Improving Code Summarization With Tree Transformer Enhanced by Position-Related Syntax Complement

Jie Song, Zexin Zhang, Zirui Tang, Shi Feng, and Yu Gu

***Abstract*—Code summarization aims to generate natural language (NL) summaries automatically given the source code snippet, which aids developers in understanding source code faster and improves software maintenance. Recent approaches using NL techniques in code summarization fall short of adequately capturing the syntactic characteristics of programming languages (PLs), particularly the position-related syntax, from which the semantics of the source code can be extracted. In this article, we present Syntax transforMer (SyMer) based on the transformer architecture where we enhance it with position-related syntax complement (PSC) to better capture syntactic characteristics. PSC takes advantage of unambiguous relations among code tokens in abstract syntax tree (AST), as well as the gathered attention on crucial code tokens indicated by its syntactic structure. The experimental results demonstrate that SyMer outperforms state-of-the-art models by at least 2.4% bilingual evaluation understudy (BLEU), 1.0% metric for evaluation of translation with explicit ORdering (METEOR) on Java benchmark, and 4.8% (BLEU), 5.1% (METEOR), and 3.2% recall-oriented understudy for gisting evaluation - longest common subsequence (ROUGE-L) on Python benchmark.**

***Impact Statement*—Current code summarization approaches have been focusing on enhancing the performance of code summaries by utilizing syntax information from PLs' ASTs. ASTs are typically fed into neural networks via serialized approaches. However, current serialization approaches neglect the relative position relation between the nodes in the AST, which results in losing syntax information, such as hierarchical and sibling relations between nodes. To overcome the aforementioned limitations, this article introduces a novel method of code summarization that retains the relative position relationships between nodes by incorporating particular tree position embedding to nodes in the serialized AST. Moreover, we propose a tree attention mechanism that enables code tokens to emphasize more on those located at crucial syntactic positions in the AST.**

***Index Terms*—Abstract syntax tree (AST), code summarization, position-related syntax information, programming language (PL) processing, tree transformer.**

Manuscript received 11 September 2023; revised 16 March 2024; accepted 24 April 2024. Date of publication 30 April 2024; date of current version 10 September 2024. This work was supported by the National Natural Science Foundation of China under Grant 62162050 and Grant 62272092. This article was recommended for publication by Associate Editor Xiang Li upon evaluation of the reviewers' comments. *(Corresponding author: Jie Song.)*

Jie Song, Zexin Zhang, and Zirui Tang are with the Software College, Northeastern University, Shenyang, Liaoning 110819, China (e-mail: songjie@mail.neu.edu.cn).

Shi Feng and Yu Gu are with the College of Computer Science and Engineering, Northeastern University, Shenyang, Liaoning 110819, China.

Digital Object Identifier 10.1109/TAI.2024.3395231

I. Introduction

SOURCE code summarization is a task that summarizes a brief natural language (NL) description of the code snippet. It enables programmers to comprehend its purpose without having to read the code. Code summarization increases the comprehensibility of the code, which is essential for lowering the maintenance phase's overhead. Modeling the relationship between NL and programming language (PL) is a fast-growing research area that can save the considerable cost of manually completing this task. Large organizations such as Google, Amazon, and Facebook are increasingly focusing on the code summarization stage of the software process. Similar to machine translation, the code summarization task uses PL as input and NL as output. As a result, neural machine translation (NMT) in natural language processing (NLP) lays the groundwork for this field of study.

Early efforts on code summarization followed NMT to represent code as a sequence of code tokens [1]. The syntaxes are strictly followed by the grammar of PL, which limits the metaphor to NMT. The recent code summarization research has created a solid agreement on using syntactic information to improve code summarization. Most recent works serialize abstract syntax tree (AST) to simultaneously incorporate syntax information and employ sequential neural networks such as LSTM, GRU, and transformer [2]. The methods of serializing the AST include traversing the AST into a single sequence [3] and extracting multiple AST paths [4]. Other works are based on graph or recursive mechanism [5] to recognize the syntactic structure of code.

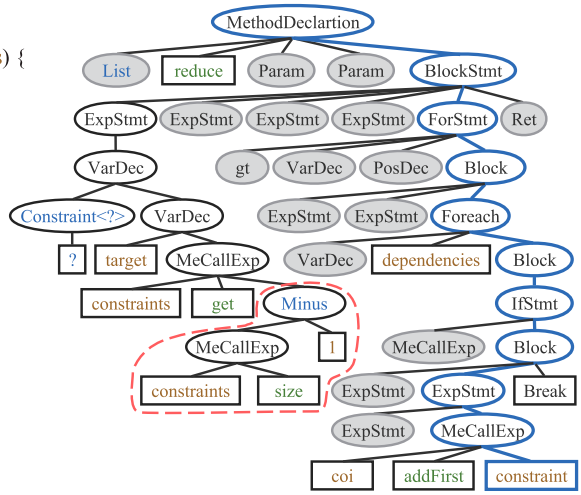
Although data-driven techniques with syntactic complement through various forms of ASTs are continually increasing the efficiency of code summarization, earlier research has failed to leverage PLs' distinguishing properties from NLs fully. Previous syntax complement research ignored the relation between AST nodes and the position-related syntactic significance included in the code token's ancestor nodes. The absence of this syntax complements results in ambiguous function calls and expression confusion. Position-related syntax complement (PSC) is the name of such crucial characteristics, which will be detailed in Section III.

In this article, we propose SyMer to improve the performance of code summarization. SyMer is the short name of

```
private static List<Constraint<?>> reduce(List<Constraint<?>> constraints) {
    Constraint<?> target = constraints.get(constraints.size() - 1);
    Set<Variable<?>> dependencies = getVariables(target);
    LinkedList<Constraint<?>> coi = new LinkedList<Constraint<?>>();
    coi.add(target);
    for (int i = constraints.size() - 2; i >= 0; i--) {
        Constraint<?> constraint = constraints.get(i);
        Set<Variable<?>> variables = getVariables(constraint);
        for (Variable<?> var : dependencies) {
            if (variables.contains(var)) {
                dependencies.addAll(variables);
                coi.addFirst(constraint);
                break;
            }
        }
    }
    return coi;
}
```

MeCallExp Leaf Node

(a)



(b)

```
Reference: apply cone of influence reduction to constraints with respect to the last constraint in the list
CODE-NN: reduce the positions of all edges in the target table
DeepCom: apply reduction to constraints with respect to the specified constraint
CodeBERT: returns the list of constraints
Transformer: apply any constraint to the current list of constraints with respect to the given constraint object
SiT: apply constraint to current list based on given object
BASTS: reduce the desired constraints
SCRIPT: reduce cone of influence reduction to constraints with respect to the constraints
SyMer (our model): apply reduction to the last constraint in the list
```

(c)

Fig. 1. (a) Code sample #2094 in comment generation dataset, which function is "apply cone of influence reduction to constraints with respect to the last constraint in the list." Purple-colored code represents keywords in the PL. Blue-colored code represents library-related names. Green-colored code represents class-related names. Brown-colored code represents variable-related names. (b) AST parsed from the code in (a). Our model can better learn the semantic information of the part circled by red dashed lines than other models. (c) Summaries generated by state-of-the-art models and SyMer, comparing with the referenced summary.

