SeTransformer: A Transformer-Based Code Semantic Parser for Code Comment Generation

Zheng Li, Yonghao Wu, Bin Peng, Xiang Chen, *Member, IEEE*, Zeyu Sun, Yong Liu, *Member, IEEE*, and Doyle Paul

***Abstract*—Automated code comment generation technologies can help developers understand code intent, which can significantly reduce the cost of software maintenance and revision. The latest studies in this field mainly depend on deep neural networks, such as convolutional neural networks and recurrent neural network. However, these methods may not generate high-quality and readable code comments due to the long-term dependence problem, which means that the code blocks used to summarize information are far from each other. Owing to the long-term dependence problem, these methods forget the previous input data's feature information during the training process. In this article, to solve the long-term dependence problem and extract both the text and structure information from the program code, we propose a novel improved-Transformer-based comment generation method, named SeTransformer. Specifically, the SeTransformer utilizes the code tokens and an abstract syntax tree (AST) of programs to extract information as the inputs, and then, it leverages the self-attention mechanism to analyze the text and structural features of code simultaneously. Experimental results based on public corpus gathered from large-scale open-source projects show that our method can significantly outperform five state-of-the-art baselines (such as Hybrid-DeepCom and AST-attendgru). Furthermore, we also conduct a questionnaire survey for developers, and the results show that the SeTransformer can generate higher quality comments than those of other baselines.**

***Index Terms*—Code comment generation, convolutional neural network (CNN), deep learning, program comprehension, Transformer.**

Manuscript received February 9, 2021; revised July 21, 2021 and December 28, 2021; accepted February 22, 2022. This work was supported in part by the National Natural Science Foundation of China under Grant 61902015 and Grant 61872026, in part by the Nantong Application Research Plan under Grant JC2019106, and in part by the High-Performance Computing Platform of the Beijing University of Chemical Technology. Associate Editor: B. Xu. *(Corresponding author: Yong Liu.)*

Zheng Li, Yonghao Wu, Bin Peng, and Yong Liu are with the College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100013, China (e-mail: lizheng@mail.buct.edu.cn; appmlk@outlook.com; 1252031372@qq.com; lyong@mail.buct.edu.cn).

Xiang Chen is with the School of Information Science and Technology, Nantong University, Nantong 226007, China (e-mail: xchencs@ntu.edu.cn).

Zeyu Sun is with the School of Electronic Engineering and Computer Science, Peking University, Beijing 100871, China (e-mail: szy_@pku.edu.cn).

Doyle Paul is with the School of Computer Science, Technological University Dublin, D07 EWV4 Dublin, Ireland (e-mail: paul.doyle@tudublin.ie).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TR.2022.3154773.

Digital Object Identifier 10.1109/TR.2022.3154773

I. Introduction

During software development and maintenance, developers often spend much of their time on program comprehension. Previous studies have shown that high-quality code comments can effectively improve a programs' comprehensibility because the developers can understand the program by just reading natural-language-based comments [1]. Therefore, researchers have proposed different automated techniques to help developers generate code comments for the target programs and, thus, substantially reduce the effort required for software maintenance.

These methods were designed to generate code comments at the class [2] or function level [1], [3]. Initial methods were usually designed based on handcrafting or information retrieval techniques. However, with the advancement of deep learning models and the sharing of many corpora gathered from open-source projects, researchers gradually turned their attention to deep learning and proposed some neural-network-based code comment generation methods. Most of these methods chose sequence-to-sequence methods for neural network training. For example, Iyer *et al.* [4] trained an embedding matrix to represent each code token in their work and coupled them with a recurrent neural network (RNN) model through an attention mechanism to generate code comments. Hu *et al.* [5] proposed a new structure-based traversal (SBT) method to serialize abstract syntax trees (ASTs). Then, they proposed a method of automatically generating code comments based on the RNN.

However, the RNN-based method has limitations in code comment generation. Some source codes may be very long, and the RNN-based model may not effectively capture the long-term dependence between the code tokens due to the long-term dependence problem. Thus, the model cannot recognize the relationship between the code contexts during the training process, which results in low-quality comments. In contrast to the RNN, the Transformer model leverages the self-attention mechanism to capture a wide range of dependence between texts [6]. Thus, the Transformer has shown the best performance in the research field of natural language processing. Moreover, to better utilize the advantages of neural network technology, we propose a new Transformer-based method named SeTransformer (syntax-based Transformer). In particular, the SeTransformer improves the traditional Transformer model to simultaneously process the textual information and structural information of software code. Finally, to speed up network training, we use convolutional

neural networks (CNNs) to compress the dimensionality of the data after embedding tokens in the code and its AST before the model training.

To verify the effectiveness of our proposed method, we conduct the experiments on a corpus constructed from large-scale open-source Java projects, which contain 87 136 Java functions and the corresponding comments. The Javadoc's first sentence is extracted as a comment, and the comments are used to explain the design purpose and the functionality of functions. Later, we split this dataset into training sets, validation sets, and test sets by 8:1:1 in our study. To evaluate the performance of the SeTransformer, we leverage two performance metrics (i.e., $BLEU$ [7] and $METEOR$ [8]) from the neural machine translation (NMT) domain. The results show that the SeTransformer can perform significantly better than five state-of-the-art baselines. Moreover, the CNN layer used in the SeTransformer can effectively reduce the training time without losing accuracy.

To the best of our knowledge, the main contributions of our study can be summarized as follows:

1) We employ an advanced Transformer model and propose a novel automated code comment generation method SeTransformer, which can simultaneously process the content and structural information of program code and generate high-quality comments for programs.
2) We propose a dimensionality reduction method for text sequences, which can reduce the model's training cost without significantly decreasing the model performance.
3) We conduct empirical studies on large-scale open-source projects, and the results show that the SeTransformer can significantly outperform other five state-of-the-art baselines.
4) To facilitate the replication of our study and evaluation of future code comment generation techniques, we share our source code and corpus in the GitHub repository.[1]

The rest of this article is organized as follows. Section II presents the background of our work. Section III introduces the framework and details of our proposed method SeTransformer. Section IV shows the experimental setup and result analysis. Section VI discusses the strength of the SeTransformer. Section VII discusses the threats to the validity of our study. Section VIII surveys the related studies and emphasizes the novelty of our study. Finally, Section IX concludes this article.

II. Background

A. Languages Models

Our work is based on NMT tasks in the field of natural language processing. The language models used in these tasks can determine word probability and, as a result, can generate whole sentences [5]. Similar to the language models, our proposed model SeTransformer aims to generate the code comments by estimating each word's probability.

For a sentence $x = (x_1, x_2, \ldots, x_n)$, the probability of the sentence is calculated from the likelihood of each word generated from the past sentence, i.e.,

$$P(x) = P(x_1)P(x_2|x_1)\cdots P(x_n|x_n - 1, \ldots, x_2, x_1). \quad (1)$$

Based on the above formula, various neural-network-based models were proposed to handle natural language processing problems. Recently, RNN [9], [10], Transformer [6], and CNN [11], [12] are three popularly used ones.

1) Recurrent Neural Networks: The structure of the RNN is a chain, and its input data is a sequence. It can use the entire history of previous input to guide each output. Therefore, it is closely related to sequence and list, and it has unique advantages for sequence information with time relationships.

However, the standard RNN model may suffer from the gradient explosion and disappearance problems during the training process [13], making the RNN unable to capture the long-term dependence between the code tokens effectively; thus, the network model is unable to train. Researchers proposed variants of RNNs (such as long short-term memory [14] and gated recurrent unit (GRU) [10]) and better neural network models (such as Transformer [6]) to solve this problem.

2) Transformer: The Transformer is an encoder–decoder model that only relies on attention for computing the contextual representations for source and target sentences. It avoids recurring model architectures in the RNN, which can avoid the disadvantages of the RNN mentioned in Section II-A1.

Since the Transformer [6] is a novel language model and superior to the RNN language model in terms of performance and time consumption on model training in the natural language processing field [9], [10], we use a Transformer-based deep learning language model to solve code comment generation problems in this article.

The Transformer model includes many components, the most critical component of which is multihead attention. Multihead attention is used to learn the relationship between each word in a sentence. The model SeTransformer proposed in this article includes all the standard Transformer components so that we will introduce them in detail in Section III-C.

3) Convolutional Neural Networks: In previous practice on code comment generation, the target code may contain many program statements. Thus, the dimensionality of the input data is accordingly tremendous. Owing to a large number of neural network connections, there will be enormous parameters to be trained. However, a considerable amount of calculation will lead to huge time cost. Therefore, it is necessary to leverage the CNN to compress the input data's dimensionality, thus reducing the computational complexity.

As shown in Fig. 1, the CNN architecture consists of five components: input, convolution layers, pooling layers, fully connected layers, and output. These components can extract meaningful information from input data while ignoring the noise.

In practice, a convolution layer aims to utilize a convolution kernel to slide the kernel over an input tensor's each area with the same size as the kernel. Followed by a convolution layer is a pooling layer, which reduces the input size and, thus, boosts

[1][Online]. Available: https://github.com/appmlk/SeTransformer

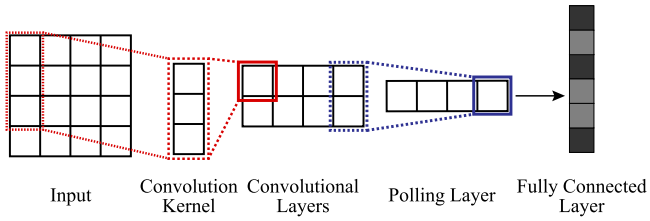


Fig. 1. Overview of the CNN.

training speed. It keeps essential information for the next layer while scaling down input. The next layer is the fully connected layer that outputs the results. By utilizing the difference between the output and the actual target output to construct a cost function, we can then train the CNN by the backpropagation algorithm. Then, we update the model parameters by minimizing the loss between the network output and the target output. Finally, the model can learn meaningful information from input data while reducing the computational cost.

