pre-training model is used to learn the knowledge of the beacon statements.

VIII. THREATS TO VALIDITY

In this section we focus on the threats that could affect the results of our case studies.

Threats to internal validity relates to the scale of the dataset using for training the comment scope detection model. Since we need to train a model to generate the code comment, it requires that we need to collect a large number of function-comment pairs to train the learning based model. However, collecting such a dataset is tedious and time-consuming, hence we directly use the dataset collected by Hu et al. (Hu, Li, Xia, Lo, Lu, & Jin, 2018), which contains more than four hundred thousand function-comment pairs. This dataset is large enough to train the comment generation model, and we believe there is little threat to the scale of the data set.

Threats to external validity relates to the generalizability of our results. Our approach is used to generate the comments for functions. All of the functions are written by Java language. When applying our approach to the projects written by other programming languages, such as C, C++, Python, the abstract syntax tree of the source code should be carefully handled because different languages have different ways for AST extraction. In the future, further investigation by analyzing even more projects written by other programming languages is needed to mitigate this threat.

Threats to construct validity refers to the suitability of our evaluation measure. We use a conventional measure to evaluate the effectiveness of the proposed method in this paper.
Because the issue in this study can be modeled as a natural language generation problem, we introduce the BLEU and Rouge to evaluate the performance of the proposed approach. Meanwhile, we compare the performances of the proposed approach with different baselines. All these metrics can evaluate the effectiveness of the proposed approach. Thus, we believe there is little threat to the suitability of our evaluation measure.

IX. CONCLUSION

Code comments play an important role in program comprehension and automatic comment generation is a goal pursued by software developers. This paper proposes a novel method, BCGM, automatically generating comment for the source code. To identify the code statements that are useful for the comment generation, we introduce the definitions of beacon statements and propose the ways of detecting them from the source code. A pre-training model is used to learn the knowledge of the beacon statements. To take advantage of the knowledge of the beacon statements, we transfer them to a basic comment generation model by sharing the training parameters. The BLEU-4 score of the proposed method is 36.56 and the experimental results demonstrate the feasibility and effectiveness of the proposed model. In the future, we will further consider to apply our comment generation approach to IDE.

DECLARATIONS

Data availability The data that support the findings of this study are available from the corresponding author upon reasonable request.

Compliance with Ethical Standards

Conflict of interest Author Nan Jia declares that she has no conflict of interest. Author Jie Chen declares that she has no conflict of interest. Author Mingliang Li declares that he has no conflict of interest.

Ethical approval: This article does not contain any studies with human participants or animals performed by any of the authors.

REFERENCES

Crosby, M. E., Scholtz, J., & Wiedenbeck, S. (2002). The roles beacons play in comprehension for novice and expert programmers. In Programmers, 14th workshop of the psychology of programming interest group, bruel university (pp. 18–21).