Syntax transforMer. As its name suggests, it enhances the tree transformer by syntax information, which is rifely ignored by existing works. Our novel idea is about PSC. As far as we know, no other work has applied similar structural considerations to encoding the code snippet. We adopt our tree-based attention technique to encode the serialized ASTs within transformer [2]. Fig. 1 shows an example, where Fig. 1(a) shows a code snippet, Fig. 1(b) shows the corresponding AST, and Fig. 1(c) lists the generated summaries of SOTA models and our model, comparing with the referenced summary. The example illustrates that the capabilities of code summarization vary. Our model learns the semantic information of partial AST, circled by red dashed lines in Fig. 1(b), better than others. Section III will further discuss the example. The four main contributions of this study are as follows.

1) We propose a novel tree-based PL-oriented model architecture for improving code summarization by encoding the AST. On average, SyMer achieves state-of-the-art performance on both Java and Python benchmarks.
2) SyMer represents the code snippet as a set of vertical paths in the AST. In addition, SyMer introduces PSC to overcome the limitation of serialized ASTs approaches.
3) SyMer applies to generating summaries in different lengths. The experimental results confirm that our proposed model is effective and outperforms the previous models.
4) The tree-based attention approach presented in this article might serve as a paradigm for models with dependency relation trees as input.

The primary usage of SyMer is saving efforts of writing annotations of the code module. The annotations offer people practical support with source code understanding. Both developers and managers could benefit from the practical usage of SyMer, explained as the following.

1) SyMer assists developers in the source code auto-annotation. These annotations are critical elements of source code documentation, indexing, maintenance, and knowledge extraction. For example, programmers could use NL to search source code and find target functions quickly.
2) SyMer provides schematic source code descriptions in NL for the quality assurance staff, significantly reducing their mental efforts in understanding a large amount of code. The descriptions help verify whether the logic of the code meets the promise, serving as a "walkthrough" tool.
3) SyMer shows a high-level view of the code implementation to project managers, helping them assess the complexity and workload of the project, which is beneficial for resource planning and task allocation.

The rest of this article is organized as follows. Section II discusses the related work. Section III presents the motivation. Section IV illustrates the design of SyMer. We elaborate on the

TABLE I
DIFFERENT MODEL STRUCTURE OF EXISTING WORKS RELATED TO CODE SUMMARIZATION

Approach	RNN	Tree/GNN	Transformer
CODE-NN [1]	✓		
DeepCom [3]	✓		
Code2Seq [4]	✓		
CoaCor [7]	✓		
Extended-Tree-LSTM [5]	✓	Tree	
Transformer [9]			✓
CodeBERT [10]			✓
GraphCodeBERT [11]		GNN	✓
SiT [12]			✓
BASTS [13]			✓
SCRIPT [14]			✓
SG-Trans [15]			✓
SyMer (our model)		Tree	✓

design and results of the experiment through four research questions in Section V. Finally, we conclude our work in Section VI.

II. RELATED WORKS

Code summarization is a special case of code representation learning which has been studied in recent years. The recent works are mainly based on RNN and transformer, as shown in Table I. SyMer innovatively employs a tree-transformer architecture to generate code summarization. In this section, we first discuss the related work on code summarization and then extend the scope to the "transformer on program representation learning."

A. Source Code Summarization

Source code summarization facilitates software developers to understand the code's purpose quickly. Automatic code summarization is a rapidly expanding research area. Promoted by the development of neural networks, data-driven strategies [4], [6], [7] replaced the previous heuristic approaches derived from empirical research [8] as the mainstream method of code summarization. Conventional works adopt template-based approaches [8]. These works use the information retrieval (IR) technique and latent semantic indexing to choose and summarize the most relevant words from the source code.

Advances in data-driven approaches in NLP have lately enabled PL processing to reach outstanding results in source code summarization. In contrast to NLP research, directly feeding the source code snippet into the deep neural network as a code token sequence underutilizes the syntactic information. Most recent PL-related works based on deep learning technology utilized syntax information in AST to help downstream tasks, such as code summarization [4], [6], code generation [16], and code completion [17]. These works utilizing syntax information in AST can be generally divided into two categories: RNN-based and transformer-based approaches.

1) RNN-Based Approaches: InferCode [18] perform tree-based convolutional neural networks on ASTs to learn source code representation. Shido et al. [5] used a recursive approach to introduce the tree structure of ASTs. This mechanism is not parallelizable, which limits its applicability to large-scale datasets and models due to the lengthy training and inference times. DeepCom [3] serialized ASTs with structure-based traversal (SBT). It considers the syntactic structure not included in the code token sequence. Code2seq [4] proposed a code summarization architecture that encodes AST pairwise paths between terminals. The encoder component of Code2seq architecture represents the pairwise paths between random sample terminals, with each path encoded to a fixed-length vector utilizing BiLSTM [19]. Code2seq also shows that similarities between different implementations that achieve the same function can be captured in AST paths. Extracting numerous paths in the AST [4], Kim et al. [20] alleviated the loss of a single path to the hierarchical structure information in the AST.

2) Transformer-Based Approaches: While transformer gained prominence in the NMT domain, Ahmad et al. [9] proposed a transformer-based model incorporating copy attention and relative position embedding. SiT [12] integrated a multiview graph matrix into self-attention and improved the performance by masking redundant attention when calculating attention scores. SG-Trans [15] injected both local semantic information and global syntactic structure into the transformer module as an inductive bias to better capture the hierarchical features of the code. SCRIPT [14] captured the structural relative dependencies through structural relative positions. It first adopted two types of transformer encoders to encode the position information among tokens and the direct inputs, respectively. Then, it stacked the information to learn representations of source code.

These models with nonsequential input use multiple methods to further obtain the local and global information of token relations or AST structure, but ignore the relation between AST nodes and the information included in the code token's ancestor nodes, which is called PSC in this article. Using PSC can significantly improve the performance of the model.

B. Transformer on Program Representation Learning

In the NLP research field, the transformer architecture, as a Seq2Seq model [21], has achieved state-of-the-art performance in various tasks and proven its robust feature learning capabilities in long-term dependencies capturing. The self-attention layer in each transformer block enables the architecture to calculate the entire sequence without recursion in the encoder. Transformer has recently been applied to nonsequential inputs [22], [23] in NLP, such as improving the machine translation using a tree-base model [24].

This breakthrough has also sparked interest in using transformer in PL-related research with nonsequential input. Harer et al. [25] proposed a Tree Convolution Block to enable the transformer to handle tree-structured data for code correction. Feng et al. [10] developed CodeBERT, a pretrained model on bimodal data of NL-PL pairs and supports downstream NL-PL applications. Guo et al. [11] proposed GraphCodeBERT to integrate the code's data flow. BASTS [13] splits the source code and generates ASTs based on the blocks in the dominator tree of the control flow graph. Then, it combines the code encoding with the split AST encoding generated by a TreeLSTM based

on a pretraining strategy. It feeds the encoding combination into a transformer to guide code summarization.

SyMer innovatively uses the tree-transformer structure combined with PSC. This unique approach captures syntactic characteristics better and leads to superior performance compared to others.

III. Motivation

This section begins with an example to describe our motivations for SyMer. We provide an intuitive explanation of what PSC is and how it benefits code summarization.

Consider the code snippet in Fig. 1(a) whose reference summary is "apply cone of influence reduction to constraints concerning the last constraint in the list." We list the AST of the code snippet, the summaries of the baselines, and our model.