B. Neural Machine Translation

NMT [15] is an end-to-end learning method for automatic translation between different natural languages, such as English to Chinese. The most representative deep learning models of NMT are RNN and Transformer. NMT solves shortcomings such as manual design function requirements and achieves surprising promising results. It usually includes an Encoder module and a Decoder module. Here, the Encoder module is used to encode the input text sequence, and the Decoder module is used to decode the Encoder module's output and generate the corresponding text sequence.

In this article, we leverage the NMT method to train our neural network because NMT makes it possible to translate between different natural languages automatically. Meanwhile, the automatic generation of code comments is a variant of the translation problem between natural languages. For example, Hu *et al.* [5] proved that the NMT method could be applied to the automatic generation of code comments. Therefore, we also comply with the common sequence-to-sequence learning framework for neural network training.

III. Our Proposed Method SeTransformer

In this section, we present the overall framework of our proposed method SeTransformer, which is shown in Fig. 2. Our proposed method consists of two steps: data processing and SeTransformer model training. Specifically, the data preprocessing step can extract the features of the source code. In this step, the source code will be extracted into two parts: lexical information (i.e., code tokens) and structure information (i.e., AST). Later, the SeTransformer model training step can simultaneously learn the program's lexical information and structure information. In the SeTransformer, we also use the CNN layer to reduce the length of the input features to improve the training speed. In the rest of this section, we will show the details of each component in our proposed method.

A. Features Representation

The inputs of the SeTransformer include code tokens and its AST. Thus, the SeTransformer can learn lexical information from code tokens and learn structure information from the AST. Next, we will describe the details of code tokens and AST representation.

1) Code Representation: The SeTransformer learns lexical information from source code tokens. The source code consists of keywords, operators, identifiers, and symbols. To extract each code tokens from the source code, we adopt a widely used tool javalang[2] to process the source code. Since the developers can freely define the identifiers, and these words in the identifiers usually indicate the features of a function or a variable, and so tokens' vocabulary size can be too large, we split each identifier in the source code according to the camel casing conversion and convert all code tokens to lowercase to decrease the vocabulary size. For example, the identifier "calculateAverage" would be split into "calculate" and "average."

We also use positional embedding [16] to show tokens position within the sequence. Because the tokens, ASTs, and comments of code are sequential data, the order relationship between words usually affects the entire sentence's semantic. However, the self-attention layer of the Transformer model does not contain position information, which means that the Transformer model ignores the position information of each part of the input sequence. Therefore, in order to utilize the location information to the training, we need to construct the location information into a sequence with the same dimension as the input sequence (such as tokens, ASTs, and comments) and then add it to the input sequence to obtain new input sequence. Finally, we send the new sequence into the Transformer model so that the position information can participate in the training.

2) AST Representation: The SeTransformer learns structure information from the AST. The AST is an abstract representation of the syntax structure of source code. Previous studies [17], [18] have proven that the AST is one of the essential features in source code analysis.

The AST has a tree structure, which cannot be directly used in neural network training. Hu *et al.* [5] put forward an SBT method to traverse ASTs. SBT uses brackets to represent the AST structure and can restore a tree unambiguously from an SBT sequence.

However, the sequence generated by SBT is relatively long. To reduce the sequence's scale while retaining the input features, we slightly improve SBT, as shown in Fig. 3. In particular, we first apply the SBT method to traverse the ASTs to get the SBT sequence. Then, we replace the brackets with the serial number of the preorder traversal sequence. Finally, we divide the SBT sequence into two sequences: the node sequence and the preorder traversal serial number sequence. We utilize the node sequence as input embedding and employ the serial number as the positional embedding. Specifically, we first embed the node sequence and the number sequence as vectors and then add the two vectors together as input data for the neural network.

[2][Online]. Available: https://pypi.org/project/javalang/

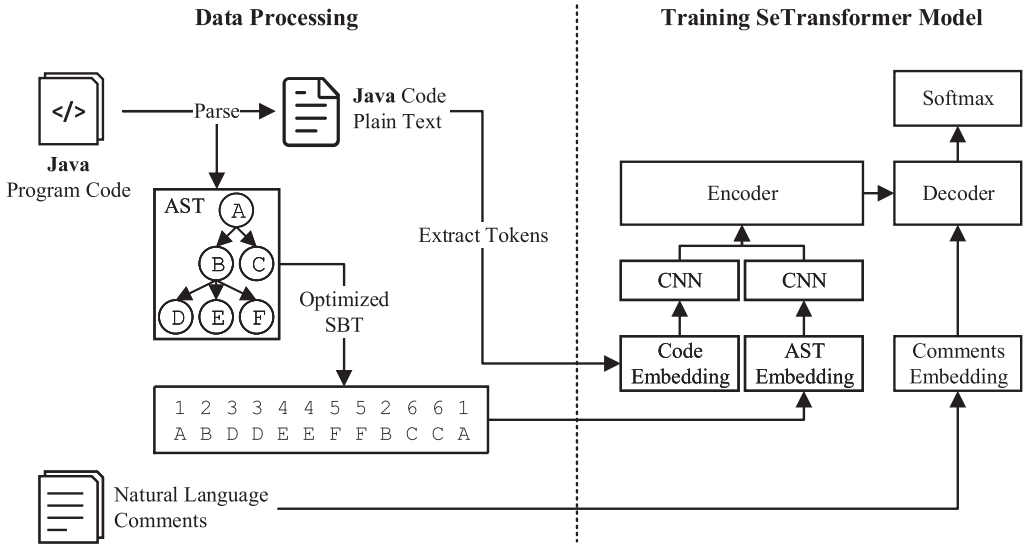


Fig. 2. Framework of the SeTransformer.

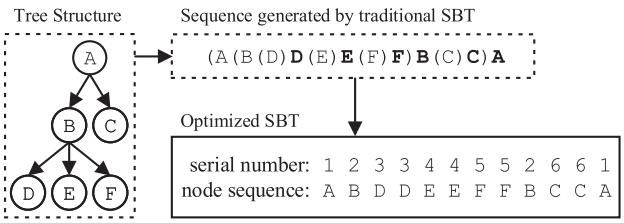


Fig. 3. SBT method.

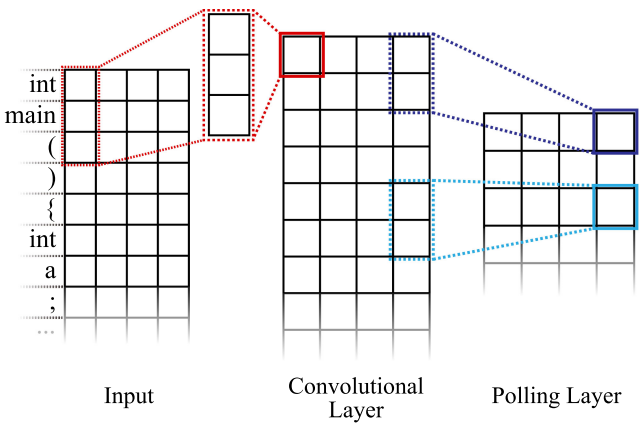


Fig. 4. CNNs for the SeTransformer.

B. Convolutional Neural Networks

The multihead attention layer's memory and computational requirements grow quadratically with sequence length [19]. When multihead attention processes a sequence of length n, its time complexity is $O(n^2)$. Therefore, the Transformer model has difficulty in handling long text sequences.

Inspired by the previous studies on image processing, the CNN can reduce the size of the image (e.g., length and width) and reduce the amount of calculation [20].

In this article, we consider the CNN, which is mainly used to shorten the input sequence's length, as shown in Fig. 4. We first use a 3×1 convolution kernel to extract the features of adjacent characters. Then, to shorten the input sequence's length and maintain the input vector dimension, we use 2×1 filter to do Maxpooling for downsampling. After this series of operations, the input data dimensions can be effectively reduced while retaining the data features.

C. SeTransformer Structure

Fig. 5 illustrates our SeTransformer model. Different from the traditional Transformer described in Section II-A2, we improve the original neural network structure so that the neural network model can simultaneously input plain code text data and AST data.

The detailed introduction of each component is illustrated as follows.

1) Multihead Attention: We construct the multihead attention component according to the standard Transformer and reimplement its calculation process [6]. Since it can compute on the same sequence, we can capture the correlation within the sequence, and this function can be called self-attention.

