A Multiple Representation Transformer with Optimized Abstract Syntax Tree for Efficient Code Clone Detection

Tianchen Yu[1*], Li Yuan[1*], Liannan Lin[1†], and Hongkui He [1]
[1] School of Software Engineering, South China University of Technology
Email:{seyutianchen,seyuanli,lnlin,sekuih}@mail.scut.edu.cn,

***Abstract*—Over the past decade, the application of deep learning in code clone detection has produced remarkable results. However, the current approaches have two limitations: (a) code representation approaches with low information utilization, such as vanilla Abstract Syntax Tree (AST), leading to information redundancy which results in performance degradation; (b) low efficiency of clone detection on evaluation, resulting in excessive time costs during practical use. In this paper, we propose a Multiple Representation Transformer with an Optimized Abstract Syntax Tree (MRT-OAST) to introduce an efficient code representation method while achieving competitive performance. Specifically, MRT-OAST strategically prunes and enhances the AST, utilizing both pre-order and post-order traversals to represent two different representations. To speed up the evaluation process, MRT-OAST utilizes a pure Siamese Network and employs cosine similarity to compare the similarity between codes. Our approach effectively reduces AST sequences to 40% and 39% of their original length in Java and C/C++ while preserving structural information. In code clone detection tasks, our model surpasses state-of-the-art approaches on OJClone and Google Code Jam. During the evaluation of BigCloneBench, our model has a 5x speed improvement compared to the state-of-the-art lightweight model and a 563x speed improvement compared to the BERT-based model, with only a 0.3% and 0.9% decrease in F_1-score.**

***Index Terms*—abstract syntax tree, code clone detection, Transformer**

I. Introduction

Code clone pertains to two or more identical or similar source code fragments within a codebase. Typically, these clones manifest at either the function-level [1] or the file-level [2]. It is believed that code cloning plays a pivotal role in several software engineering tasks, including code classification [3], plagiarism detection [4], [5], bug identification [6], [7] and reuse [8]. Currently, the main challenge in code clone detection lies in achieving a more concise and efficient representation of code [9] while preserving its structural information.

Traditional approaches often treated code as plain text and utilized methods based on natural language processing or code token-based techniques for matching, such as information retrieval [10], [11]. However, these methods are unable to effectively detect semantically code clones with different formatting while having identical functionality. Recently, most researchers have proposed deep learning techniques with different representations for code clone detection, such as RNN-based approaches with Abstract Syntax Tree (AST) [12]–[14], RNN-based approaches with Program Dependency Graph(PDG) [15], Tree-based CNN approach with AST [16], DNN-based approaches with AST [17]–[21].

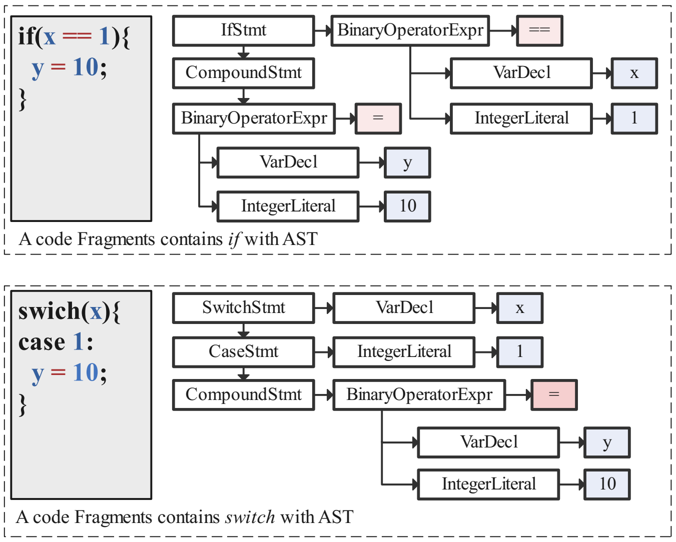
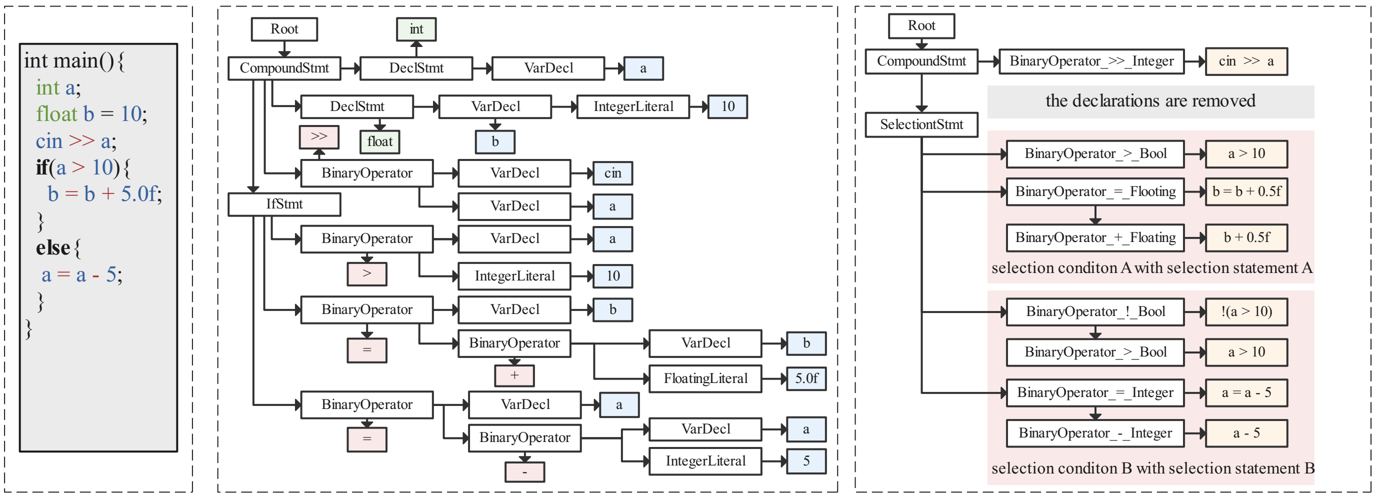


Fig. 1: An example of code using *if* and *switch* constructs with identical functionality, resulting in completely different ASTs.

The primary constraint of the AST lies in its partial information loss and redundancy. Figure 1 illustrates two code snippets with the shared objective of assigning a value of 10 to variable y when x equals 1, yet they have completely different ASTs, resulting in logical redundancy. AST also fails to differentiate between symbols and types, resulting in information loss. An example of AST optimization is shown in Figure 2, where AST presents the following three significant issues: (1) Many nodes in the AST introduce excessive granularity when representing code structures, leading to information redundancy. For example, *VarDecl* nodes and *Literal* nodes in Figure 2(b) are redundant. (2) Expressions like binary operator expression and unary operator expression may lack symbol annotations, which results in the inability to differentiate between expression

* Equal contribution.
† Corresponding author.



```
int main(){
  int a;
  float b = 10;
  cin >> a;
  if(a > 10){
    b = b + 5.0f;
  }
  else{
   a = a - 5;
  }
}
```

(a)Code (b)AST (c)OAST

Fig. 2: An example of AST optimization, where (a) is a C++ code fragment, (b) and (c) denote its AST and OAST respectively. Only the nodes highlighted in white are stored in the OAST, while the nodes highlighted in orange demonstrate the link between the OAST and the code.

nodes with different operators. All the *BinaryOperator* nodes in Figure 2(b) do not represent the type of operation, causing the ASTs for floating-point and integer arithmetic operations to appear very similar. (3) Code exhibits diverse variations in loops and branches. Different constructs such as *for*, *while*, and *do-while* can be used to express loops, while *if* and *switch* can be used for branching.