A. Why Do We Need PSC?

1) Gathered Attention: The token "constraint" appears in the summary provided by all models. It implies that the models account for the importance of the code token constraint. The line coi.addFirst (constraint) is the last operation statement of the return variable coi and the expression in the block of the dual loop. The semantic significance of the code token constraint exceeds other code tokens since it is the function call parameter and the relevant variable of the outer loop.

This structural observation demonstrates that the code token's syntactic information in the AST could deduce its significance to the code summarization. In the example of Fig. 1(b), the blue vertical path in the AST reflects part of the position-related syntactic information of code token constraint. According to the syntactic structure, the model should make code representation more prone to pay attention to code tokens in crucial positions. As a result, we claim that the syntactic information in the vertical path could aid the model in recognizing the relevance of various code tokens.

2) Unambiguous Relations: Only SyMer's summary contains the token "last" compared with summaries of other models. Other models miss this semantic information since the token "last" is never used in the code snippet. For the semantic information of "last," the code statement constraints.get(constraints.size() - 1) is critical. The syntactic information in this expression is reflected in the AST as the section encompassed by the red dotted line, particularly the sibling and hierarchical relationship between the code tokens constraints, size, and 1. Although many PL-oriented models take syntactic information from the AST into account, they lose the sequential relations between code tokens when serializing the AST.

B. What Are the Aims of Proposing PSC?

In combination with the above, to better capture the syntactic structure of the source code while encoding it, we emphasize syntactic characteristics that were overlooked by previous models but are essential for code summarization:

1) *Gathered Attention* on crucial code tokens. The code token's syntax provides position-related information that might impact its significance in the code snippet. More specifically, the effect of the terminal node on code representation should be influenced by the nonterminal node in the AST.
2) *Unambiguous Relations* among code tokens. Hierarchical and sibling relations among code tokens allow the model to capture additional semantic information from the code snippet.

The proposed PSC introduces the above characteristics.

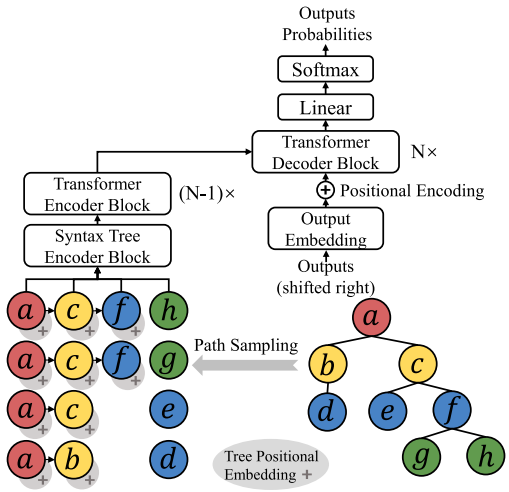


Fig. 2. Our model architecture. The same color in the AST indicates that the node has the same hierarchical position. Red, yellow, blue, and green represent the first, second, third, and fourth layer, respectively. Tree positional embedding is used when path sampling and syntax tree encoder block is used before the normal transformer encoder block.

IV. Enhanced Model

We update the encoder component of transformer to adapt it to tree structure input, which is seen in Fig. 2. In this section, we employ syntax-tree attention and tree positional embedding to correlate to PSC's gathered attention and unambiguous relations. Our model accepts the vertical path set serialized from the AST as input. We use distinct learnable embedding matrices to represent terminals and nonterminals, respectively. After that, for each nonterminal node, SyMer adds the matching tree positional embedding (the shadow at the bottom right of the nonterminals in Fig. 2). Finally, SyMer provides the model with the final representation of each vertical path. This section explains the components of SyMer in the above order.

A. Code Representation

Given a code snippet, SyMer serializes the AST by extracting the vertical paths from the root to the terminals, as shown in Fig. 2. This section details how SyMer derives the node and syntactic vector representations for each vertical path.

The AST is a hierarchical representation of program structure based on the grammar of the PL. Leaf nodes of the tree are labeled with *terminals*, which correspond to code tokens of the source code. Internal nodes are labeled with *nonterminals*, corresponding to the hierarchical structure of terminals. We

give the following definitions. The AST created from code snippet C as

$$\mathcal{F}(C) = (\mathcal{N}, \mathcal{L}, \mathcal{R}, \mathcal{D}) \tag{1}$$

where $\mathcal{L}$ denotes the set of terminals of the AST. $\mathcal{N}$ denotes the set of nonterminals. $\mathcal{R}(x)$ represents the set of ancestors of the node x, in which x could be terminals or nonterminals. $\mathcal{D}(x^{\mathcal{N}})$ represents the set of descendant nodes of a nonterminal node $x^{\mathcal{N}}$.

We offer a concept to identify nonterminals on distinct vertical paths since the same nonterminal node in different AST vertical paths carries different syntactic information for the associated terminals. For $\mathcal{L} = (x_1^{\mathcal{L}}, ..., x_n^{\mathcal{L}})$, the nonterminals in the vertical path of terminal node $x_i^{\mathcal{L}}$ will be $\mathcal{R}(x_i^{\mathcal{L}}) = (x_{i,1}^{\mathcal{N}} x_{i,2}^{\mathcal{N}}, ..., x_{i,m}^{\mathcal{N}})$ suppose $\mid \mathcal{R}(x_i^{\mathcal{L}}) \mid = m$. We define $\mathcal{R}^j(x_i^{\mathcal{L}}) = x_{i,j}^{\mathcal{N}}$ for the convenience of reading as follows.

Formally, SyMer represents C as a set of vertical AST paths

$$C = (x_{1,1}^{\mathcal{N}} x_{1,2}^{\mathcal{N}}, ..., x_1^{\mathcal{L}}), ..., (x_{n,1}^{\mathcal{N}} x_{n,2}^{\mathcal{N}}, ..., x_n^{\mathcal{L}}). \tag{2}$$

SyMer includes the type of the terminals between them and their parent nodes while serializing. It is beneficial for the model to capture the semantics of code tokens better. However, Fig. 2 and equations do not show this operation for abbreviation.

Example. In Fig. 2, $\mathcal{L} = (d, e, g, h)$, $\mathcal{R}(g) = (a, c, f)$, and $\mathcal{R}^3(g) = x_{3,3}^{\mathcal{N}} = f$. SyMer serializes the AST as $\{(abd), (ace), (acfg), (acfh)\}$.

1) Node Representation: The terminals and the nonterminals have different representations to distinguish them.

For encoding a nonterminal node, we use a learnable embedding matrix E^{nonterm}

$$u_{i,j} = E^{\text{nonterm}}_{x_{i,j}^{\mathcal{N}}} \in \mathbb{R}^d. \tag{3}$$

For encoding terminal nodes whose values are code tokens, we divide the code token into subtokens according to the camelcase and then represent each subtoken with a learnable embedding matrix $E^{\text{subtokens}}$. We aggregate the vector representations of each subtoken to encode the entire terminal node

$$w_i = \sum_{s \in \text{split}(x^{\mathcal{L}})} E^{\text{subtokens}}_s \in \mathbb{R}^d. \tag{4}$$

Example. The code token `HashSet` is decomposed into `Hash` and `Set`. The vector representation of `HashSet` is $E^{\text{subtokens}}_{\text{Hash}} + E^{\text{subtokens}}_{\text{Set}}$.