The particular attention in Transformer is "scaled dot-product attention." The input of scaled dot-product attention consists of queries ($x_i W^Q$), keys ($y_j W^K$), and values ($y_j W^V$). For each attention head, source sequence x_i (where $x_i \in R^{d_{\text{model}}}$) and target sequence y_j (where $y_j \in R^{d_{\text{model}}}$) can be transformed into the output sequence o_i, where $o_i \in R^{d_k}$. Thus, we compute the outputs o_i as

$$o_i = \sum_{j=1}^{m} \alpha_{ij} (y_j W^V) \tag{2}$$

$$\alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^{m} \exp e_{ik}} \tag{3}$$

$$e_{ij} = \frac{(x_i W^Q)(y_j W^K)^T}{\sqrt{d_k}} \tag{4}$$

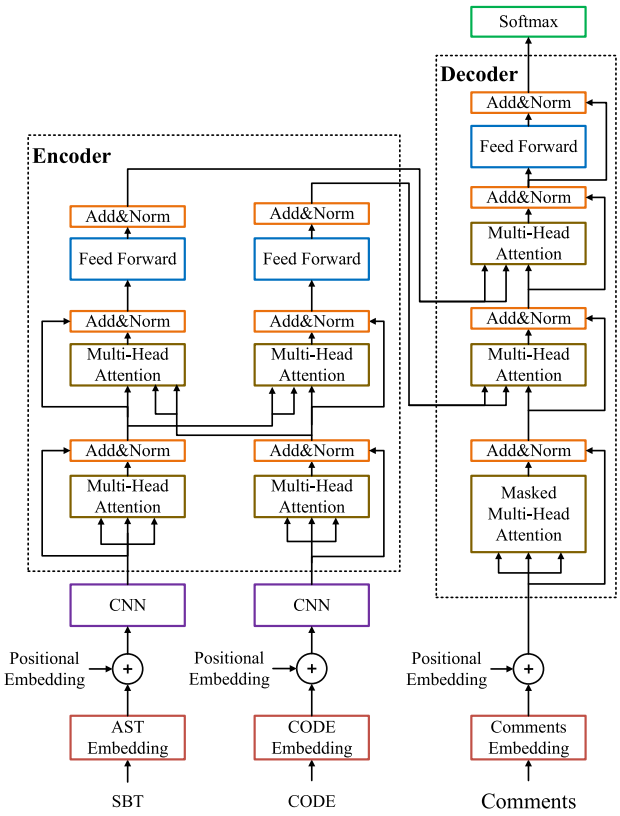


Fig. 5. Structure of the SeTransformer model.

where α_{ij} is a weight coefficient. W^V, W^Q, and W^K are the parameters that are unique per layer and attention head. After the operation of multihead attention, we can extract the internal correlation features of input sequences.

2) Residual Connections and Batch Normalization: Residual connections [21] are mainly used to solve the problem of deep neural network degradation. Batch normalization [22] keeps the input of each layer of the neural network in the same distribution during the deep neural network training process.

Especially, following the previous study [6], we add a layer of residual connections and batch normalization after each layer of multihead attention layer and fully connected layer. This kind of component is labeled as "Add&Norm" in Fig. 5.

3) Encoder: Previous studies usually independently handle coded AST and code tokens in the Encoder component [23], [24]; they did not reflect the association within different features of a program. However, we found some relationship between the AST and code tokens. For example, the leaf nodes of the AST are identifiers or operators in code tokens. To learn the relationship between the AST and code tokens, we use multihead attention to encode the AST and code tokens jointly.

The input of the Encoder includes SBT with serial number information and code with position information. First, to learn the correlation within SBT nodes, we use the SBT sequence as the source sequence and the target sequence for multihead attention. Then, apply code tokens to perform the above operation again to understand the correlation within code tokens. Besides, to acquire the relationship between the AST and code tokens, we input the SBT sequence as the source sequence and the code sequence as the target sequence into multihead attention, plus using code as the source sequence and SBT as the target sequence. Next, we also leverage residual connections, batch normalization, and full connection to complete this neural network contract. As shown in Fig. 5, we utilize a three-layer Encoder network. Finally, the Encoder outputs the encoded SBT and code vectors and then input them into the Decoder.

4) Decoder: The Decoder's input includes partial comments with location information, encoded SBT sequences, and code token sequences. We describe the Decoder based on the Transformer [6].

First, to learn the relationship between comments words, we use the comment sequence as the source sequence and the target sequence for masked multihead attention. In particular, to ensure that the predictions for position i can depend only on the known outputs at positions less than i, we masked multihead attention to hide the words behind each word.

After that, we implement two more multihead attention to decode code information and AST information. One uses the comment sequence as the source sequence and the code sequence as the target sequence. Another uses the comment sequence as the source sequence and the SBT sequence as the target sequence. Next, we also leverage the residual connections, batch normalization, and full connection. Finally, Softmax outputs the probability of each word through the Softmax layer.

IV. Empirical Setup

A. Research Questions

To evaluate the effectiveness of our proposed method SeTransformer, we conduct both a large-scale empirical study and a human study to evaluate the performance of the SeTransformer. The designed research questions are introduced as follows.

RQ1: Can SeTransformer outperform state-of-the-art code comment generation baselines?

We design this RQ to verify the effectiveness of the SeTransformer for code comment generation. To answer this RQ, we conduct a set of empirical studies and compare the SeTransformer with five state-of-the-art code comment generation baselines, including DeepCom [5], Hybrid-DeepCom [24], AST-attendgru [23], Dual Model [25], and Full Model [26].

RQ2: How does code and comment length affect the performance of our proposed method SeTransformer?

We design this RQ to confirm that our method can achieve the best effectiveness under different corpus situations. In this RQ, we intend to analyze the source code and comment length on the SeTransformer method's effectiveness. To answer this RQ, we collected and analyzed the experimental results when using SeTransformer to generate comments for different source code lengths or code comment lengths.

RQ3: What effect does CNN operation have on the performance of the SeTransformer?

We designed this RQ to analyze the influence of the CNN in the SeTransformer. To answer this RQ, we compared the performance of the SeTransformer and its model training time when using CNN and not using CNN.

TABLE I
STATISTICS OF CORPUS

Dataset	Java
Training set	69,708
Validation set	8,714
Test set	8,714
Avg. tokens in comment	17.7
Avg. tokens in code	98.8
Avg. SBT length	101.2

B. Corpus and Preprocessing

We conduct our experiments on a Java corpus [27]. Table I shows the statistics of this corpus. This corpus was collected from GitHub, which contains 87 136 pairs of <method, comment>. The dataset is split into training, test, and validation sets by 8:1:1 based on the suggestion by Hu *et al.* [27]. We preprocessing the dataset following the studies of Wei *et al.* [25], Ahmad *et al.* [26], and Hu *et al.* [27]. The first sentence of Javadoc is extracted as a code comment, and the code comment is used to explain the function of the Java method. Besides, identifiers in the Java method are split via camel casing[3] to alleviate the out-of-vocabulary problem.

C. Metrics

We use two machine translation metrics (i.e., $BLEU$ and $METEOR$) to evaluate the performance of the SeTransformer. These metrics have been widely used in the field of NMT [28] and have now been widely used in the current studies of code comment generation [25], [26].

1) $BLEU$: $BLEU$ [7] is a machine translation metric that we use to evaluate the performance of our proposed method SeTransformer. The score of $BLEU$ represents the similarity between the generated sequence and the reference sequence. The percentage of $BLEU$ scores ranges from 0 to 100%, and a higher $BLEU$ score indicates a more similar generated sequence according to the reference sequence. If two sequences are the same, the $BLEU$ score is 100%. If two sentences do not have the same word, the $BLEU$ score is 0. Recent studies [5], [23], [26] already have widely used $BLEU$ scores to evaluate the quality of code comments.
2) $METEOR$: $METEOR$ [8] is also a machine translation metric. It evaluates the generated sequence by aligning the generated sequence with the reference sequence and calculating the sentence-level similarity score. $METEOR$ employs WordNet[4] to calculate the matching relationship between specific sequences, synonyms, roots, affixes, and definitions, which can be regarded as the supplement of $BLEU$.

[3]For example, the identifier "hashCode" can be split into "hash" and "code" and the identifier "isAgentEmpty" can be split into "is," "agent," and "empty."

[4][Online]. Available: https://wordnet.princeton.edu/

D. Baselines

We compared the SeTransformer with five baselines, which achieved state-of-the-art performance in the code comment generation field [24], [26]. The details of these baselines are summarized as follows:

1) *Baseline 1:* DeepCom [5] formulates the code comment generation task as a machine translation task and uses an attention-based Seq2Seq model to generate comments of Java methods. DeepCom uses the SBT method to convert these ASTs into sequences and takes the SBT sequence as the model input. To compare the performance of SeTransformer and DeepCom, in our study, we reran the code of DeepCom.
2) *Baseline 2:* Hybrid-DeepCom [24] is an extended version of DeepCom, and its performance is better than DeepCom. Hybrid-DeepCom uses a variant of the attention-based Seq2Seq model to generate comments for Java methods. Hybrid-DeepCom combines the source code and the traversed AST sequences to generate the comments. Our study reran the code of Hybrid-DeepCom mode to perform the comparison.
3) *Baseline 3:* AST-attendgru [23] involves two unidirectional GRU layers: one is used to process the words from source code, and the other is designed to process the AST. AST-attendgru first uses an attention mechanism to associate the words in the output comments with the words in the code text and then use a different attention mechanism to associate the comments words with each part of the AST. In our study, we reran the code of AST-attendgru to perform the comparison.
4) *Baseline 4:* The Dual Model [25] is a dual learning framework to train code generation and code summarization models simultaneously to exploit the duality of them. To strengthen the duality, Dual Model adopts a new constraint on the attention mechanism. In our study, we directly use the experimental results of the corresponding study [25] to perform a comparison.
5) *Baseline 5:* The Full Model [26] explores the Transformer model that uses a self-attention mechanism and has shown to capture long-range dependence effectively. They additionally added extra copy attention to the decoder stack to learn the copy distribution, which can allow the Transformer to copy unusual tokens from the source code (e.g., function names and variable names) [29] and, therefore, improve the original Transformer.

To conduct the comparison, we reran the code of Full Model in accordance with the corresponding study [26].

We employ the original RNN and Transformer as baselines to evaluate the performance of our proposed method. Ahmad *et al.* [26] also employed the original RNN and Transformer. Then, they conducted experiments on the same dataset as our study. Therefore, we directly use their experimental results to make the comparison.

Note that the methods in which we directly use the original results (i.e., Dual Model, original RNN, and original Transformer) also use the same dataset and data split method as this article.

Thus, we can directly compare their results in this article's comparison.

Besides, because the experiments in the papers of DeepCom, Hybrid-DeepCom, and AST-attendgru were conducted on a different dataset than the one used in our study, we must rerun their code on the dataset used in our article to complete the comparison. This also explains why the results of these methods in this article differ from their original paper.