To address the limitations resulting from redundant AST. some recent researchers [14] employ a subset of AST node sequences, while Zhang et al. [22] take the AST into different subtrees for learning. However, the effectiveness of this approach varies depending on the granularity of the partition, and an excessive number of subtrees can result in significantly slower predictions. In contrast, graph-based methods often introduce additional edges to the AST to incorporate more information [9], [23] or use Graph Convolutional Network (GCN) [9] to simplify the AST. Despite these efforts, the limitation of redundant AST has not been effectively addressed, limiting practical applicability to real-world projects. Furthermore, many methods [12]–[14] utilize a single AST traversal approach to obtain sequences, which may lead to model overfitting and inadequate learning of structural information.

Another critical consideration in code clone detection is the repetitive extraction of features from the same code segments [9], [22], [24]–[26], leading to lower detection efficiency, especially with Bert-based models that prioritize accuracy over speed [24]–[26]. Such approaches are impractical in real-world scenarios. For instance, when dealing with OJClone [16] containing 7,500 code fragments with pairwise comparisons of similarities. The comparisons within the model would exceed 28 million times, making the process entirely difficult.

To address these mentioned challenges, in this paper, we propose the Multiple Representation Transformer with Optimized Abstract Syntax Tree (MRT-OAST) for code clone detection. Firstly, to address limitations related to large AST, we propose an AST optimization technique, which aims to remove redundant nodes while introducing symbols and abstract types to expression nodes to enhance the AST. Additionally, we mitigate logical redundancy by unifying different loop and branch structures. To effectively preserve structural information when converting the AST into sequences and to capture hierarchical relationships and dependencies between nodes, we obtain the pre-order and post-order sequences of the OAST. These sequences are then input into a Multiple Representation Transformer (MRT) model for learning bidirectional representations. Finally, we employ a pure Siamese Network [27] architecture to make MRT-OAST better match the requirements from real-world scenarios. This approach enables each code segment to generate a feature only once through the model during evaluation, facilitating the swift determination of feature similarities using cosine distance calculations. We conducted experiments on OJClone [16], GCJ [28], and BCB [29] to validate the performance of our model. Our method prunes and enhances the AST, preserving structural information while avoiding redundancy, which improving the model's ability to extract representations. The contributions of this paper are as follows:

- We propose a straightforward and effective AST optimization approach that reduces the size of the AST while preserving structural information. Experimental results indicate that AST optimization reduces the size of ASTs to 40% and 39% of their original size in Java and C/C++ respectively, effectively addressing redundancy within the AST.
- We present a multiple representation Transformer model with pure Siamese Network architecture that efficiently

captures structure representations from diverse perspectives.

- Our proposed method surpasses the state-of-the-art models on two benchmarks OJClone [16] and Google Code Jam (GCJ) [28] with 0.9% and 0.2% increase in F_1-score, while maintaining high-speed evaluation
- our model demonstrated a 5x speed improvement compared to the state-of-the-art lightweight model and a 563x speed improvement compared to the BERT-based model, with only a 0.3% and 0.9% decrease in F_1-score.

II. Background

A. Abstract Syntax Tree

The AST serves as an abstract representation of the syntactic structure of the source code [30]. It depicts the grammar structure of a programming language as a tree, where each node represents a structure in the source code. Unlike the programming language itself, the AST does not incorporate user-defined names or include non-executable content, such as comments. Therefore, the AST proves to be an effective choice for representing programming languages and has found extensive applications in various software engineering tools. Some studies directly utilize AST in token-based methods for source code search [31], program repair [32], and source code differencing [33]. However, these methods may capture only limited syntactical information of the source code.

B. Transformer Siamese Network

The Transformer Siamese Network is a widely used model architecture for contrastive learning, which has been successfully applied in various domains, such as video object segmentation [34], semantic change detection [35], and image change detection [36]. Notably, researchers have recently extended its application to code research [9]. As shown in Figure 3, the validation process of the Transformer Siamese Network does not require redundant computation of the feature for the same code file. When comparing a single code file with multiple others, its feature vector only needs to be computed once, significantly reducing the time required for detecting code clones across numerous files.

The Transformer [37] is comprised of alternating layers of Multi-Head Self-Attention (MSA) and Feed-Forward Neural Networks (FNN) blocks. Before each block, Layer Normalization (LN) and residual connections [38] are applied. In the Self-Attention mechanism, three matrices, Q (Query), K (Key), and V (Value), are obtained through linear transformations of the same input. It begins by calculating the dot product between Q and K, which is then divided by a scale factor of $\sqrt{d_k}$, where d_k represents the dimensions of the query and key vectors. The resulting matrix is then normalized into a probability distribution using the softmax operation and multiplied by the matrix V, yielding a weighted sum representation. This operation can be expressed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{1}$$

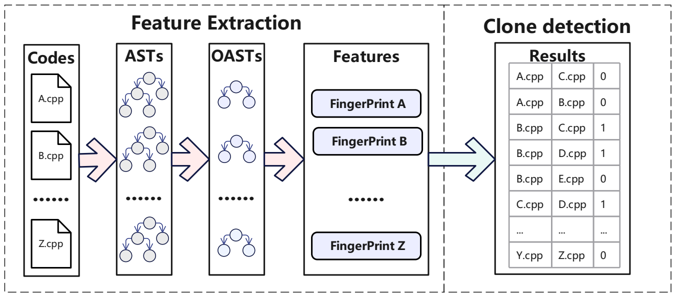


Fig. 3: The evaluation process of Transformer Siamese Network, the feature of a code snippet is computed only once during the feature extraction stage. During the clone detection stage, the features of different codes are compared with each other as required for the comparison of clone pairs.

Multi-Head Attention is a mechanism that enables parallel execution of attention multiple times within a single attention module. By maintaining the matrices Q, K, and V unchanged, attention computations are performed multiple times, and the output heads are concatenated to obtain the final output h. This process can be described as follows:

$$\begin{aligned} \text{MSA}(Q, K, V) &= \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \\ \text{head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{2}$$

where W^O serve as weight matrices to linearly combine the outputs of multiple attention heads.

C. Multiple Representation

The concept of Multiple Representation is extensively employed in deep learning and plays a pivotal role in various tasks. Several studies utilize different visual descriptors to describe images [39], [40]. By incorporating multiple representations, models can capture diverse aspects of data, leading to a more comprehensive understanding of complex patterns. This technique is also applied in 3D Shape Recognition [41], enhancing the models' ability to handle diverse and challenging scenarios. Furthermore, multiple representations can be a form of regularization [42]. When one representation tends to overfit, another can effectively contribute, preventing excessive fitting to the training data. In code clone detection, adopting multiple representations can offer diverse perspectives and facilitate the discovery of similarities among functionally similar code segments. Identifiers, different traversal methods of the AST, and the Control Flow Graph (CFG) can be regarded as multiple representations of the code. [43].