2) Syntactic Representation: The nonterminals in a vertical path collectively constitute the syntactic positional information of the corresponding terminal node. Based on this purpose, SyMer employs nonterminals on the AST vertical path to make the model pay more attention to the code tokens in the crucial position. SyMer projects the concatenated vector of the nonterminals to produce the syntactic vector representation of the vertical path $(x_{i,1}^{\mathcal{N}} x_{i,2}^{\mathcal{N}}, ..., x_{i,m}^{\mathcal{N}})$

$$z_i = W_{\text{path}} \Big\|_{j=1}^{m} u_{i,j} \tag{5}$$

where $\|$ denotes the concatenation operation between hidden dimensions, $W_{\text{path}} \in \mathbb{R}^{m \cdot d \times d}$ is projection matrix.

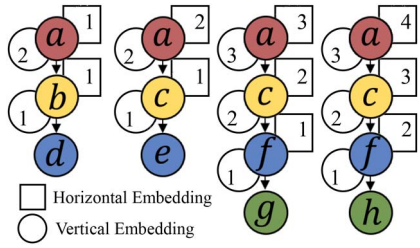


Fig. 3. Example of tree positional embedding on the vertical path of Fig. 2. The rectangle at the top right represents the tree positional embedding index in the horizontal direction, and the circle at the bottom left represents the tree positional embedding index in the vertical direction. The integer in the ellipse represents the $\mid \mathcal{V}(i,j) \mid$ or $\mid \mathcal{H}(i,j) \mid$. The node colors have the same meaning as Fig. 2.

B. Tree Positional Embedding

Transformer's positional encoding addresses the absent ability of the self-attention mechanism. It sequentially adds positional word information regarding NLP-related tasks. Extending the sequence structure to the tree structure also requires a technique to introduce the node's positional information. Inspired by Hierarchical Embedding [22], a tree positional embedding injects vertical and horizontal positional information for each nonterminal node in SyMer. It utilizes two learnable embedding matrices E^v and E^h to represent vertical and horizontal direction positional information

$$\begin{cases} p_{i,j} = E^v_{|\mathcal{V}(i,j)|} + E^h_{|\mathcal{H}(i,j)|} \\ \mathcal{V}(i,j) = \mathcal{D}(x_{i,j}^{\mathcal{N}}) \\ \mathcal{H}(i,j) = \{x_k^{\mathcal{L}} \mid k \le i \text{ and } x_k^{\mathcal{L}} \in \mathcal{D}(x_{i,j}^{\mathcal{N}}) \cap \mathcal{L}\} \end{cases} \tag{6}$$

where $\mathcal{V}(i,j)$ indicates the ancestor node set of the node $x_{i,j}^{\mathcal{N}}$, and $\mathcal{H}(i,j)$ indicates the set of terminals to the left of $x_i^{\mathcal{L}}$ on the subtree rooted at $\mathcal{R}^j(x_i^{\mathcal{L}})$.

Previous AST serializations suffer the drawback of failing to capture syntactic structure comprehensively. The nodes on the serialized ASTs may only reflect their relative positional relation in the current serialized sequence, rather than the ASTs. As mentioned in Section III, unambiguous positional relations among code tokens are crucial for syntactic and semantic code analysis. For example, expressions and statements in loop statements, arguments, and function bodies may be misunderstood if positional relations among code tokens are confused. It is why SyMer separates the positional information of the nonterminals in the tree into vertical and horizontal positional information. Horizontal positional information distinguishes the same nonterminal node on different AST paths, whereas vertical positional information introduces the node's order on the associated AST path. After tree positional embedding, the syntactic vector representation of the vertical path $(x_{i,1}^{\mathcal{N}} x_{i,2}^{\mathcal{N}}, ..., x_{i,m}^{\mathcal{N}})$ is defined as

$$\tilde{z}_i = W_{\text{path}} \Big\|_{j=1}^{m} [u_{i,j} + p_{i,j}]. \tag{7}$$

Example. Fig. 3 demonstrates how tree positional embedding establishes unambiguous relations among code tokens.

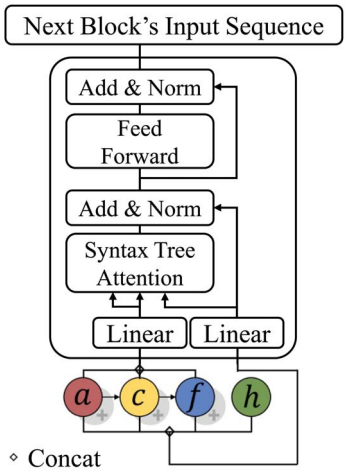


Fig. 4. Syntax-tree encoder block. Diamonds represent the concatenation operation of vectors. Compared to the normal transformer encoder, multihead attention is replaced by syntax-tree attention, and linear is used before it. The node colors have the same meaning as Fig. 2.

A tree structure analysis reveals that g and h share the same parent node f, making their respective vertical paths identical except for themselves. As a result, explicitly encoding the vertical path of nodes g, h confuses their sibling relation order. The differing horizontal positional embedding indexes $\mid \mathcal{H}(i,j) \mid$ of the nodes g, h identify the two vertical paths.

C. Syntax-Tree Attention

The standard self-attention mechanism fails to encode tree structure. Therefore, SyMer replaces the input of "multihead attention" in native transformer with the proposed syntax-tree encoder block, and renames it as "syntax-tree attention," as shown in Fig. 4

$$Q_{Si} = \tilde{z}W_i^Q, K_{Si} = \tilde{z}W_i^K$$
$$V_{Si} = (\tilde{z}; w)W_i^V \tag{8}$$

$$\text{head}_i = \text{softmax}\left(\frac{Q_{Si}K_{Si}^T}{\sqrt{d_k}}\right)V_{Si} \tag{9}$$

where $W_i^Q, W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ and $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ are projection matrices for head i. The syntax-tree attention projects the syntactic representation with tree positional embedding into Q_S and K_S to jointly compute the attention score. To avoid the situation that the syntactic information disappears after computing the attention score, SyMer leverages projection matrix W^V to project the concatenation of the syntactic representation $\tilde{z}$ and terminals representation w

$$\text{SyntaxTreeAttention}(Q_S, K_S, V_S) = \left(\Big\Vert_{i=1}^{h} \text{head}_i\right)W^O \tag{10}$$

where h is the number of heads and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$ is a learnable project matrices. The output of the syntax-tree encoder is conveyed to the next standard transformer encoder.

V. Experiments

A. Purposes

In this section, we evaluate SyMer from different aspects following these research questions (RQs).

1) How does our approach perform compared to the baselines?
2) What role does each component have in the performance of SyMer?
3) Whether the different characteristics of the dataset have an obvious correlation with the model performance?
4) Whether the PSC contains the syntactic and semantic complement as preset?

B. Setup

1) Experiment Environment: We trained SyMer on a machine with two Nvidia GeForce GTX 1080Ti GPUs, Intel(R) Core(TM) i9-9900K CPU @ 3.60 GHz, 32 GB main memory.

2) Model Architecture: Fig. 2 shows the overall architecture of SyMer. The special syntax-tree encoder block takes the vertical path set serialized from the AST as input, followed by N-1 transformer encoder blocks. We use the original transformer decoder with mask multihead attention and cross attention. Finally, a fully connected layer and a softmax function output the predicted probabilities.

3) Parameter Settings: SyMer is composed of six encoder layers, six decoder layers, and eight heads in its transformer architecture and its hidden size is 512. For updating our model's parameters, SyMer adopts Adam optimizer [26] with $\beta_1 = 0.9, \beta_2 = 0.98, \epsilon = 10^{-9}$ and warm-ups the learning rate according to the formula the same as the transformer. With the batch size set to 128 and the dropout rate set to 0.1 during the training, we train our model for at most 100 epochs and select the checkpoint with the best performance on the validation. SyMer also employs label smoothing of value $\varepsilon = 0.1$ to avoid overfitting.