E. Hyperparameters

We followed Hu *et al.* [24] by replacing the constant numbers and strings in the source code with special tags <num > and < str>, respectively. We found that about 86% of code comments are less than 30 words in this corpus, approximately 89% of Java methods are less than 200 tokens, and 93% of SBT sequences are less than 300 tokens. Therefore, the maximum length of code sequence, ISBT sequence, and code comment is set to 200, 300, and 30, respectively. We add two special tokens <start > and < eos> for each comment. <start > means the start of the comment, and < eos> means the end of the comment. The vocabulary sizes of the code tokens, SBT tokens, and comments are set to 30 000, 30 000, and 23 428, respectively. Finally, out-of-vocabulary tags will be replaced by <unk>.

Besides, during training, the model is validated every 5000 minibatches on the validation set by $BLEU$ metric, and the maximum number of minibatches is 500 000, which means that the reading of samples from the training dataset will be repeated recursively 500 000 times before the training process is terminated.

The hyperparameters of the neural network model are essential factors that affect the performance of the model. Our model has undergone multiple hyperparameter adjustments and finally selected an optimal parameter combination. To ensure the fairness of our parameter settings, we only use the training set and the validation set to adjust the hyperparameters. The hyperparameters of our model are set as follows:

1) We use the Adam algorithm [30] to train the parameters, and the minimum batch size (i.e., the number of samples selected from training examples for one training) is set to 32 due to the GPU memory limitation.
2) The hidden size is set to 768, and the embedded word has 768 dimensions. We future discuss this parameter setting and comparison in Section VI-A.
3) We use a three-layer Encoder block and a three-layer Decoder block. The multihead attention setting has eight heads. Feedforward has 2048 neurons. These parameters are optimized by the validation set.
4) We train the Transformer model using the Adam optimizer [30], and the initial learning rate is set to 1e-4. The learning rate is decayed using the rate of 0.99.
5) We use the dropout strategy [31] during the training process and set dropout probability to 0.8. This parameter's value is optimized based on the validation set.
6) The SeTransformer uses cross-entropy minimization as the cost function by referencing the work of Hu *et al.* [24].

TABLE II
AVERAGE PERFORMANCE OF DIFFERENT METHODS

Method	$BLEU$ (%)	$METEOR$ (%)
RNN	39.33	21.38
Transformer	43.41	25.91
DeepCom	40.51	22.24
Hybrid-DeepCom	42.26	24.86
AST-attendgru	40.82	24.54
Dual Model	42.39	25.77
Full Model	44.88	25.54
SeTransformer	**49.41**	**30.40**

F. Statistical Analysis

1) Hypothesis Testing Method: We use the Wilcoxon signed-rank test to further assess the trial findings since we conduct experiments between our proposed method and existing baselines on the same data with the same code and comments.

The Wilcoxon signed-rank test [32] is used as an alternative hypothesis test when the test data cannot be assumed to be normally distributed. As a result, it can provide a solid statistical foundation for comparing the effectiveness of various methods.

2) Effect Size: An effect size is a quantitative measure of the strength of the association between two variables in a population or a sample-based estimation of that quantity in statistical analysis. To quantify the magnitude of difference between the two groups, we calculate the Cliff's Delta [33], which is a nonparametric effect size measure. Specifically, we use the Cliff's Delta to quantify the difference in terms of $BLEU$ or $METEOR$ metrics between the SeTransformer and other baselines.

Furthermore, the effect size classifies Cliff's Delta values of less than 0.147, between 0.147 and 0.33, between 0.33 and 0.474, and above 0.474 as negligible, small, medium, and large, respectively.

V. RESULT ANALYSIS

A. Result Analysis for RQ1

RQ1: Can SeTransformer outperform state-of-the-art code comment generation baselines?

We first use two machine translation metrics (i.e., $BLEU$ score and $METEOR$) to measure the difference between automatically generated comments and manually written comments and then employ five state-of-the-art baselines to verify the performance of the SeTransformer. Table II lists the overall results of the SeTransformer model and the five state-of-the-art baselines. In Table II, it is not hard to find that the SeTransformer outperforms all the other five baselines. More specifically, the SeTransformer improves by 4.53–8.90% on $BLEU$ compared to state-of-the-art baselines and improves by 4.63–8.16% on $METEOR$.

The SeTransformer extracts lexical information, and grammatical information then encodes them jointly in the Encoder. In comparison, the shortcomings of these baselines are summarized as follows:

1) The Dual Model and the Full Model only use lexical information but ignore grammatical information.

TABLE III
p-VALUE OF HYPOTHESIS IN SECTION V-A

Method	$BLEU$	$METEOR$
DeepCom	1.33e-216	0.00
Hybrid-DeepCom	3.52e-123	6.11e-238
AST-attendgru	2.45e-152	9.12e-225
Full Model	2.69e-105	2.28e-171

TABLE IV
CLIFF'S DELTA BETWEEN THE SETRANSFORMER AND BASELINES

Method	$BLEU$	$METEOR$
DeepCom	0.15	0.25
Hybrid-DeepCom	0.09	0.12
AST-attendgru	0.09	0.13
Full Model	0.09	0.10

2) DeepCom only utilizes grammatical information but ignores lexical information.
3) Hybrid-DeepCom and AST-attendgru use lexical information and grammatical information. However, the grammatical information and lexical information are separately coded in the model's Encoder. Thus, the relationship between lexical and grammar is ignored.

Note that the results in terms of BLEU and METEOR metrics in the Full-Model-related paper are 44.58 and 26.43%, respectively, and the slight differences are caused by random factors.

The experimental results show that the SeTransformer achieves the best performance and proves the importance of lexical information and grammatical information as feature information in this research field. Therefore, these two features should be considered simultaneously in the encoding process.

Moreover, after further analysis of the performance between five state-of-the-art baselines in Table II, we can achieve the following findings.

1) The performance of Hybrid-DeepCom is better than DeepCom, which proves that grammatical information used in DeepCom cannot generate high-quality comments.
2) Hybrid-DeepCom is superior to ASTattendgru, which demonstrates that the generation of code comments is suitable for the encoder–decoder framework.
3) The Dual Model is better than Hybrid-DeepCom, which shows a correlation between the code generation task and the comment generation task.
4) The Full Model can achieve the best performance among the five baselines, which shows that the Transformer model is better than the RNN model.

Furthermore, we leverage the Wilcoxon signed-rank test to verify the competitiveness of our proposed method SeTransformer. The results of the hypothesis testing are shown in Table III. Owing to the reason that we only implemented four baseline methods, we can only conduct hypothesis testing on these four baselines. The following is the hypothesis used in our study, H_0: In terms of $BLEU$ and $METEOR$, there is no significant difference between the SeTransformer and the other method. This test's significance threshold is set to 0.05. Because all of the p-values in Table III are less than 0.05, the null hypothesis was rejected by the statistical results. These results indicate that in terms of $BLEU$ and $METEOR$ metrics, the performance of our proposed method differs significantly from that of these baselines. Since the results in Table II show that our method outperforms other baselines, we can conclude that the SeTransformer can achieve significantly better results than those of other baseline methods.

Cliff's Delta is also used to assess the difference in terms of $BLEU$ and $METEOR$ metrics between the SeTransformer and other baselines. The Cliff's Delta values are less than or equal to 0.25, which equates to a negligible or small effect size. As shown in Table IV. The results demonstrate that our method outperforms existing baselines to a lesser extent.

Summary for RQ1: Experimental results show that the performance of the SeTransformer is better than that of the five state-of-the-art baselines and two basic baselines. It proves that lexical information and grammatical information are essential features in the code comment generation task, and researchers should consider both the features in the Encoder process of this field research.

B. Result Analysis for RQ2

RQ2: How does code and comment length affect the performance of our proposed method SeTransformer?

Code and comment length is one of the main factors that affect the performance of the code comment generation model; therefore, we further analyze the impact of code and comment length on the performance of the SeTransformer. We reran three baselines, namely DeepCom, Hybrid-DeepCom, and ASTattendgru. Therefore, we use these three baselines as a comparison to evaluate the performance of the SeTransformer. Fig. 6 shows the performance of the SeTransformer and the other three baselines under different lengths of code and comments.

To make the results of Fig. 6 more concise and significant, we take the approximate length of the code to study the model's performance. We calculated the approximate length as follows:

$$F(x) = \begin{cases} 300, & x \geq 300 \\ (\lfloor x/10 \rfloor + 1) * 10, & x < 300 \end{cases} \tag{5}$$

where $F(x)$ is the length of the code shown on the x-axis of Fig. 6, and x is the original code length.

Fig. 6(a) shows the impact of code length on the performance of the four models on the $BLEU$ metric. It can be seen that all models perform very poorly when the code length is short; this is because these short code may be incomplete. When the code length is larger than 200, all models are unstable; this is because developers will crop too large code; therefore, this kind of sample size is small, which cannot reflect this interval's true situation. From Fig. 6(a), we can find that our model's performance is the best. Compared with DeepCom, we notice that except for the first point, the $BLEU$ of the SeTransformer is higher than DeepCom at other points. Among the four models, the SeTransformer can achieve the best performance at 22 points, with a ratio of 73.33%.