III. Our Approach

In this section, we present an overview of MRT-OAST, as is shown in Figure 4. The vanilla AST is first obtained through a compiler, widely used in previous research [14], [22]. Subsequently, the vanilla AST is pruned and enhanced to obtain the OAST through AST Optimizer. Additionally, we derive the pre-order and post-order traversals of the OAST and extract output vectors using the MRT. Finally, the code's

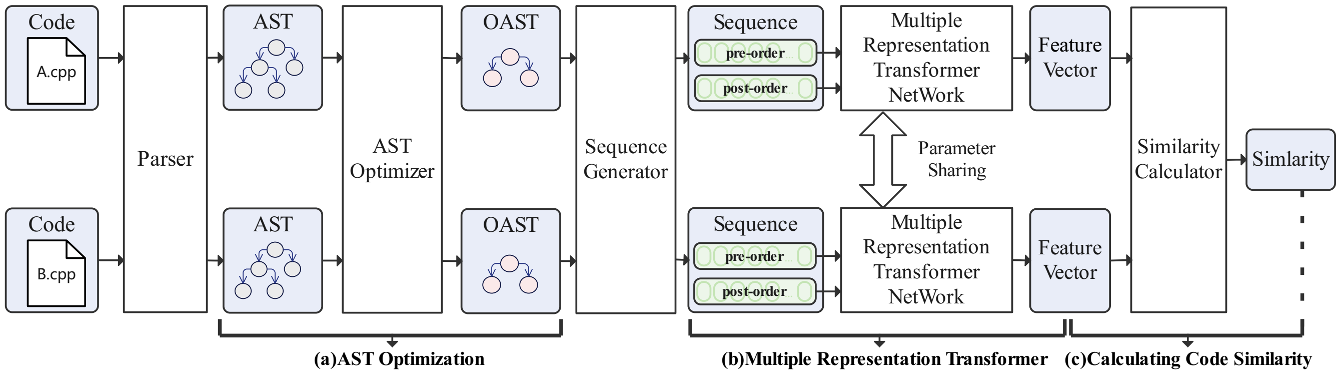


Fig. 4: The overview of our approach. The entire method consists of three main processes: (a) AST Optimization, (b) Multiple Representation Transformer, and (c) Calculating Code Similarity.

similarity is measured by computing the cosine similarity of the code output vectors.

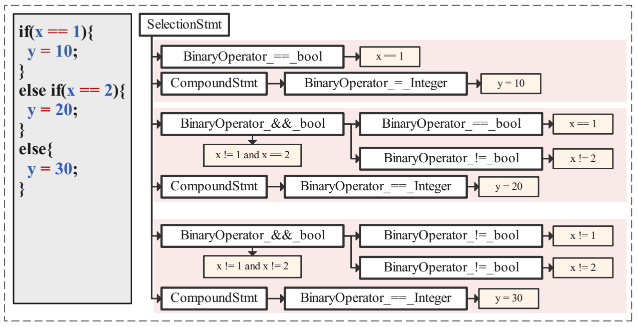
A. *AST Optimization*

To address the issues of redundant large AST, our AST optimization process comprises three processes:

1) Clean: We remove nodes generated due to compilation errors, automatically added by the compiler, as well as nodes related to declarations and constants. The removal of these redundant nodes significantly alleviates the issue of redundant large AST.
2) Enhanced Operators: we refine expressions by adding symbols and types and remove the reference nodes under the expression. This resolves the problem of missing symbols and types for operators. A *BinaryOperator* node computing the addition of two floating-point numbers would be optimized as a *BinaryOperator_+_float* node. For certain function call statements whose return value is not feasible, we will enhance these statements by incorporating the function name.
3) Unify Logic: we unify *if* and *switch* into *selection* statement, as well as unifying *while*, *do-while*, and *for* into *loop* statement. This addresses the issue of diversified loops and branch structures.

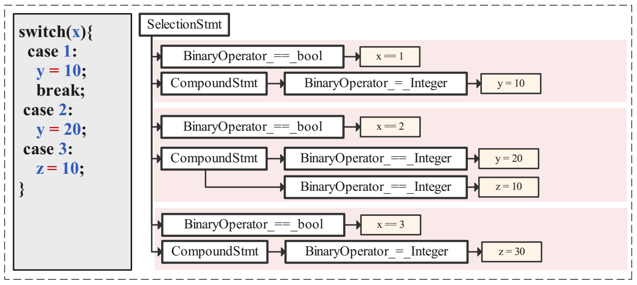
In the process of unifying logic, we have introduced two new node types: the *selection* statement and the *loop* statement. The *selection* statement node comprises children consisting of N sets of AST subtrees representing the branch conditions and the corresponding code for branch execution. On the other hand, the *loop* statement node includes children encompassing the loop condition and the AST subtree representing the code for loop execution. Three examples of AST optimizations are shown in Figure 5: Figure 5(a) displaying a code with an *if* statement and its OAST with a *selection* statement, Figure 5(b) displaying a code with a *switch* statement and its OAST with a *selection* statement, and Figure 5(c) displaying a code with a *for* statement and its OAST with a *loop* statement. Moreover, we have devised processing approaches for five node types:

- **If statement**: All conditions and code blocks corresponding to *if* and *else-if* are directly extracted as $[E_1, ..., E_n]$ and $[B_1, ..., B_n]$, respectively. If an *else* statement exists, its code block is obtained as B_{else}, and the corresponding statement $E_{else} = \neg(E_1 \wedge E_2 \wedge ... \wedge E_n)$. A new *selection* statement is created with the parent node as the parent of the original *if* statement, and its children are represented as $[E_1, B_1, ..., E_n, B_n, E_{else}, B_{else}]$.



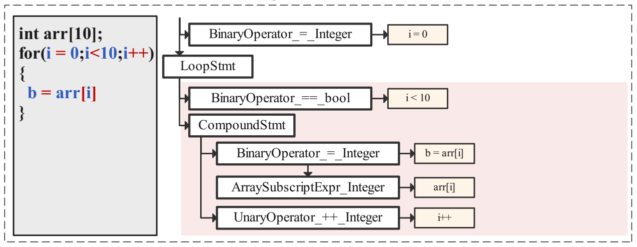
```
if(x == 1){
  y = 10;
}
else if(x == 2){
  y = 20;
}
else{
  y = 30;
}
```

(a)A code Fragments contains *if* with OAST



```
switch(x){
 case 1:
   y = 10;
   break;
 case 2:
   y = 20;
 case 3:
   z = 10;
}
```

(b)A code Fragments contains *switch* with OAST



```
int arr[10];
for(i = 0;i<10;i++)
{
  b = arr[i]
}
```

(c)A code Fragments contains *for* with OAST

Fig. 5: Three examples of codes with their OASTs. Only the nodes highlighted in white are stored in the OAST, while the nodes highlighted in orange demonstrate the link between the OAST and the code.

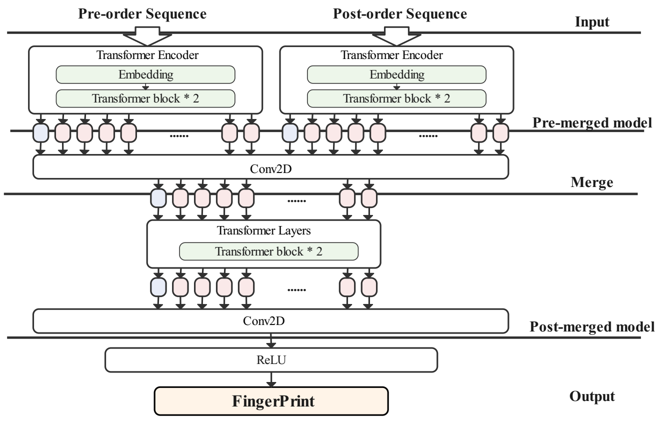


Fig. 6: The structure of our MRT model. The pre-merged model utilizes two Transformer Encoders, each containing an Embedding layer and two Transformer blocks and the post-merged model consists a simple two-layer Transformer.