4) Training Methodology: We apply the gradient accumulation method to our training to accelerate training speed and improve computational performance. For one batch, we do n times forward propagation and one back propagation to update parameters. Layerwise learning rate decay (LLRD) [27] is a technique that lets lower layers encode more general information, and higher layers capture more specific information. LLRD makes our model's performance more stable. To avoid overfitting, we stop the training early if the performance on the validation set does not increase for ten epochs.

5) PL Selection: PLs are either scripting or compiled languages. The theoretical difference between the two is that scripting languages do not require the compilation step and are rather interpreted. On one hand, due to their execution mechanism and programming scenario, the scripting languages have similar keywords and programming styles. Among the popular scripting languages, such as Python, JavaScript, Ruby, and Perl, Python is a shining star. On the other hand, most popular compiled languages, such as Java, C++, and C#, are object-oriented PLs with similar programming styles, of

TABLE II
COMPARATIVE BLEU, ROUGE-L, AND METEOR SCORES FOR SYMER AND THE BASELINES

Model	Java			Python		
	BLEU	**METEOR**	**ROUGE-L**	**BLEU**	**METEOR**	**ROUGE-L**
CODE-NN [1]	27.60	12.61	41.10	17.36	09.29	37.81
DeepCom [3]	39.75	23.06	52.67	20.78	09.98	37.35
CodeBERT [10]	43.33	26.20	54.64	33.47	21.69	49.35
Transformer [9]	44.58	26.43	54.76	32.52	19.77	46.73
SiT [12]	45.76	27.58	55.58	34.11	21.11	48.35
BASTS [13]	46.25	28.33	55.96	34.05	21.83	50.20
SG-Trans [15]	45.89	27.85	55.79	33.04	20.52	47.01
SCRIPT [14]	46.89	28.48	**56.69**	34.00	20.84	48.15
SyMer	**48.05**(↑1.16)	**29.14**(↑0.66)	56.60	**35.76**(↑1.65)	**22.96**(↑1.13)	**51.84**(↑1.64)

Note: Best results are highlighted in bold, while the brackets include improvement percentages for SyMer relative to the best baseline are included in brackets.

which Java is the most representative. Therefore, we choose Python and Java as evaluation languages as other related works do [14], [15].

6) Dataset and Preprocessing: To demonstrate SyMer's effectiveness in generating summaries with various PLs. Our experiments are conducted on two benchmarks of Java [3] and Python [6]. We aligned with their training, testing, and validation divisions. We released our code, including source code and dataset, on GitHub [28].

7) Evaluation Metrics: BLEU [29], METEOR [30], and ROUGE [31] score are three metrics to evaluate the effectiveness of comment generation. Bilingual evaluation understudy (BLEU) measures the *n*-gram precision between the generated summary and reference with a brevity penalty for short sentences. The formula to compute BLEU-N(1/2/3/4) is as follows:

$$\text{BLEU-N} = \text{BP} \cdot \exp \sum_{n=1}^{N} \omega_n \log p_n \tag{11}$$

where p_n denotes the *n*-gram precision score between the generated sentences and references, and ω_n is the uniform weight $1/N$. BP is the brevity penalty. We adopt S-BLEU, which indicates the average sentence-level BLEU score.

Recall-oriented understudy for gisting evaluation - longest common subsequence (ROUGE-L) compares the generated summary and reference based on the longest common subsequence (LCS). For generated sentence A of length m and reference B of length n, we adopt the F1 score for ROUGE-L, which is the harmonic mean of precision and recall

$$P_L = \frac{\text{LCS}(A,B)}{m}, R_L = \frac{\text{LCS}(A,B)}{n}, F_L = \frac{(1+\beta^2)P_L R_L}{\beta^2 P_L + R_L}. \tag{12}$$

Metric for evaluation of translation with explicit ORdering (METEOR) calculates the similarity scores by aligning the generated sentences and references. Consequently, METEOR metric is more relevant to manual discrimination

$$\text{METEOR} = (1 - \gamma \cdot \text{frag}^{\beta}) \frac{PR}{\alpha P + (1-\alpha)R} \tag{13}$$

where frag is a fragmentation fraction, P and R are the unigram precision and unigram recall, and α, β, and γ are penalty parameters with default values of 0.9, 3.0, and 0.5, respectively.

C. RQ_1: Effectiveness Comparison

In RQ_1, we compare SyMer with several state-of-the-art methods as follows.

1) *CODE-NN* [1] is a traditional encoder–decoder architecture in NMT that encodes code snippets and generates summaries in the decodes with the attention mechanism.
2) *DeepCom* [3] uses SBT to serialize AST. It generates summaries using an attention mechanism based on SBT path and code token sequence.
3) *CodeBERT* [10] trains the model at masked language modeling (MLM) and replaced token detection (RTD) tasks and takes into account the NL-PL pairs as multimodal input. This approach is representative of the pretrained model designed for PL-related tasks specially.
4) *Transformer* [9] uses the transformer model [2] for the code summarization task.
5) *SiT* [12] introduces a structure-induced self-attention mechanism to capture the information from AST. It transforms the AST into adjacency matrices of three views and combines them to represent the code snippet.
6) *BASTS* [13] splits the given code snippet based on the blocks in the dominator tree of the control flow graph. BASTS uses TreeLSTM to encode the code snippet based on the ASTs of each code split, and feds the combination of syntax encoding and code encoding into transformer to generate summaries.
7) *SG-Trans* [15] injects local semantic information and global syntactic structure into the self-attentive module in the transformer as an inductive bias to better capture the hierarchical features of source code.
8) *SCRIPT* [14] operates two types of transformer encoders to encode the information of structural relative positions among the tokens and the direct input, respectively, to capture the structural relative dependencies.

Table II shows the generated results, namely the NL summaries for Java and Python benchmarks. SyMer outperforms the baselines for all evaluation metrics except ROUGLE-L score for Java benchmarks, demonstrating that PSC facilitates the model to capture additional syntactical context. For Java benchmark, SyMer improves by 1.16 than the previous best BLEU score achieved by SCRIPT. SyMer outperforms SCRIPT with the

highest METEOR score by 0.21 points, and the ROUGLE-L score is slightly lower than SCRIPT. For Python benchmark, SyMer achieves higher relative improvement, improving by 1.65, 1.13, and 1.64 points on BLEU, METEOR, and ROUGE-L, respectively. Out-of-vocabulary (OOV) is more frequent in Python benchmarks than in Java, leading to poor baseline performance. SyMer's gathered attention mechanism reduces the impact of the OOV token by using its syntactic position in calculating attention rather than its representation.

Baselines with LSTM as the backbone model [1], Hu et al. [3] significantly underperform other baselines because of their difficulty in capturing long-term dependencies. Due to its reliance solely on the code token embedding and disregard for the syntactic structure of the code, CODE-NN receives the lowest grade. Similarly, DeepCom's performance is also lower than other baselines, indicating that serialized ASTs without extra context gathered through traversal cannot completely reflect the correlation between code tokens. However, BASTS scores much higher than the other baselines except SCRIPT due to their additional context in using ASTs to represent source code. Compared with BASTS, SyMer, which considers unambiguous relations among code tokens and uses gathered attention to reduce the impact of OOV, achieves better performance. The latest state-of-the-art model SCRIPT considers information related to the shortest path between AST nodes, but neglected the horizontal and vertical information in AST, which makes SyMer achieve better performance.