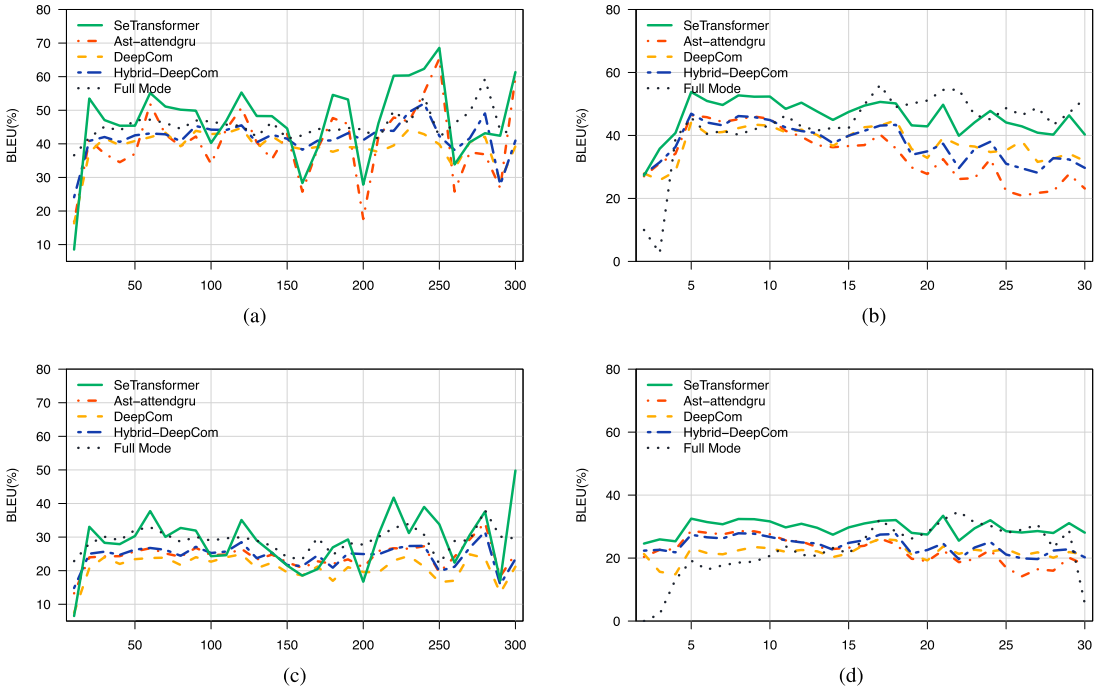


Fig. 6. Performance of the SeTransformer and the three baselines under different lengths of code and comments. (a) $BLEU$ scores for different code lengths. (b) $BLEU$ scores for different comment lengths. (c) $METEOR$ scores for different code lengths. (d) $METEOR$ scores for different comment lengths.

Fig. 6(c) shows the impact of code length on the performance of the four models on the $METEOR$ metric. As in Fig. 6(a), when the code length is short, the performance of the model is low, and when the code length is greater than 200, the performance of the model becomes unstable. From Fig. 6(b), we can find that our model's performance is the best. Among the four models, the SeTransformer can achieve the best performance at 20 points, with a ratio of 66.67%.

Fig. 6(b) and (d) shows the impact of code length on the $BLEU$ and $METEOR$ metrics of the four models. It can be seen from the figure that our model achieves the best results on both $BLEU$ and $METEOR$. Among the four models, the SeTransformer can achieve the best performance at 30 points on both $BLEU$ and $METEOR$, with a ratio of 100%. Besides, the three baselines models' performance decreases significantly when the comment length exceeds 20. In other words, when faced with the long-term dependence problem, the SeTransformer can still achieve the best performance compared with the other three baselines.

In addition, we use the Wilcoxon signed-rank test to verify the competitiveness of our proposed method. In other words, we utilize statistical analysis to determine whether our method's polyline in Fig. 6 is significantly higher than that of other methods. The hypothesis used in this section is specifically specified as follows, H_0: In terms of $BLEU$ and $METEOR$, there is no significant difference between SeTransformer and the other methods in terms of various code lengths or comment lengths. Table V shows the p-value of H_0 for four different methods using the Wilcoxon signed-rank test. Because the p-value in most cases is less than 0.05, the preceding results that the SeTransformer can produce significantly superior performance than DeepCom, Hybrid-DeepCom, and AST-attendgru are safe to accept. The Full Model does not differ significantly from our solution in this research question, indicating that the advantages of the Full Model can achieve better performance under certain code or comment lengths.

TABLE V
p-VALUE OF HYPOTHESIS IN SECTION V-B

	Method	*BLEU*	*METEOR*
Code length	DeepCom	3.73e-04	8.07e-06
	Hybrid-DeepCom	7.05e-03	1.65e-03
	AST-attendgru	1.18e-05	9.28e-04
	Full Model	0.29	0.71
Comment length	DeepCom	3.00e-06	2.70e-06
	Hybrid-DeepCom	3.00e-06	2.70e-06
	AST-attendgru	2.70e-06	2.70e-06
	Full Model	0.32	1.68e-04

In addition, Table VI compares the Cliff's Delta between the SeTransformer and alternative techniques for various code or comment lengths using the $BLEU$ and $METEOR$ metrics. The results in Table VI show that, in the majority of cases, the SeTransformer outperforms alternative baselines in terms of $BLEU$ and $METEOR$ metrics for different code and comment lengths.

Summary for RQ2: When considering the impact of code and comment length on the model, the performance of the SeTransformer is better than that of the other three baselines. Besides, the experimental results also show that when the code is too short, the SeTransformer performs poorly because the source

TABLE VI
CLIFF'S DELTA UNDER DIFFERENT CODE LENGTHS AND COMMENT LENGTHS

	Method	$BLEU$	$METEOR$
Code length	DeepCom	0.61	0.64
	Hybrid-DeepCom	0.45	0.43
	AST-attendgru	0.41	0.44
	Full Model	0.25	0.03
Comment length	DeepCom	0.72	0.98
	Hybrid-DeepCom	0.67	0.88
	AST-attendgru	0.74	0.85
	Full Model	0.08	0.55

TABLE VII
IMPACT OF THE CNN ON THE PERFORMANCE OF THE SETRANSFORMER

Method	$BLEU$ (%)	$METEOR$ (%)	$Time$ (min)
SeTransformer without CNN	49.11	30.33	3745
SeTransformer with CNN	**49.41**	**30.40**	**2344**

code with a shorter length is incomplete, which will affect the performance of the code comment generation method.

C. Result Analysis for RQ3

RQ3: What effect does CNN operation have on the performance of the SeTransformer?

We add a CNN layer to compress the dimensionality of the input data before the input data enters the Transformer model. In this section, we will analyze the impact of the CNN on the performance of the SeTransformer.

Note that regardless of whether the neural network structure contains the CNN, the training process must be completed in its entirety and will not be aborted. The CNN's focus on reducing training time is reflected in each feedforward and feedback operation of the neural network, rather than limiting the number of training times, thereby reducing the total time required for training.

To study CNN's performance on the SeTransformer, we compare the SeTransformer with CNN (used in the previous experiment) and the SeTransformer without CNN. Table VII shows the final comparison results. We found that the SeTransformer with CNN and the SeTransformer without CNN are very similar in $BLEU$ and $METEOR$ metrics, which shows that adding a CNN layer to compress the dimensionality of the input data will not affect the performance of the model. However, in terms of training time, the SeTransformer without CNN takes 3745 min to train a model, but the SeTransformer with CNN only takes 2344 min, which shortened training time by 37.41%. Compared with the SeTransformer without CNN, although the SeTransformer with CNN has one more CNN layer network, the training time of the SeTransformer is shorter, which shows that using CNN to compress the data dimension can reduce the training time.

To verify whether there are significant differences in comment generation performance after using CNN, we use the Wilcoxon signed-rank test. In particular, we want to know if the evaluation metrics' values have decreased significantly after using CNN. First, the confidence level is set to $\alpha = 0.05$, and we define the null hypothesis (H_0) as that the performance of the SeTransformer without CNN is significantly better than the SeTransformer with CNN in terms of $BLEU$ and $METEOR$ metrics.

Finally, the p-value of the Wilcoxon signed-rank test is 0.00001 in terms of $BLEU$ metric, which is less than 0.05; therefore, we reject H_0 and accept H_1. That is, the performance of comments generation is not significantly reduced after using CNN in terms of $BLEU$ metric. For $METEOR$ metric, the p-value is 0.10196, which is larger than 0.10196; therefore, H_0 is adopted in this case. That is, the two approaches produce similar results.

Summary for RQ3: Using our proposed CNN layer in the SeTransformer can save 37.41% of the training time. The performance of the SeTransformer model would not be significantly reduced only in terms of $BLEU$ metric.

TABLE VIII
IMPACT OF HIDDEN SIZE ON THE PERFORMANCE OF THE SETRANSFORMER

Hidden-size	$BLEU$ (%)	$METEOR$ (%)
128	47.17	28.72
256	48.78	30.01
284	49.21	30.64
512	49.27	30.59
640	49.36	30.74
768	49.41	30.40

VI. DISCUSSIONS

A. Impact of Hidden Size

The hidden size is an essential parameter of neural networks. It significantly impacts the performance of the trained neural network models. We performed a sensitivity analysis by adjusting the size of the neural network's hidden layer. To ensure a fair comparison of the experiments, we set the same value for the parameters except for the hidden size, and the results are presented in Table VIII.

As shown in Table VIII, the value of $BLEU$ and $METEOR$ generally increases with the increase in the hidden size, which means that expanding the scale of hidden size can effectively improve the performance of the SeTransformer.

However, we also observed that the growth rate of $BLEU$ and $METEOR$ becomes smaller with the hidden size increase to a certain value. In particular, although $BLEU$ can reach the highest value when hidden size is 768, the value of $METEOR$ slightly reduces. Therefore, we set the scale of hidden size to 768 in our experiment.

B. Human Evaluation

In this section, we conduct a manual evaluation on the quality of comments generated by the Full Model and the SeTransformer because, sometimes, the automated evaluation formula is not equal to the real evaluation of developers. Therefore, we conducted a human survey to evaluate the automated comment

Answer questions for the following code:

```
public void forward(HttpServerRequest request)
{
    forward(request, null);
}
```

Reference comment: handles the request and forwards it to the hook specific destination.