- **Switch statement**: The creation of a new *selection* statement through a *switch* statement is achieved using Algorithm 1. Each *case* statement within a *switch* statement is associated with the generation of a condition and its corresponding *compound* statement. For each condition, its corresponding *compound* statement depends on whether the *case* statement contains a *break* statement. If the *case* statement includes a *break* statement, all the statements within that *case* statement are considered as child nodes of the *compound* statement. If the *case* statement lacks a *break* statement, the statements under that *case* statement, as well as the subsequent *case* statement are viewed as child nodes of the *compound* statement until a *break* statement is encountered. All conditions and their corresponding *compound* statements will be considered as the child nodes of the new *selection* statement.
- **For statement**: The initialization, condition, iteration, and code block of the *for* statement are extracted as E_{init}, E_{cond}, E_{iter}, and B, respectively. If E_{init} exists, it is added to the parent node. If E_{iter} exists, it is added as a child of B. A new *loop* statement is created with the parent node as the parent of the original *for* statement. Its children are represented as $[E_{cond}, B]$.
- **While statement**: The condition and code block of the *while* statement are extracted as E_{cond} and B, respectively. A new *loop* statement is created with the parent node as the parent of the original *while* statement, and its children are represented as $[E_{cond}, B]$.
- **Do-while statement**: Since the *do-while* statement only differs from the *while* statement by executing the loop body one more time, the processing approach is the same as the *while* statement.

B. Multiple Representation Transformer

To further process the obtained AST from the previous step, we utilize a multiple representation Transformer-based network for feature extraction. The structure of our model is illustrated in Figure 6. In essence, our model comprises five modules.

Algorithm 1 Convert the *switch* to *selection* statement

```
Require:
  Input: a node of switch statement $N_{switch}$;
  Output: a node of selection statement $N_{select}$
Ensure: There are no syntax errors in $N_{switch}$.

  $N_{select} \leftarrow$ new a node of selection statement
  $N_{expr} \leftarrow$ the expression child node of $N_{switch}$
  $S_{cond} \leftarrow \varnothing$ $S_{cache} \leftarrow \varnothing$ $S_{stmt} \leftarrow \varnothing$
  $S_{child} \leftarrow$ children of $N_{switch}$ except expression node
  for $N_{child} \in S_{child}$ do
      if $N_{child}$ is a case statement node then
          $N_{eq} \leftarrow$ new a node of expression with operator '=='
          Set $N_{child}$ and $N_{expr}$ as child nodes of $N_{eq}$
          $N_{comp} \leftarrow$ new a node of compound statement
          $S_{cache} \leftarrow S_{cache} \cup \{N_{eq}\}$
          $S_{cond} \leftarrow S_{cond} \cup \{N_{eq}\}$
          $S_{stmt} \leftarrow S_{stmt} \cup \{N_{comp}\}$
      else if $N_{child}$ is a break statement node then
          for $i$ in $0$ to size($S_{cache}$) $- 1$ do
              Set $S_{cache}[i]$ and $S_{stmt}[i]$ as
              child nodes of $N_{select}$
          end for
          $S_{stmt} \leftarrow \varnothing$ $S_{cache} \leftarrow \varnothing$
      else if $N_{child}$ is a default statement node then
          $N_{expr}$ is a new node equivalent to taking the logical
  conjunction of all expression nodes in $S_{cond}$ and negating
  it
          $N_{comp} \leftarrow$ new a node of compound statement
          $S_{cache} \leftarrow S_{cache} \cup \{N_{expr}\}$
          $S_{stmt} \leftarrow S_{stmt} \cup \{N_{comp}\}$
      else
          for $N_{comp} \in S_{stmt}$ do
              Set $N_{child}$ as a child node of $N_{comp}$
          end for
      end if
  end for
  for $i$ in $0$ to size($S_{cache}$) $- 1$ do
      Set $S_{cache}[i]$ and $S_{stmt}[i]$ as child nodes of $N_{select}$
  end for
```

Input: For the optimized AST obtained from the previous step, we perform a pre-order traversal and a post-order traversal starting from the root node of the tree. This allows us to obtain the pre-order sequence S_{pre} and the post-order sequence S_{post} respectively.

Pre-merged model: We utilize a complete Transformer Encoder in pre-merged model.S_{pre} and S_{post} are transformed into matrices through their respective word embedding layers. To incorporate positional information into the sequences, we employ the widely adopted cosine positional encoding technique, described as follows:

$$\text{PositionalEncoding}(pos, 2i+1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \quad (3)$$

where i represents the dimension index of the encoding vector, d_{model} refers to the dimension of the word embedding layer's word vectors, and pos indicates the position index.

In MRT, the FFN consists of two linear layers using the ReLU activation function and a dropout layer to enhance robustness. The format can be represented as follows:

$$\text{FFN}(x) = (\max(0, \text{Dropout}(xW_1 + b_1)))W_2 + b_2 \quad (4)$$

where W_1 and W_2 represent two weight matrices, b_1 and b_2 denote biases. x corresponds to the output of MSA. After passing through the pre-merged model, the sequences S_{pre} and S_{post} are transformed into two corresponding matrices M_{pre} and M_{post}.

Merge: To effectively combine two sequences, M_{pre} and M_{post}, while minimizing parameter usage, we utilized a convolutional layer with a kernel size of $[1, 1]$. This operation transforms the merging of features into an interpolation operation between the two matrices. Treating the two sequences as separate channels, the convolutional layer generated a single-channel output, resulting in the matrix M_{concat}. The applied convolution formula is as follows:

$$M_{concat}[i, j] = M_{pre}[i, j] \times K_1 + M_{post}[i, j] \times K_2 \quad (5)$$

where K_1 represents a trainable parameter that connects the first input channel to the output channel, while K_2 represents a trainable parameter that connects the second input channel to the output channel. $M_{pre}[i, j]$ and $M_{post}[i, j]$ denotes the input element at position (i, j) of M_{pre} and M_{post}.

Post-merged model: The single-channel output M_{concat} obtained from the previous step represents the merged sequence. It is further trained through multiple layers of the Transformer to obtain M_{result}. To compress the extracted feature matrix, we employ a convolutional layer for output vector extraction, using the following formula:

$$M_{result}[c, i] = \sum_{j=1}^{J} M_{pre}[i, j] \times K_{cj} \quad (6)$$

where K_{cj} is a trainable parameter connecting the input channel to the c-th output channel. c ranges from 1 to the total number of output channels which is denoted as c_{max}. The symbol J represents the dimension of word vector embeddings. $M[i, j]$ denotes the input element at position (i, j). By controlling the number of output channels, we can obtain output vectors of different sizes. Finally, the matrix M_{result} will be flattened to obtain an output vector V_{result} with a dimension of c_{max} * J. For two pieces of code that require similarity comparison, they will be assigned V_A and V_B respectively.

Output: To better assess the similarity of the code and mitigate overfitting, we set all elements in V_A and V_B that are less than 0 to 0. This serves as the final output of the model.

C. Calculating Code Similarity

Cosine similarity is a commonly used similarity measure that returns values between -1 and 1. It is used to compare the output vectors, V_A and V_B, obtained in the previous step. The final predict output y between V_A and V_B is as follows:

$$y = f(V_A, V_B) = \frac{V_A \cdot V_B}{\|V_A\| \cdot \|V_B\|} \quad (7)$$

where f denotes the cosine similarity

Since V_{result} is a vector with all values greater than 0, the computation of cosine similarity will yield values between 0 and 1. We consider 0 as dissimilar and 1 as similar. For training, we ultimately employ cross-entropy as the loss function to measure the ground truth distribution $\hat{y}$ and predicted distribution y. The formula is as follows:

$$\text{BCELoss}(y, \hat{y}) = -\left(y \log(\hat{y}) + (1-y)\log(1-\hat{y})\right) \quad (8)$$

We set a threshold value denoted as α to determine whether the final output is either similar or dissimilar. The formula is as follows:

$$\text{sim} = \begin{cases} 1, & \text{if } f(V_A, V_B) > \alpha, \\ 0, & \text{otherwise.} \end{cases} \quad (9)$$

IV. Experimental settings

Before presenting the experimental results, we first provide an overview of the dataset and basic settings of our model.