D. RQ_2: Ablation Analysis

Since the original transformer architecture in RQ_1 has demonstrated that SyMer's outperformance comes from PSC. In this experiment, we prefer to know how important each proposed component contributes to performance improvement. The experiment removes several components or mechanisms of SyMer, given as follows.

1) Without-tree-positional-embedding: SyMer adopts nodes on AST where content is without its tree positional embedding.
2) Vertical-embedding-only: SyMer only keeps the vertical trainable matrix in tree positional embedding.
3) Horizontal-embedding-only: SyMer only keeps the horizontal trainable matrix in tree positional embedding.
4) Without-syntax-tree-attention: SyMer only uses the standard transformer encoder block.
5) Without-terminals-type: SyMer does not add the token's type above their values on AST.

Table III shows the performance of each alternate model. Tree positional embedding heavily influences the performance of SyMer. When it is removed, BLEU and ROUGE-L decrease to 5.26 and 4.33, respectively. The disappearance of the tree positional embedding causes the degradation of the order information of the AST path. After that, we individually analyze the influence of vertical and horizontal positional embedding on the model performance.

The vertical-embedding-only case obfuscates the same node on different path sequences. The horizontal-embedding-only case makes SyMer not treat the path as a sequence. Table III shows that horizontal embedding affects performance more than vertical embedding. After removing the horizontal embedding, SyMer loses the ability to distinguish the correlation between code tokens, which is essential for PSC.

TABLE III
VARIATIONS OF SYMER, PERFORMED ON JAVA BENCHMARK

Model	BLEU	METEOR	ROUGE-L
SyMer(base)	**48.05**	**29.14**	**56.60**
Without-tree-positional-embedding	42.79	26.90	52.27
Vertical-embedding-only	44.58	25.90	53.60
Horizontal-embedding-only	45.03	28.75	55.52
Without-syntax-tree-attention	43.84	25.14	53.46
Without-terminals-type	46.56	28.32	56.07

Note: The bolded entry represent the one with the best performance among the compared models, that is our model without removing any modules.

Retaining the type of terminals (code tokens) impacts model performance less. The alternative without-syntax-tree-attention case employs the V_{Si} in (8) as the vector before being projected by W^Q, W^K, W^V in "multihead attention." The results show that the BLEU decreases to 4.21 as the attention mechanism, derived from the code token syntax structure, is removed. Such a phenomenon also confirms that SyMer pays more attention to the crucial code token under the PSC enhancement; as a result, it generalizes to the code snippets not contained in the training set.

The experiment collects evidence that PSC exerts its effect through the modules designed in SyMer. We present a qualitative analysis to show how Gathered Attention influences SyMer's performance improvement.

E. RQ_3: Parameter Analysis

The experiment investigates the impact of comment length and code length on comment generation quality, as the comment length may influence the evaluation of the summary, and the code length may influence representation learning. We directly divide the test dataset into the code length and the comment length since the comment length and the code length are unrelated. According to the statistics, we further limited both of them to the interval where the sample size has statistical significance.

Figs. 5 and 6 investigate how the performance of each model varies with the code length or comment length in the Java dataset. To inform the following analysis, Fig. 7(a) and 7(b) shows the specific statistics of the samples with different comment lengths and code lengths in the test dataset. According to Fig. 5(a)–5(c), the results reveal that each model's overall performance decreases with the increasing of comment length on all three metrics, while SyMer outperforms the baselines in the majority of circumstances. Especially on the BLEU and METEOR metrics, all models perform poorly when the comment length was too short or too long. When the comment length is between 20 and 25, all the models have a peak performance, and we speculate that it is caused by data noise. SyMer performs best when the comment length is relatively short, especially in terms of the BLEU metric. At longer comment lengths, each model shows significant performance fluctuations, which can be deduced from Fig. 7 because the number of samples decreases significantly as the comment length grows.

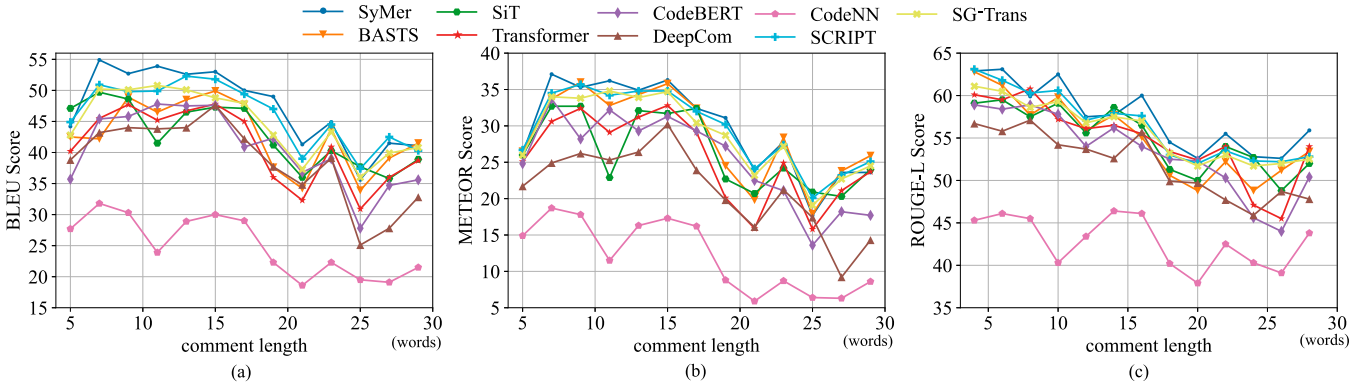


Fig. 5. Performance of each model varies with the comment length in the Java dataset. (a), (b), and (c) Subplots correspond to the BLEU, METEOR, and ROUGE-L metrics, respectively.

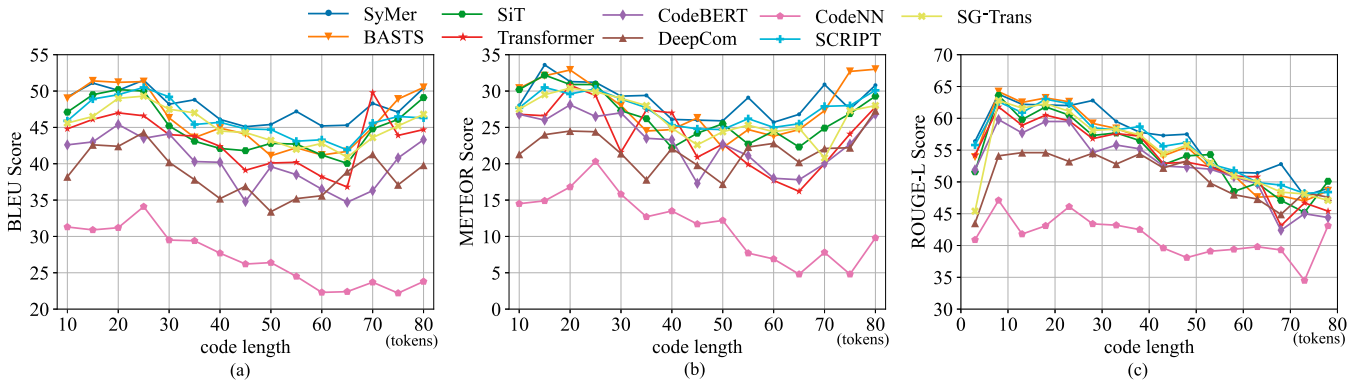


Fig. 6. Performance of each model varies with the code length in the Java dataset. (a), (b), and (c) Subplots correspond to the BLEU, METEOR, and ROUGE-L metrics, respectively.