Candidate 1: send a http request to the request queue.	
Please evaluate the naturalness of Candidate 1:	Score from 1-5 (5 is the best)
Please evaluate the relevance of Candidate 1:	Score from 1-5 (5 is the best)
Candidate 2: forwards the http request url back to the request.	
Please evaluate the naturalness of Candidate 2:	Score from 1-5 (5 is the best)
Please evaluate the relevance of Candidate 2:	Score from 1-5 (5 is the best)

Fig. 7. One page in the questionnaire of manual evaluation.

generation method by real developers. We hired four volunteers to participate in our manual evaluation (including undergraduates, masters, and Ph.D. students), each with two to five years of programming experience and large-scale project development experience.

To balance efficiency and experimental credibility, we used algorithms commonly used in sampling survey research [34] to calculate the number of comments that need to be extracted for evaluation. As described in Section IV-B, 87 136 Java programs are used in this experiment; it is almost impossible for the volunteers to mark all of them. The calculation formula of the sample size MIN is computed as follows:

$$MIN = \frac{n_0}{1 + \frac{n_0 - 1}{populationsize}} \tag{6}$$

$$n_0 = \frac{Z^2 \times 0.25}{e^2} \tag{7}$$

where $populationsize$ indicates the number of code comments, Z means the confidence level, and e is the error margin. In our experiment, we set the value of Z to 95% and the value of e to 0.05. Then, the calculated value of MIN is 384.

We invite five volunteers with extensive development expertise to provide feedback for our comparison. We followed the experimental design principles[5] and conducted a within-subject experiment, as each volunteer will respond to the same questions under the same conditions.

We randomly chose 384 pairs of prediction outcomes and their references from the test set. The questionnaire contains 384 pages, and each page contains an input source code, comments generated by the SeTransformer and the Full Model, and a hand-written reference comment. We send each volunteer a copy of the 384-page questionnaire and invite them to evaluate two comments for each code. Additionally, to guarantee fairness, we randomly rank the comments created by the two methods on each page and remove their tags to ensure that the volunteers cannot know whether the comments are generated by the Full Model or the SeTransformer. During the manual evaluation, volunteers can resort to search engines (such as Google) for related material and unfamiliar concepts.

To enable volunteers to assess the quality of generated comments from a variety of perspectives, we adopt Gao *et al.*'s approach of considering two perspectives: naturalness and relevance [35]. Naturalness relates to the grammatical accuracy and fluency of generated comments, i.e., whether the content of a comment is easily readable and understandable by humans. Relevance relates to the relationship between the generated comments and the input code, i.e., whether humans can deduce the code's design intent from the corresponding comment. Fig. 7 depicts one page of our questionnaire on which volunteers should read the input code, reference comment, and two generated comments. Then, the volunteers should grade the naturalness and relevance of the two generated comments by using a scale of 1–5 (5 is the best).

TABLE IX
MANUAL ANALYSIS

Type	SeTransformer	Full Model
Naturalness	**4.09**	3.90
Relevance	**3.95**	3.79

Finally, we calculate the mean results of the five volunteers' feedback, as shown in Table IX. For instance, the number 3.79 in the second row and the second column indicates that the Full Model's average relevance score is 3.79. It is discovered that SeTransformer's average naturalness and relevance scores outperform those of the Full Model by 0.19 and 0.16 points, respectively, which means that the volunteers prefer the comments generated by the SeTransformer. Besides, we employ Cliff's Delta [33] to evaluate the difference in naturalness and relevance of comments between the SeTransformer and the Full Model. Cliff's Delta values for naturalness and relevance are 0.11 and 0.09, respectively, indicating that the SeTransformer has a weak advantage when compared with the Full Model.

C. Explanation of How the SeTransformer Generates Comments

The attention map in Fig. 8 shows an example of how the SeTransformer generates a comment. Specifically, the tokenized input is shown on the y-axis, while the comment generated

[5][Online]. Available: https://opentextbc.ca/researchmethods/chapter/experimental-design/

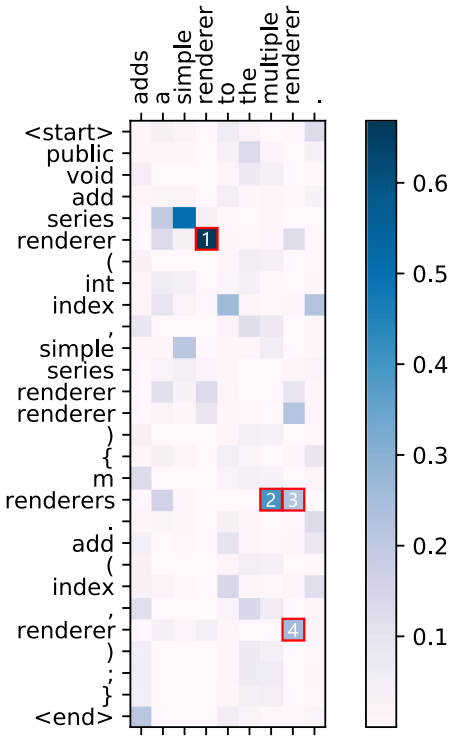


Fig. 8. Attention map for the correct comments generated by the SeTransformer.

output is displayed on the x-axis. The attention map represents the relationship between the input code tokens and the generated comments. The color range of the color block on the right side of the image is light (light blue) to dark (dark blue). The deeper the color, the higher the correlation degree of the token.

This attention map can help us understand how the SeTransformer generates specific words for input tokens. For example, the words "renderer" in comments are generated because of the tokens "renderer" in inputs (labeled 1, 3, and 4 in Fig. 8), which means that the SeTransformer can capture the keywords in the input tokens and keep them in the output comments. Besides, the plural word "multiple" in comments is generated because the "renderers" in the input tokens is plural (labeled 2 in Fig. 8), which means that the SeTransformer can capture the difference between singular nouns and plural nouns and present them in the output comments. Therefore, this example can explain how the SeTransformer can generate correct comments.

To further investigate the effect of code features on SeTransformer performance, we attempted to conduct qualitative study by analyzing the training data and output comments.

Three different cases we chose to analyze are listed as follows:

1) *Perfect case:* The manual comment in this case is "get a sorted array containing all column values for a given tuple iterator and field," and the comment generated by the SeTransformer is exactly the same as the manual comment. We counted the number of occurrences of the key phrases "get a," "a sorted," and "sorted array" in the training set as 317, 37, and 26, respectively. Therefore, the neural network trained by this training set will be familiar with the generation process of this series of phrases.

TABLE X
IMPACT OF THE CONNECTION MODEL ON THE PERFORMANCE OF THE SETRANSFORMER

Data Type	*BLEU* (%)	*METEOR* (%)
Re-adjusted SeTransformer	49.59	30.96
Original SeTransformer	49.41	30.40

2) *Medium case:* The manual comment for this case is "compute the union size of two bitsets," while the comment generated by the SeTransformer is "compute the intersection size of two bitsets." It can be seen that the difference between the two comments appears in the middle part of the sentence. We counted the number of occurrences of the phrase "compute the union" in the training set only once, while the number of occurrences of "compute the intersection" was nine. It can be seen that the imbalance of the data causes the neural network to deviate when judging the two words "union" and "intersection."
3) *Poor case:* According to our observations, cases with poor performance are generally wrong in the first word, which ultimately results in a huge difference between the entire sentence and the manual comment. For example, the beginning of a human comment is the word "ignorable," and this word only appears twice as the beginning of a sentence in the entire training set. The SeTransformer predicts the beginning of this sentence as "the," which occurs as the beginning of a sentence 666 times. This imbalance caused the neural network to forget "ignorable," and in the end, the deviation of the first word led to errors in the entire comment generation process.

From the above analysis, we think the balance of the dataset plays a crucial role in the effectiveness of the SeTransformer. Therefore, we will apply a more balanced dataset to our method in our future work.

D. Impact of the Encoder and Decoder's Connection Mode

As illustrated in Fig. 5, the default connection method for the encoder and the decoder in the SeTransformer proposed in our study is to transfer the code token information first, followed by the SBT information. To determine the influence of this order on SeTransformer's effectiveness, we reverse the order of data transfer (i.e., transfer SBT information first, followed by code token information) and reconduct the experiment. Finally, Table X summarizes the experimental results.

In Table X, the term "Readjusted SeTransformer" refers to the neural network structure that alters the order in which data are transferred between the encoder and the decoder. As can be found in Table X, the effect of the readjusted SeTransformer is no worse than the original SeTransformer. Therefore, the connection mode of the encoder and the decoder in the SeTransformer does not lead to performance loss.

E. Impact of CNN Convolution Kernel Size

Our proposed SeTransformer incorporates a CNN component prior to the encoder, lowering the dimensionality of the input

TABLE XI
IMPACT OF THE CNN SIZE ON THE PERFORMANCE OF THE SETRANSFORMER

CNN size	*BLEU* (%)	*METEOR* (%)
2×1	49.42	30.82
5×1	49.49	30.70
10×1	49.64	30.84
20×1	49.48	30.86
Original SeTransformer	49.41	30.40

TABLE XII
IMPACT OF DATA TYPE ON THE PERFORMANCE OF THE SETRANSFORMER

Data Type	*BLEU* (%)	*METEOR* (%)
Only Code Tokens	48.48	30.39
Only SBT	47.28	28.56
Original SeTransformer	**49.41**	**30.40**

data and increasing the neural network's training speed. A larger convolution kernel results in a better compression ratio, which allows for a reduction in the dimensionality of the input data. A well-chosen convolution kernel can lower the dimensionality of the input data while retaining its features. However, an excessively large convolution kernel will cause the input data to lose its original information.