A. Dataset Details

We evaluate our approach on three datasets: GCJ [28] , OJClone [16], and BCB [29]. Following the previous works' setting [15], [23], [44]. GCJ is an annual international programming competition conducted by Google. GCJ contains 1669 Java files submitted under 12 different competition problems. In GCJ, participants were required to write code fragments to solve different competition problems. The code fragments submitted under the same competition problem have the same function but were implemented differently. Like many other approaches to code clone detection [15], [22], [23], [44], [45], it is assumed that the code fragment pairs under the same competition problem are regarded as positive clone pairs, while the code fragment pairs under different competition problems are considered as negative clone pairs.

OJClone [16] contains 104 programming problems and various source codes students submit for each problem. When two distinct source codes resolve the same programming problem, they are regarded as a code clone pair attributed to their identical functionality. Our dataset configuration follows the commonly employed methods [9], [18], [23]. Specifically, the OJClone dataset contains 15 programming problems and each of them have 500 problems. We selected 50,000 samples from OJClone and divided them into an 8:1:1 ratio for training, validation, and testing purposes.

BCB [29] contains over 6,000,000 positive clone pairs and 260,000 negative clone pairs. The detailed distribution of true clone pairs is presented in Table I, where types T1,

TABLE I: The Distrubution of Clone Types in BCB

Clone type	T1	T2	ST3	MT3	WT3/4
Percentage(%)	0.455	0.058	0.243	1.014	98.23

TABLE II: Dataset Statistics

Dataset		GCJ	OJClone	BCB
Basic	Programing language	Java	C,C++	Java
	Clone level	file	file	function
	Parser	javalang	clang	javalang
	Code fragments	1,669	7,500	9,133
	Average code lines	56.62	35.92	32.80
AST	Node types	55	47	56
	Average tree nodes	223.4	158.3	129.28
	Max tree nodes	1,574	1,677	4,998
	Average tree depth	15.64	13.91	9.57
	Max tree depth	34	64	61
	Average *switch* statements	0.013	0.028	0.119
	Average *if* statements	3.114	2.889	2.723
	Average *for* statements	4.064	3.698	0.422
	Average *while* statements	0.437	0.206	0.441
	Average *do-while* statements	0.006	0.029	0.014
OAST	Node types	416	113	4,398
	Average tree nodes	78.54	61	53.74
	Max tree nodes	435	2,984	2,317
	Average tree depth	13.17	11.05	7.72
	Max tree depth	32	108	59
	Average *selection* statements	3.09	2.96	2.71
	Average *loop* statements	4.49	3.93	0.87

T2, strongly type-3 (ST3), and moderately type-3 (MT3) are classified as syntactic clones, while weakly type-3/type-4 (WT3/T4) are considered semantic clones [23]. ST3, MT3, and WT3/T4 are defined with similarity in [0.7, 1.0), [0.5, 0.7), and [0.0, 0.5). The majority of clone pairs fall under the category of WT3/T4, BCB proves suitable for evaluating semantic clone detection. BCB dataset for evaluation in our experiments contains 359,223 negative clone pairs and 57,105 positive clone pairs, which is widely adopted by multiple methods [9], [13], [23], [44]. BCB serves as the performance evaluation dataset in our experimental.

Table II shows the basic and preprocessing information of the three datasets. The disparity in the number of lines of code between the GCJ and BCB is apparent. This discrepancy can be attributed to the fact that the code clones in GCJ operate at the file level, allowing a single GCJ code file to potentially encompass multiple functions or even several classes. A similar situation is observed between OJClone and BCB. However, due to the concise nature of the C/C++ language and the relatively simpler problems it addresses, OJClone demonstrates only a marginally higher average number of code lines compared to BCB. Furthermore, the use of structural statements, particularly *loop* statements, is less frequent in BCB compared to GCJ and OJClone. Notably, BCB exhibits a substantially higher number of OAST nodes compared to OJClone and GCJ. This is due to the fact that numerous codes in the BCB dataset are employed to address practical requirements [29], resulting in a substantial volume of diverse library function calls. Consequently, during the generation OASTs of BCB and GCJ, type names and function names that appear 8 times or more are retained, while those with infrequent occurrences and unknown type names and function names are recorded as *Builtin*.

B. Implementation Details

We developed our training data on OJClone following the conventions of other methods [9], [18], [23]. For BCB and GCJ, we directly utilized the well-balanced datasets from FA-AST [23], which have also been used by other methods [9]. We truncated OAST sequences exceeding a length of 256. Less than 1% of the code in the three datasets corresponded to OAST lengths exceeding 256. Notably, we utilize balanced positive and negative samples during training on the training sets, ensuring a near 1:1 ratio between positive and negative clone pairs.

Our method takes code as input and then utilizes AST optimization to obtain OAST. OAST is saved in a sequential representation using a structure-based traversal method [46]. Subsequently, the sequential representation is restored back to an AST before being fed into the enhanced neural network model. The model's word embedding dimension J is set to 128, the number of heads in the multi-head attention mechanism of all Transformer modules is set to 8, and the intermediate layer dimension of the FFN layer in the Transformer is set to 512. Additionally, c_{max} is set to 4, and the output vector dimension is 512. During the training process, we utilized the Adam optimizer. We implemented a warm-up strategy during the training process, gradually elevating the learning rate from 0 to the target learning rate within the initial 20% of steps. Following this, the learning rate steadily decreased in a linear fashion until the completion of training, ultimately reaching 10% of the target learning rate. The learning rates are set to 0.001 for OJClone, 0.0001 for BCB and GCJ. We train OJClone, BCB and GCJ for 30, 10 and 10 epochs. The dropout is set to 0.3 and batch size is set to 64. We implemented our compilation and AST optimization steps on C/C++ using libtooling from clang [47]. The compilation and AST optimization steps of Java are completed using javalang in Python. Our model was implemented using the PyTorch framework [48] and all experiments are conducted on a computer with a Platinum 8255C CPU and an RTX 3090 24G GPU. We have set the evaluation thresholds at 0.4, 0.9, and 0.8 for OJClone, BCB, and GCJ respectively. Our choice of commonly used evaluation metrics includes Precision (P), Recall (R), and F_1-score (F_1).

V. Experimental results

A. Effects of Our Approach

RQ1: How does our method perform in the field of code clone detection? To evaluate the performance of our method, we compared it with 14 baselines. For a fair comparison, we evaluated our method against other methods using the same evaluation set on the BCB and GCJ datasets. On OJClone, we compared our results with those obtained a similar data processing approach. All three datasets almost exclusively contain type-3 and type-4 clones, so the experimental results primarily focus on the detection of semantic clones. One baseline may appear in one or more dataset evaluation results.

The baselines are as follows:

TABLE III: Results on OJClone, GCJ and BCB, all experiments are setting with the same CPU and GPU.