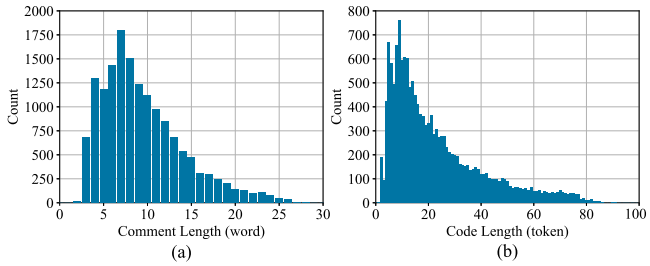


Fig. 7. (a) Statistics of test dataset at comment length. (b) Statistics of test dataset at code length.

```
Example Source Code
void write(Encoder encoder, Collection<T> value) {
        encoder.writeInt(value.size());
        for (T t : value) {
                entrySerializer.write(encoder, t);
        }
}
```

H5,7 H7 H2,3,6 H4

Fig. 8. Example Java code snippet in the dataset of Code [4]. Its function is to serialize a collection object. The code colors have the same meaning as Fig. 1(a).

Compared to the comment length, Fig. 6(a)–6(c) shows that the code length impacts SyMer's performance less than the comment length. SyMer is less sensitive to the code length parameter than the baselines, especially in terms of the BLEU metric. Furthermore, there is a noticeable decrease in code length between 35 and 55. We attribute this to the noise in the dataset, due to all models performing poorly in this range.

F. RQ4: Qualitative Analysis

To visually demonstrate that Gathered Attention does contribute to the code summarization, we chose a method name generation task [4] because its generation content is shorter than the two benchmarks of Section V-C so it is easier to observe the distribution of attention. The data are from three-scale datasets made up of various open-source Java projects. The code snippet in Fig. 8 is used in the example, along with the method name generation task. The code snippet is from Gradle [32]. SyMer correctly generated the target method name "write." Fig. 9(a)–9(h) visualizes a snapshot of the multihead syntax-tree attention and the depth of color represents the magnitude of attention score among tokens ranging from 0 to 1. PSC has a considerable impact on SyMer's performance for the following reasons. First, separate multihead syntax-tree attention attends to the information of different representation subspaces. Second, the vertical path of code token *write* is clearly attended to by more than one attention head, such as head 2, head 3, and head 6. Since the code token *write* matches the method name reference, it is evident that the gathered attention object is as

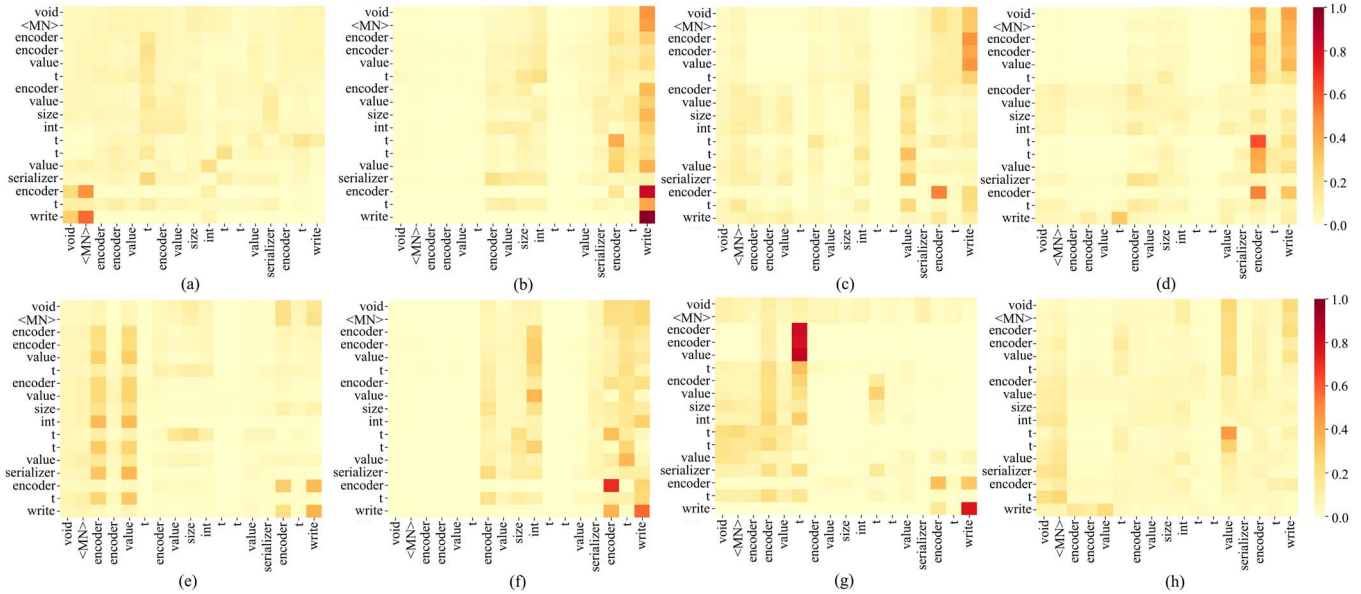


Fig. 9. Visualization of syntax-tree attention of Fig. 8 on method name generation task. <MN> denotes a special token used in order not to expose the method name. The darker the color, the larger the value. (a)–(h) Eight heads in multihead attention, respectively.

expected. Third, the remaining attention heads, such as head 5 and head 7, focus on the parameters of the code method, demonstrating that various attention heads pay attention to distinct syntactic representation subspaces.

Alternatively in Fig. 9, an apparent phenomenon is that attention focuses attention on critical code tokens. The attention distribution identifies the semantic information in the code snippet, thanks to the syntax-tree attention. This phenomenon is consistent with the PSC's initial concern, which is "Gathered Attention on crucial code tokens." Gathered attention can capture coding information at a finer granularity than distracted attention. In the example, the attention gathered on code token *write* demonstrates that SyMer detects crucial code tokens using syntactic information. On the other hand, SyMer learns to use the code token *write* directly from the source code. As a result, SyMer generates a summary that includes the code token *write*. Taking into account the above example, as well as the gathered attention and the performance degradation of the without-syntax-attention model in RQ_2, we conclude that SyMer learns syntactic and semantic information from PSC as preset.

VI. Conclusion

In this article, we studied the problem of code summarization and proposed a novel tree-based PL-oriented model called SyMer. SyMer serializes the AST as a collection of the corresponding vertical path. The main challenges are to handle the structural information loss of serialized ASTs and stress the crucial characteristics of source code. To overcome these challenges, SyMer emphasizes syntactic attention related to the code token position and the unambiguous relation between code tokens. SyMer incorporates a novel syntax-tree attention into the transformer, allowing it to prioritize crucial code tokens based on syntactic importance. Our experiments results demonstrate that SyMer outperforms previous models by at least 2.4% (BLEU), 1.0% (METEOR) on Java benchmark and 4.8% (BLEU), 5.1% (METEOR), and 3.2% (ROUGE-L) on Python benchmark.