In order to study the influence of the size of the convolution kernel on the SeTransformer, we set the convolution kernel from the original 3×1 to 5×1, 10×1, and 20×1 and then conducted experiments. The experimental results are shown in Table XI.

As shown in Table XI, the SeTransformer performs almost the same with different values of CNN's convolution kernel size.

F. Impact of Data Type

To investigate the influence of input data type on our proposed SeTransformer, we conducted an experiment in this subsection. Specifically, we train the SeTransformer using only code tokens or SBT sequences. When one data type is used, the other input is a sequence of zeros. Table XII summarizes the experimental results.

As shown in Table XII, the SeTransformer can achieve the best performance when two types of data are used together. Therefore, this experiment demonstrates that combining two types of data together can increase the SeTransformer method's performance.

VII. THREATS TO VALIDITY

In this section, we discuss the potential threats to our study.

A. Internal Validity

One threat to internal validity comes from possible errors in our experimental program code. To avoid this problem, we carefully checked the code and conducted a small-scale test before the formal experiment. Besides, we implemented our neural network model based on the well-known open-source machine learning platform TensorFlow[6] to ensure the neural network's correct operation.

Besides, the replication error may cause the results of the baseline method in this article to be inaccurate. To mitigate this threat, we used the same parameter settings as the studies they conducted.

Another threat to internal validity is that the performance of our method may depend on the hyperparameter configuration. In this article, hyperparameter settings mainly come from validation set optimization and previous studies, which is discussed in Section IV-E.

In addition, the internal validity of the SeTransformer is threatened by the unpredictability of how the connection between the Encoder and the Decoder influences the experimental outcomes. In this article, we complete the network model by sequentially inputting two types of data into the Decoder. However, there are additional implementations of joint learning that are possible (e.g., using a simple merged feedforward network or a bilateral neural network). To address this threat, we intend to incorporate more joint learning approaches into our future work to go deeper into discovering improved connections between the Encoder and the Decoder and the causes for their effect differences.

B. External Validity

The external validity relates to the corpus we collected for our experiment. This corpus was gathered from many open-source Java projects, and their comments from GitHub refer to previous comments generation studies [25]–[27]. Although the previous research has removed the corpus's noises, the corpus contains the same comment function pairs that could not match because of programs' rapid updations. In the future, we want to collect more programs with higher quality comments for experiments.

C. Construct Validity

The construct validity relates to the suitability of the evaluation metrics used in our study. We utilize two evaluation metrics, namely *BLEU* and *METEOR*, because these metrics have been widely used in previous NMT and natural language process domains [25], [26], [28].

VIII. RELATED WORK

A. Code Summarization

Code summarization improves the comprehensibility of the source code by generating alternative natural language descriptions for the source code. Code summarization methods can be divided into two categories: template-based code summarization [2], [36]–[40] and artificial intelligence (AI)-based code summarization [23], [24], [41], [42].

Template-based automatic generation of code comments is the earliest automatic generation method of code comments, which can use various intermediate information extracted to generate

[6][Online]. Available: https://www.tensorflow.org/

comments for the source code. The software word usage model (SWUM) [?] is a technique for finding the part of speech of words in the code. Sridhara *et al.* [1] adapted the SWUM to use templates to create short comment phrases for the source code. Dawood *et al.* [43] defined some templates of program structure information and used the templates to generate comments for the source code. For example, a template defines the number of interfaces in the package or the parameter types used by methods. Wang *et al.* [44] use natural language processing to identify actions, topics, and auxiliary parameters to fill in templates and some basic information.

With AI technology development, AI-based code comment generation methods have become more and more popular. AI-based code comment generation methods are similar to machine translation, using neural network technology to generate code comments. Iyer *et al.* [4] were the first to try to use neural network technology to generate code comments and proved that neural networks could be used in the field of code comment generation. They developed a new AI-based code comment generation method called CODE-NN, which uses the RNN with an attention mechanism to generate natural language descriptions for C# code snippets and SQL queries. CODE-NN uses the source code as plain text as the input of the neural network. Hu *et al.* [5] propose an SBT method to traverse ASTs and generate SBT sequence. They regard the code comment generation task as a machine translation task and propose a novel code comment generation method called DeepCom. This method takes SBT as input. Hu *et al.* [24] proposed Hybrid-DeepCom to automatically generate natural language descriptions for the java method, which is an extension of DeepCom. Compared with DeepCom, Hybrid-DeepCom adds lexical information to the input. Moreover, it proves that lexical information is helpful to the generation of code comments. Wei *et al.* [25] argue that code generation and code summarization are related to each other and joint training of two models can learn this relationship. Therefore, they designed a dual learning framework to train both the code summarization and code generation models to take advantage of their duality. Wei *et al.* [41] argue that neural code comments generation methods tend to generate high-frequency words. Therefore, they concluded that the neural model was not sufficient to generate comments only based on the source code. And they combine information retrieval and neural network technology to propose a comment generation framework, namely Re2Com. Ahmad *et al.* [26] propose a Transformer-based method to generate natural language descriptions for java methods. This method uses the Transformer model to learn the order of tokens in a sequence or model the relationship between tokens. LeClair *et al.* [45] developed a method for generating comments that use a graph-based neural architecture that is more comparable to the default structure of the AST. Wang *et al.* [46] presented a novel strategy for generating code comments that incorporates numerous code features, such as type-augmented ASTs and program control flows, and the experimental results outperform existing approaches.

We introduced an AI-based code comment generation method called SeTransformer. Unlike the previous method, the SeTransformer uses the Transformer framework to learn the semantic information of the code. The empirical results also verify the effectiveness of our proposed method.

B. Language Models for the Source Code

The language model is a formal system. The objective facts of the language can be automatically processed by the computer after being described by the language model. Therefore, the language model is of great significance to the information processing of natural language. In recent years, the language model for the source code has been successfully applied to many software engineering tasks, such as code comment generation [26], clone [47]–[49], code generation [17], [50], [51], and defect prediction [52]–[54].

Hindle *et al.* [55] first tried to model the language model of the source code and proved that the model did capture the advanced statistical laws of the software at the n-gram level. Allamanis *et al.* [56] propose a framework called NATURALIZE to solve the problem of local convention coding convention reasoning. The framework can provide some suggestions to improve the style consistency of the code base. NATURALIZE can be applied to rule-based formatter inference rules. Mou *et al.* [57] proposed a tree-based CNN framework called TBCNN, which is based on the AST of the program. They also put forward the concept of "continuous binary tree." The TBCNN model is a general architecture and can be used in many software engineering tasks (code comment generation and clone detection). Yin and Neubig [58] propose a grammar-based neural network framework to generate code for natural language AST. This method generates the AST by applying actions in the grammar model.

Our research proposes a novel code comment generation method called SeTransformer, which uses and improves the Transformer framework. This method can generate corresponding comments by learning the semantic information of the code.

IX. Conclusion

In this article, we proposed a code comment generation method SeTransformer based on the Transformer neural network structure. In particular, we used the source program's lexical and grammatical information by inputting the linguistic data and the encoded AST information. Moreover, we leveraged CNN to compress the dimensionality of the data for speeding up the neural network's training process. Finally, the SeTransformer used an improved transformer model to perform encoding and decoding, thus generating useful and readable code comments.

To verify the performance of our method, we conducted a number of empirical studies on a public, large-scale, and open-source corpus. The experimental results showed that the performance SeTransformer is significantly better than that of the other five state-of-the-art baseline methods. Specifically, the SeTransformer's effectiveness led by 8.9% highest in terms of the $BLEU$ metric and led by 8.16% highest in terms of the $METEOR$ metric. Besides, we also conducted the questionnaire survey and the results showed that the SeTransformer can generate high-quality comments and lead the comparison method Hybrid-DeepCom by 0.35 points.

In the future, we plan to extract more features from the program's dynamic execution information and combine them with the more acceptable neural network structure to generate more accurate code comments.

REFERENCES

[1] G. Sridhara, E. Hill, D. Muppaneni, L. L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proc. 25th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2010, pp. 43–52.

[2] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Proc. 21st Int. Conf. Prog. Comprehension*, 2013, pp. 23–32.

[3] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proc. 22nd Int. Conf. Prog. Comprehension*, 2014, pp. 279–290.

[4] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, 2016, pp. 2073–2083.

[5] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. 26th Conf. Prog. Comprehension*, 2018, pp. 200–210.

[6] A. Vaswani *et al.*, "Attention is all you need," in *Proc. Int. Conf. Neural Inf. Process. Syst*, 2017, pp. 5998–6008.

[7] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics*, 2002, pp. 311–318.

[8] S. Banerjee and A. Lavie, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proc. Workshop Intrinsic Extrinsic Eval. Measures Mach. Transl. Summarization*, 2005, pp. 65–72.

[9] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[10] K. Cho *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2014, pp. 1724–1734.

[11] Y. Kim, "Convolutional neural networks for sentence classification," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2014, pp. 1746–1751.

[12] J. Gehring, M. Auli, D. Grangier, and Y. N. Dauphin, "A convolutional encoder model for neural machine translation," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics*, 2017, pp. 123–135.

[13] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2013, pp. 1310–1318.

[14] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," in *Proc. NIPS 2014 Workshop Deep Learn.*, 2014.

[15] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144.*

[16] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2019, pp. 4171–4186.

[17] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural CNN decoder for code generation," in *Proc. 33rd AAAI Conf. Artif. Intell.*, 2019, pp. 7055–7062.

[18] V. Jayasundara, N. D. Q. Bui, L. Jiang, and D. Lo, "TreeCaps: Tree-structured capsule networks for program source code processing," 2019, *arXiv:1910.12306.*

[19] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," 2019, *arXiv:1904.10509.*

[20] F. Jiang, W. Tao, S. Liu, J. Ren, X. Guo, and D. Zhao, "An end-to-end compression framework based on convolutional neural networks," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 28, no. 10, pp. 3007–3018, Oct. 2018.