Group	Methods	OJClone					GCJ					BCB				
		P	R	F_1	Pairs	Times	P	R	F_1	Pairs	Times	P	R	F_1	Pairs	Times
Token-Based	RtvNN	56	65.6	48.9	-	-	20.0	90.0	33.0	-	-	10.0	90.0	18.0	-	-
	DLC	70.0	83.0	75.9	-	-	-	-	-	-	-	-	-	-	-	-
	Toma	-	-	-	-	-	-	-	-	-	-	92.0	88.0	90.0	548,838	22s†
Tree-Based	Deckard	-	-	-	-	-	45.0	44.0	44.0	-	-	93.0	6.0	12.0	-	-
	DeepSim	-	-	-	-	-	82.0	71.0	76.0	-	-	-	-	-	-	-
	SSFL	97.0	95.0	96.0	-	-	-	-	-	-	-	-	-	-	-	-
	ASTNN	98.9	92.7	95.7	5,000	54s†	98.0	93.0	95.0	13,695	145s†	93.4	92.3	92.9	359,223	3,747s
	AT-BiLSTM	96.0	95.3	96.8	5,000	24.2s†	-	-	-	-	-	95.9	91.1	93.4	416,328	243s†
Graph-Based	FA-AST+GGNN	-	-	-	-	-	96.0	99.0	97.0	13,695	178s†	85.0	90.0	88.0	416,328	1,037s
	FA-AST+GMN	-	-	-	-	-	99.0	97.0	98.0	13,695	252s†	95.8	93.6	94.7	416,328	1,469s
	CTL	99.3	98.3	98.8	5,000	21.6s†	-	-	-	-	-	96.1	95.2	95.6	416,328	1,793s†
	PNIAT+PDG	-	-	-	-	-	99.3	99.5	99.4	13,695	9.9s	93.6	**97.0**	95.2	413,595	58s
	PNIAT+CFG	-	-	-	-	-	99.5	99.7	99.6	13,695	9.9s	96.3	96.8	96.5	413,595	58s
Pretrained	GraphCodeBERT	-	-	-	-	-	-	-	-	-	-	-	-	**97.1**	413,595	6,195s
Ours	MRT+DAST*	-	-	-	-	-	98.3	98.2	98.3	13,695	2.9s	89.6	89.1	89.3	416,328	11.0s
	MRT+SAST*	-	-	-	-	-	99.2	99.3	99.3	13,695	**2.7s**	90.5	90.2	90.4	416,328	11.0s
	MRT+OAST	**99.8**	**99.5**	**99.7**	5,000	**7.5s**	**99.8**	**99.9**	**99.9**	13,695	**2.7s**	**96.7**	95.7	96.2	416,328	**10.9s**

† indicates that the time cost is based on our reproduced results. The other time costs are reported from PNIAT [44].
DAST is Decomposed Abstract Syntax Tree [14]. SAST is Sub Abstract Syntax Tree [49]

- RtvNN [12]: A RNN-based method that learns code lexical-level embeddings and full binary ASTs at the grammar level to match code clones.
- DLC [50]: A method based on deep learning that employs a recursive auto-encoder for code feature extraction.
- Toma [51]: A method that extracts token type sequences and employs six similarity calculation methods to generate feature vectors. These vectors are then input into a trained machine learning model for classification.
- Deckard [52]: A syntax tree-based code clone approach that employs Euclidean distance for calculating the similarity between code fragments.
- Deepsim [15]: A method that translates code control flow and data flow into semantic matrices, utilizing a deep learning model for detecting code clones.
- SSFL [18]: A model that develops a unique joint code representation to capture hidden semantic and syntactic features of source codes.
- ASTNN [22]: A model that converts the AST of source code into the sequence of its subtrees and uses bidirectional RNN to learn code representation for code cloning detection
- AT-BiLSTM [14]: A model that utilizes an attention mechanism with BiLSTM to better understand the structural information of the AST.
- FA-AST+GMN and FA-AST+GGNN [23]: Models that augment original ASTs with explicit control and data flow edges, using GMN and GGNN to measure the similarity of code pairs.
- Code Token Learner (CTL) [9]: A state-of-the-art model on OJClone, using GCN to learn AST features and employing Transformer and Cross-attention for code clone detection.
- PNIAT+PDG and PNIAT+CFG [44]: A state-of-the-art lightweight model on BCB and GCJ, utilizing Parallel Node Internal multi-head Attention (PNIAT) based on attention mechanisms to capture key tokens in each statement in parallel.
- GraphCodeBERT [26]: A state-of-the-art BERT-based model on BCB. GraphCodeBERT is a multi-programming-lingual model pretrained on 6 programming languages including Java.

The experimental results of three datasets are presented in Table III, with the best results highlighted in bold. Specifically, our precision, recall and F_1 score are **0.5%, 1.2%, 0.9%** higher than the current best model [9] on OJClone. The precision, recall and F_1 score are **0.3%, 0.2%, 0.3%** higher than the current best model [44] on GCJ. The results indicate that our model outperforms both PNIAT [44] and CTL [9] through the optimization and preprocessing of the AST, in addition to the integration of multiple representations.

Furthermore, preserving the structural information of the AST enhances the results. Lastly, the Transformer [9] model demonstrates superior ability in learning long-term dependencies compared to RNN [12] or LSTM [14], [22], particularly in capturing code features. Our results are slightly lower than PNIAT [44] and GraphCodeBERT [26] on BCB, which is also shown in Table III. The shorter sequences in BCB may pose a challenge for Transformers in capturing the relationships between tokens. Simultaneously, an excessively large vocabulary can be detrimental to the training of lightweight models. However, our model achieves similar results while boasting a lower evaluation time and a simpler method for comparing similarities. We will elaborate on our efficiency advantages in Section V-B.

We also test the effectiveness of combining our MRT model with other AST processing methods in Table III. Specifically, we test two tree-based AST processing methods on Java datasets GCJ and BCB. We followed their descriptions [14], [49] to reproduce the construction logic of DAST and SAST.

- Decomposed Abstract Syntax Tree(DAST) [14]: DAST is a method that divides each node into two parts: block

TABLE IV: Ablation experimental results of our method on OJClone, GCJ and BCB

Dataset	Pairs	Code fragments	Ablation experiments	P	R	F_1	Feature extraction stage	Clone detection stage	Total time consumption
OJClone	5,000	7,500	w/o OAST	94.9	96.2	95.6	7.9s	0.1s	8.0s
			w/o SN	99.8	99.5	99.7	-	-	10.1s
			-	99.8	99.5	99.7	7.4s	0.1s	7.5s
GCJ	13,695	1,699	w/o OAST	99.7	97.7	98.7	2.7s	0.1s	2.8s
			w/o SN	99.8	99.9	99.9	-	-	8.2s
			-	99.8	99.9	99.9	2.6s	0.1s	2.7s
BCB	416,328	9,133	w/o OAST	83.1	82.3	82.7	9.1s	1.9s	11.0s
			w/o SN	96.7	95.7	96.2	-	-	256.1s
			-	96.7	95.7	96.2	9.0s	1.9s	10.9s

SN is Siamese Network

and body. The block records the node's attributes, while the body records the AST subtree under the node.

- Sub Abstract Syntax Tree(SAST) [49]: SAST also divides the AST into different subtrees, but only eight types of complex nodes are separated into individual subtrees. Other nodes are connected to the original tree structure.

The experimental results indicate that using OAST as the input for MRT achieved the best results in terms of both efficiency and performance. Although DAST and SAST divide the original AST, they do not perform pruning to avoid redundancy present in the AST, making it more difficult for MRT to learn useful information from the AST. Different AST processing methods have little impact on MRT's evaluation time cost, but OAST still has a slight time advantage due to its shorter sequence.

B. The Efficiency Analysis

RQ2: How is the efficiency of our approach? To evaluate the efficiency, we assessed the test time performance of our model on OJClone, GCJ, and BCB, which are shown in Table III. PNIAT [44] is the state-of-the-art model on BCB and GCJ. Our training time on BCB amounted to 4 hours, significantly less than the 5 days required by PNIAT [44].