Our work can be extended in multiple directions. We believe that the gathered attention and unambiguous relations proposed for PSC can serve as the foundation for various PL-related tasks. Syntax-tree attention, as proposed in this research, could be used as a component for models that accept dependency relation trees as input. For future work, we intend to investigate how to incorporate languages with less AST properties into our tree-based model and make better use of the relation among function calls to build up more accurate code summarization for general applications.

References

[1] 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 (Vol. 1: Long Papers)*, Berlin, Germany. Stroudsburg, PA, USA: Association for Computational Linguistics, 2016, pp. 2073–2083.

[2] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Red Hook, NY, USA: Curran Associates, Inc., 2017, pp. 6000–6010.

[3] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. 26th Conf. Program Comprehension*, New York, NY, USA: ACM, 2018, pp. 200–210.

[4] U. Alon, S. Brody, O. Levy, and E. Yahav, "Code2Seq: Generating sequences from structured representations of code," in *Proc. 7th Int. Conf. Learn. Representations (ICLR)*, New Orleans, LA, USA, May 2019, pp. 6969–6990. [Online]. Available: OpenReview.net

[5] Y. Shido, Y. Kobayashi, A. Yamamoto, A. Miyamoto, and T. Matsumura, "Automatic source code summarization with extended Tree-LSTM," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Budapest, Hungary. Piscataway, NJ, USA: IEEE Press, Jul. 2019, pp. 1–8.

[6] Y. Wan et al., "Improving automatic source code summarization via deep reinforcement learning," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, Montpellier, France. New York, NY, USA: ACM, Sep. 2018, pp. 397–407.

[7] Z. Yao, J. R. Peddamail, and H. Sun, "CoaCor: Code annotation for code retrieval with reinforcement learning," in *Proc. World Wide Web Conf. (WWW '19)*, San Francisco, CA, USA. New York, NY, USA: ACM, 2019, pp. 2203–2214.
[8] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng.*, vol. 2, May 2010, pp. 223–226.
[9] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, Association Computational Linguistics, 2020, pp. 4998–5007.
[10] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Assoc. Comput. Linguistics (EMNLP)*, T. Cohn, Y. He, and Y. Liu, Eds., Stroudsburg, PA, USA: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.
[11] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," in *Proc. 9th Int. Conf. Learn. Representations (ICLR)*, Virtual Event, Austria, May 2021, pp. 1–14. [Online]. Available: OpenReview.net
[12] H. Wu, H. Zhao, and M. Zhang, "SIT3: Code summarization with structure-induced transformer," 2020, *arXiv:2012.14710*.
[13] C. Lin, Z. Ouyang, J. Zhuang, J. Chen, H. Li, and R. Wu, "Improving code summarization with block-wise abstract syntax tree splitting," in *Proc. IEEE/ACM 29th Int. Conf. Program Comprehension (ICPC),* Madrid, Spain. Piscataway, NJ, USA: IEEE Press, May 2021, pp. 184–195.
[14] Z. Gong, C. Gao, Y. Wang, W. Gu, Y. Peng, and Z. Xu, "Source code summarization with structural relative position guided transformer," in *IEEE Inter. Conf. Softw. Analysis*., Honolulu, HI, USA: Evolution and Reengineering (SANER), 2022, pp. 13–24.
[15] S. Gao et al., "Code structure–guided transformer for source code summarization," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 1–32, Jan. 2023.
[16] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "TreeGen: A tree-based transformer architecture for code generation," in *Proc. AAAI Conf. Artif. Intell.*, vol. 34, no. 5, pp. 8984–8991, Apr. 2020.
[17] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," in *Proc. 37th Int. Conf. Mach. Learn.*, H. D. III and A. Singh, Eds., vol. 119, PMLR, Jul. 2020, pp. 245–256.
[18] N. D. Q. Bui, Y. Yu, and L. Jiang, "InferCode: Self-supervised learning of code representations by predicting Subtrees," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, Madrid, Spain. Piscataway, NJ, USA: IEEE Press, May 2021, pp. 1186–1197.
[19] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997.
[20] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," Jul. 2020, *arXiv:2003.13848*.
[21] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, Eds., vol. 27, Cambridge, MA, USA: Curran Associates, Inc., 2014, pp. 3104–3112.
[22] X.-P. Nguyen, S. Joty, S. C. H. Hoi, and R. Socher, "Tree-structured attention with hierarchical accumulation," Feb. 2020, *arXiv:2002.08046*.
[23] M. Ahmed, M. R. Samee, and R. E. Mercer, "You only need attention to traverse trees," in *Proc. 57th Annu. Meeting Assoc. Comput. Linguistics*, Florence, Italy. Stroudsburg, PA, USA: Association for Computational Linguistics, 2019, pp. 316–322.
[24] H. Shi, H. Zhou, J. Chen, and L. Li, "On tree-based neural sentence modeling," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, Eds., Brussels, Belgium. Association for Computational Linguistics, Oct./Nov. 2018, pp. 4631–4641.
[25] J. Harer, C. Reale, and P. Chin, "Tree-Transformer: A transformer-based method for correction of tree-structured data," Aug. 2019, *arXiv:1908.00449*.
[26] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Representations (ICLR)*, Y. Bengio and Y. LeCun, Eds., San Diego, CA, USA, 2015.
[27] T. Zhang, F. Wu, A. Katiyar, K. Q. Weinberger, and Y. Artzi, "Revisiting few-sample BERT fine-tuning," in *9th Inter. Conf. Learning Represen.*, Virtual Event, Austria, 2021, pp. 1–22.
[28] J. Song, Z. Zhang, Z. Tang, S. Feng, and Y. Gu, "Symer," GitHub. 2023. Accessed: May. 15, 2023. [Online]. Available: https://github.com/CloudLab-NEU/SyMer
[29] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics (ACL '02),* Philadelphia, Pennsylvania. Association for Computational Linguistics, 2001, Art. no. 311.
[30] B. Satanjeev, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proc. ACL Workshop Intrinsic Extrinsic Eval.,* pp. 228–231, 2005.
[31] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Proc. ACL Workshop: Text Summarization Braches Out,* 2004, p. 8.
[32] A. Murdoch, P. Merlin, and S. Wolf, "Gradle," GitHub. Accessed: May. 10, 2023. [Online]. Available: https://github.com/gradle/gradle

Jie Song received the Ph.D. degree in computer software and theory from Northeastern University, Shenyang, Liaoning, China, in 2008.

He is currently a Professor with the Software College, Northeastern University. His research interests include machine learning and big data management.

Zexin Zhang received the B.S. degree and the M.S. degrees in software engineering from Northeastern University, Shenyang, Liaoning, China, in 2020 and 2023, respectively. This article is the main work of his master's study.

His research interests include machine learning and programming language processing.

Zirui Tang is currently working toward the B.S. degree in software engineering with Northeastern University, Shenyang, Liaoning, China, in 2011.

His research interests include machine learning and programming language processing.

Shi Feng received the Ph.D. degree in computer software and theory from Northeastern University, Shenyang, Liaoning, China, in 2011.

He is currently an Associate Professor with the College of Computer Science and Engineering, Northeastern University. His research interests include sentiment analysis and dialogue systems.

Yu Gu received the Ph.D. degree in computer software and theory from Northeastern University, Shenyang, Liaoning, China, in 2010.

He is a Professor in computer science with Northeastern University. His research interests include graph data management, spatial data management, and big data analysis.