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[22] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. 32nd Int. Conf. Mach. Learn.*, 2015, pp. 448–456.

[23] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 795–806.

[24] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Softw. Eng.*, vol. 25, no. 3, pp. 2179–2217, 2020.

[25] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2019, pp. 6559–6569.

[26] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "A transformer-based approach for source code summarization," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 4998–5007.

[27] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred API knowledge," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, 2018, pp. 2269–2275.

[28] R. Aharoni and Y. Goldberg, "Towards string-to-tree neural machine translation," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics*, 2017, pp. 132–140.

[29] K. Nishida *et al.*, "Multi-style generative reading comprehension," in *Proc. 57th Annu. Meeting Assoc. Comput. Linguistics*, Jul. 2019, pp. 2273–2284.

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980.*

[31] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.

[32] V. R. Prybutok, "An introduction to statistical methods and data analysis," *Technometrics*, vol. 31, no. 3, pp. 389–390, 2012.

[33] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," *Psychol. Bull.*, vol. 114, no. 3, pp. 494–509, 1993.

[34] R. Singh and N. S. Mangat, *Elements of Survey Sampling*, vol. 15. New York, NY, USA: Springer, 2013.

[35] Z. Gao, X. Xia, J. C. Grundy, D. Lo, and Y. Li, "Generating question titles for stack overflow from mined code snippets," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, pp. 1–37, 2020.

[36] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proc. 17th Work. Conf. Reverse Eng.*, 2010, pp. 35–44.

[37] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng.*, 2010, pp. 223–226.

[38] P. Rodeghero, C. Liu, P. W. McBurney, and C. McMillan, "An eye-tracking study of java programmers and application to source code summarization," *IEEE Trans. Softw. Eng.*, vol. 41, no. 11, pp. 1038–1054, Nov. 2015.

[39] P. W. Mcburney and C. Mcmillan, "Automatic source code summarization of context for java methods," *IEEE Trans. Softw. Eng.*, vol. 42, no. 2, pp. 103–119, Feb. 2016.

[40] E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of NL-queries for software maintenance and reuse," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 232–242.

[41] B. Wei, "Retrieve and refine: Exemplar-based neural comment generation," in *Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2019, pp. 1250–1252.

[42] R. Cai, Z. Liang, B. Xu, Z. Li, Y. Hao, and Y. Chen, "TAG: Type auxiliary guiding for code comment generation," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 291–301.

[43] K. A. Dawood, K. Y. Sharif, and K. T. Wei, "Source code analysis extractive approach to generate textual summary," *J. Theor. Appl. Inf. Technol.*, vol. 95, no. 21, pp. 5765–5777, 2017.

[44] X. Wang, L. L. Pollock, and K. Vijay-Shanker, "Automatically generating natural language descriptions for object-related statement sequences," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reeng.*, 2017, pp. 205–216.

[45] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proc. 28th Int. Conf. Prog. Comprehension*, 2020, pp. 184–195.

[46] W. Wang *et al.*, "Reinforcement-learning-guided source code summarization via hierarchical attention," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 102–119, Jan. 2022.

[47] L. Büch and A. Andrzejak, "Learning-based recursive aggregation of abstract syntax trees for code clone detection," in *Proc. 26th IEEE Int. Conf. Softw. Anal. Evol. Reeng.*, 2019, pp. 95–104.

[48] J. Svajlenko and C. K. Roy, "Fast, scalable and user-guided clone detection," in *Proc. 40th Int. Conf. Softw. Eng. Companion*, 2018, pp. 352–353.

[49] H. Liu, Z. Yang, Y. Jiang, W. Zhao, and J. Sun, "Enabling clone detection for Ethereum via smart contract birthmarks," in *Proc. 27th Int. Conf. Prog. Comprehension*, 2019, pp. 105–115.

[50] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "TreeGen: A tree-based transformer architecture for code generation," in *Proc. 34th AAAI Conf. Artif. Intell.* 2020, pp. 8984–8991.

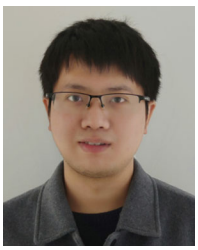
[51] C. Zhang, X. Niu, and B. Yu, "A method of automatic code generation based on AADL model," in *Proc. 2nd Int. Conf. Comput. Sci. Artif. Intell.*, 2018, pp. 180–184.
[52] N. C. Shrikanth and T. Menzies, "Assessing practitioner beliefs about software defect prediction," in *Proc. 42nd Int. Conf. Softw. Eng.: Softw. Eng. Pract.*, 2020, pp. 182–190.
[53] L. Gong, S. Jiang, R. Wang, and L. Jiang, "Empirical evaluation of the impact of class overlap on software defect prediction," in *Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2019, pp. 698–709.
[54] Z. Wang and L. Lu, "A semantic convolutional auto-encoder model for software defect prediction," in *Proc. 32nd Int. Conf. Softw. Eng. Knowl. Eng.*, 2020, pp. 323–328.
[55] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 837–847.
[56] M. Allamanis, E. T. Barr, C. Bird, and C. A. Sutton, "Learning natural coding conventions," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 281–293.
[57] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proc. 30th AAAI Conf. Artif. Intell.*, 2016, pp. 1287–1293.
[58] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics*, 2017, pp. 440–450.

Zheng Li received the B.Sc. degree in the computer science and technology from the Beijing University of Chemical Technology, Beijing, China in 1996, and the Ph.D. degree in computer science from the CREST Centre, King's College London, London, U.K., in 2009.

He is currently a Professor with the College of Information Science and Technology, Beijing University of Chemical Technology. He was a Research Associate with King's College London and University College London, London. He has authored or coauthored more than 60 papers in referred journals or conferences, such as IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, International Conference on Software Engineering, *Journal of Software: Evolution and Process*, *Information and Software Technology*, *Journal of Systems and Software*, International Conference on Software Maintenance, IEEE International Conference on Software Maintenance and Evolution, International Computer Software and Applications Conference, IEEE International Working Conference on Source Code Analysis and Manipulation, and IEEE International Conference on Software Quality, Reliability and Security. His research interests include software engineering, in particular program testing, source code analysis, and manipulation.

Yonghao Wu received the B.S. degree in computer science and technology from Nanchang Hangkong University, Nanchang, China, in 2017, and the M.S. degree in the computer science and technology in 2020 from the Beijing University of Chemical Technology, Beijing, China, where he is currently working toward the Ph.D. degree.

His research interests include fault localization and software testing.



Bin Peng received the B.S. degree in computer science and technology from the Southwest University of Nationalities, Chengdu, China, in 2018. He is currently working toward the master's degree in computer science and technology with the Beijing University of Chemical Technology, Beijing, China.

His research interests include source code analysis and software testing.

Xiang Chen (Member, IEEE) received the B.Sc. degree in information management and system from the School of Management, Xi'an Jiaotong University, Xi'an, China, in 2002, and the M.Sc. and Ph.D. degrees in computer software and theory from Nanjing University, Nanjing, China, in 2008 and 2011, respectively.

He is currently an Associate Professor with the Department of Information Science and Technology, Nantong University, Nantong, China. He has authored or coauthored more than 60 papers in referred journals or conferences, such as IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, *Information and Software Technology*, *Journal of Systems and Software*, IEEE TRANSACTIONS ON RELIABILITY, *Journal of Software: Evolution and Process*, *Software Quality Journal*, *Journal of Computer Science and Technology*, International Conference on Software Engineering, IEEE/ACM International Conference Automated Software Engineering, IEEE International Conference on Software Maintenance and Evolution, IEEE International Conference on Software Analysis, Evolution and Reengineering, and International Computer Software and Applications Conference. His research interests include software engineering, in particular software maintenance and software testing, such as software defect prediction, combinatorial testing, regression testing, and fault localization.

Dr. Chen is a Senior Member of the China Computer Federation and a member of the Association for Computing Machinery.

Zeyu Sun received the B.S. degree in computer science and technology from the Beijing University of Chemical Technology, Beijing, China, in 2017. He is currently working toward the Ph.D. degree with the School of Electronics Engineering and Computer Science, Peking University, Beijing.

His current research interests include code generation, software testing, and deep learning testing.

Yong Liu (Member, IEEE) received the B.Sc. and M.Sc. degrees in computer science and technology and the Ph.D. degree in control science and engineering from the Beijing University of Chemical Technology, Beijing, China, in 2008, 2011, and 2018, respectively.

He is currently an Assistant Professor with the College of Information Science and Technology, Beijing University of Chemical Technology. He has authored or coauthored more than ten papers in referred journals or conferences, such as *Journal of Systems and Software*, *Information Sciences*, IEEE International Conference on Software Quality, Reliability and Security, International Conference on Software Analysis, Testing and Evolution (SATE), and International Computer Software and Applications Conference. His research interests include software engineering, in particular software debugging and software testing, such as source code analysis, mutation testing, and fault localization.

Dr. Liu is a member of the China Computer Federation and the Association for Computing Machinery.

Doyle Paul received the Ph.D. degree in astronomical distributed data processing from the Dublin Institute of Technology, Dublin, Ireland, in 2015.

He is currently the Head of the School of Computer Science, Technological University Dublin, Dublin. He has spent more than 20 years in industry in Silicon Valley, CA, USA and Dublin. He was a Product and Quality Director of CR2, a banking software company, Dublin; a Senior Manager with Sun Microsystems, Menlo Park, CA; and a Senior Developer with a BlueStar Financial Investment. His research areas include big data processing of astronomical images, distributed systems, systems infrastructure, and educational pedagogy.