The experiment results on BCB show our model demonstrated a **5x speed improvement** compared to the state-of-the-art lightweight model PNIAT [44] and a **563x speed improvement** compared to the BERT-based model GraphCodeBERT [26] with only a 0.9% decrease in F_1 score. The faster speed of MRT-OAST is mainly due to its separation of clone detection into two steps: converting codes into vectors and calculating the similarity. P represents the number of clone pairs, and C represents the number of code fragments. The time cost of our method for converting code into vectors is O(C), while the time cost for calculating the similarity is O(P^2). Therefore the total time cost is O($C + P^2$). Although PNIAT [44] also separates the stages of code feature extraction and clone detection, PNIAT has a more complex model structure and requires an LSTM-based model which is O($C \times d^2, d \gg 1$) in the clone detection stage. This makes PNIAT less efficient than our model. Toma [51] is a fast CPU-based method that also separates the two stages, but its overly simplistic model and feature extraction approach result in decreased accuracy. Other models like CTL [9] do not completely separate these two steps, resulting in the entire process being O(P^2). GraphCodeBERT [26] is slower because it has a large number of parameters.

TABLE V: Results of Different Merge and Post-merged Models on OJClone

Group	Merge	Post-merged	P	R	F_1
Only merge	Concat	-	96.2	95.1	95.7
	Max	-	97.3	95.9	96.6
	ConV	-	98.0	96.3	97.2
Post-merged Model	Concat	MLP	98.7	98.6	98.6
	Max	MLP	98.9	98.2	98.6
	ConV	MLP	98.6	98.7	98.7
	Concat	Transformer	98.9	99.2	99.1
	Max	Transformer	99.3	98.9	99.2
	ConV	Transformer	**99.8**	**99.5**	**99.7**

TABLE VI: Results of Multiple Representations and Single Representation on OJClone

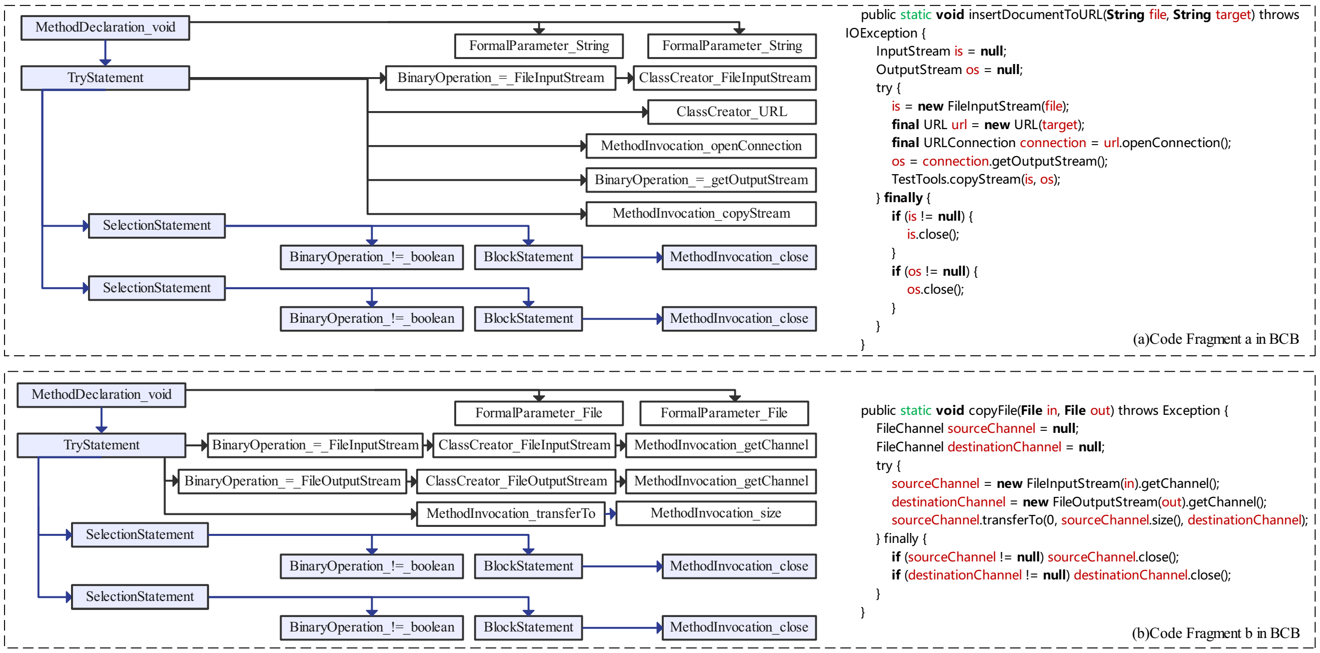
Dataset	Description	P	R	F_1
OJClone	Only Pre	95.1	98.4	96.5
	Only Post	94.3	94.9	94.6
	Pre+Post	**99.8**	**99.5**	**99.7**

Additionally, we conducted experiments to investigate the costs associated with the feature extraction stage and the clone detection stage. The experimental results are presented in Table IV. Removing the Siamese Network leads to our method's inability to distinguish between the two stages, which significantly increases the time cost to O(P^2). Therefore, designing MRT as a pure Siamese Network structure is crucial for ensuring its efficiency. Conversely, using AST instead of OAST slightly increases the time cost of the feature extraction stage and significantly reduces the model's accuracy. The lack of structural information and redundancy in AST makes it difficult for the model to extract features, leading to a decrease in accuracy.

C. Effects of Using Multiple Representations

RQ3: What effects does the use of multiple representations? To further investigate the effectiveness of multiple representations, we conducted experiments to validate different merging strategies.

The results are shown in Table V. The **ConV** represents the default convolution merge model proposed in Section III-B; **Max** and **Concat** respectively replace the convolution by a max pool operation and concatenating the result of the first token from the pre-merged model; **Transformer** represents the default post-merged model proposed in Section III-B;


MethodDeclaration_void
FormalParameter_String
FormalParameter_String
TryStatement
BinaryOperation_=_FileInputStream
ClassCreator_FileInputStream
ClassCreator_URL
MethodInvocation_openConnection
BinaryOperation_=_getOutputStream
MethodInvocation_copyStream
SelectionStatement
BinaryOperation_!=_boolean
BlockStatement
MethodInvocation_close
SelectionStatement
BinaryOperation_!=_boolean
BlockStatement
MethodInvocation_close
(a)Code Fragment a in BCB
MethodDeclaration_void
FormalParameter_File
FormalParameter_File
TryStatement
BinaryOperation_=_FileInputStream
ClassCreator_FileInputStream
MethodInvocation_getChannel
BinaryOperation_=_FileOutputStream
ClassCreator_FileOutputStream
MethodInvocation_getChannel
MethodInvocation_transferTo
MethodInvocation_size
SelectionStatement
BinaryOperation_!=_boolean
BlockStatement
MethodInvocation_close
SelectionStatement
BinaryOperation_!=_boolean
BlockStatement
MethodInvocation_close
(b)Code Fragment b in BCB

```
public static void insertDocumentToURL(String file, String target) throws
IOException {
    InputStream is = null;
    OutputStream os = null;
    try {
        is = new FileInputStream(file);
        final URL url = new URL(target);
        final URLConnection connection = url.openConnection();
        os = connection.getOutputStream();
        TestTools.copyStream(is, os);
    } finally {
        if (is != null) {
            is.close();
        }
        if (os != null) {
            os.close();
        }
    }
}
```

```
public static void copyFile(File in, File out) throws Exception {
    FileChannel sourceChannel = null;
    FileChannel destinationChannel = null;
    try {
        sourceChannel = new FileInputStream(in).getChannel();
        destinationChannel = new FileOutputStream(out).getChannel();
        sourceChannel.transferTo(0, sourceChannel.size(), destinationChannel);
    } finally {
        if (sourceChannel != null) sourceChannel.close();
        if (destinationChannel != null) destinationChannel.close();
    }
}
```

Fig. 7: Two code snippets and their OASTs from BCB constitute a negative clone pair.

MLP replaces the transformer in post-merged model with multilayer perceptron. Compared to only using the merged model, the introduction of the post-merged model led to significant improvements. This suggests that merging can yield additional potential structures. The Transformer consistently outperformed the MLP, as its attention mechanism enables better capturing of long-range dependencies. Ultimately, we opted for the combination of ConV+Transformer, which achieves the best performance.

To validate the effect of feature selection, we experimented to compare the performance of utilizing only the pre-order sequence, post-order sequence, and merging with the learning method. The results are shown in Table VI. **Only Pre** utilizes only pre-order traversed sequences through a Transformer model; **Only Post** utilizes only post-order traversed sequences through a Transformer model; **Pre+Post** utilizes the default model described in Section III-B that combines both pre-order and post-order traversed sequences. Multiple representations supplement the information that cannot be learned from a single representation and serve as a form of regularization, which contributes to superior outcomes.

D. Effects of the Proposed AST Optimization Method

RQ4: What is the effect of the proposed AST optimization?

To validate the effectiveness of AST optimization, we conducted ablation studies to investigate the impact of each process in the AST optimization, including the clean, enhanced operators, and unified logic.

The corresponding AST information and results are recorded in Table VII. The experimental results reveal that the clean process significantly reduces the average number of nodes and average depth of the AST. The integration of

TABLE VII: Results of AST Optimization at Different Steps on the OJClone

AST Optimization	P	R	F_1	Avg.N	Avg.D
No optimization	94.9	96.2	95.6	158	13.9
P1	97.8	97.0	97.4	87	11.4
P1+P2	99.3	99.6	99.4	59	10.9
P1+P2+P3	**99.8**	**99.5**	**99.7**	61	11.0

P1: clean; P2: enhanced operators; P3: unify logic;
Avg.N: Average AST nodes; Avg.D: Average AST depth;

P2 benefits the introduction of missing symbols and types to the original AST's expression nodes, resulting in notable enhancements. This facilitates the model's ability to distinguish a significant number of negative clone pairs originating from different data types, while also allowing the model to extract features from an extensive vocabulary. **P1** improves the efficiency of the self-attention mechanism in the Transformer by reducing the number of AST nodes. **P3** achieves better identification of structurally similar positive clone pairs by unifying branching and loop structures.

OAST works well in detecting most positive clone pairs. However, when negative clone pairs have similar structures, they can generate very similar OAST structures. This causes the model to struggle in extracting the differences in representational information from the tree structures, leading to incorrect classifications. Figure 7 illustrates a case from BCB. In Figure 7(a), the code uploads the contents of a local file to a server location specified by a URL. In Figure 7(b), the code copies the contents of one file to another file. Although they are not clone pairs, they have similar structures. Therefore, our model incorrectly classifies them as a positive clone pair.

In addition, in Figure 8, we analyze the results obtained for various sizes of AST. It can be observed that AST with a node count between 32 and 48 is the most prevalent. Our

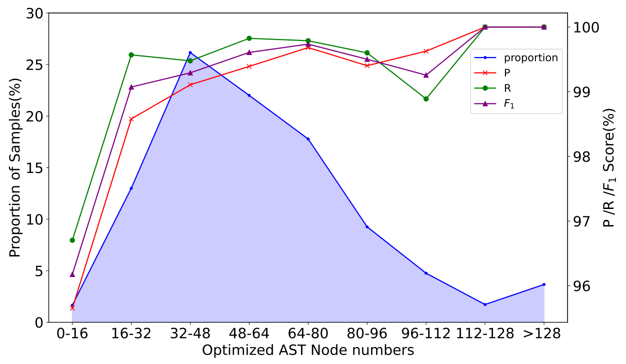


Fig. 8: Evaluation of the performance of our method on code of varying sizes in terms of Precision (P), Recall (R), and F_1-score (F_1).

approach achieved favorable results across ASTs of different sizes. However, there is a slight decrease in performance on very small ASTs, attributed to compilation errors that generated incorrect small ASTs. Overall, our AST optimization demonstrates favorable results across ASTs of varying lengths.

VI. Related work

A. Deep Learning for Code Clone Detection

Traditional code clone detection methods include text-based approaches [53], token-based approaches [3], [54], and AST-based matching methods [52]. With the development of deep learning, there have been approaches that use RNN to process code structures, such as DLC [50], CDLH [13], RAE [12], and ASTNN [22]. Among them, ASTNN [22] is a representative approach that segments a large AST into a series of small statement trees, encodes the statement trees into vector representations, and then passes them through a bidirectional recursive neural network to generate a vector representation for the code fragment. Some methods use DNN as a base model to directly extract AST features, such as CCLearner [17], SSFL [18], TECDD [19], CLCDSA [20], Oreo [21], and DeepSim [15]. For instance, DeepSim [15] attempts to incorporate control flow and data flow encoding using PDGs into the deep learning model. SSFL [18] proposed a categorized, and processes function method, which aims to code dependencies to obtain output vectors through a deep learning model. AT-BiLSTM [14] treats AST as a sequence and obtains output vectors through a bidirectional LSTM and multi-head self-attention model. Recent research often focuses on the characteristics of AST itself. FA-AST [23] is an approach that adds additional edges to the AST to include more information and then obtains output vectors through graph neural networks. Code token learner [9] is an automatic GNN network that compresses AST to a fixed size and obtains output vectors through Transformer and cross attention. PNIAT [44] is an efficient model utilizing parallel node internal multi-head attention based on attention mechanisms to capture key tokens in each statement in parallel. GraphCodeBERT [26] is a BERT-based model used for various code-related tasks, including code clone detection. There are also new methods adopt novel approaches for code clone detection. DSFM [55] utilizes deep clone detectors with deep subtree interactions to compare every two subtrees extracted from ASTs of two code snippets. Prism [56] employs architecture assembly code to detect code clones.

B. Transformer in Software Engineering

Recently, the Transformer-based method [37] has achieved competitive performance across diverse domains [57], [58]. In the software engineering field, its application has been widespread, encompassing tasks such as code repair [59], code analysis [60], code generation [61]–[64], and documentation generation [65]. Specifically, Tfix [59] utilizes pretrained Transformer models for automated bug detection and fixing. Siddiq et al. [60] employed a Transformer to analyze code smells. For code generation, Treegen [66] adopt a tree-based Transformer model to learn code structures. Furthermore, employing copying attention [65] with Transformer facilitates the generation of summary text. Additionally, recent large language models like CodeX [61] and CodeGen [64] have also leveraged Transformer for code generation tasks. These diverse applications underscore the broad scope and effectiveness of Transformer in the software engineering domain.

VII. Threats to Validity

There are two main threats to the validity of our approach. Firstly, our method was tested solely on the Java-based BCB [29] and GCJ [28] datasets, as well as the C/C++ based OJClone [16] dataset. Despite the fact that most programming languages can be translated into an AST and share similar node types, we have yet to verify the model performance in these languages. Therefore, we cannot claim its effectiveness on other programming languages. Another concern is that some of our evaluation time measurements for other models were reported from PNIAT [44], as we both utilized the RTX 3090 24G GPU and a same performing CPU. We believe this comparison is justifiable.

VIII. Conclusion And Future Work

In this paper, we propose a Multiple Representation Transformer with an Optimized Abstract Syntax Tree for code clone detection. Our approach involves an AST optimization method comprising three processes to enhance results. Furthermore, we employed a pure Siamese Network to ensure our model operates at an exceptionally high speed during evaluation. Notably, we simultaneously learn from both the pre-order and post-order representations. Experimental results demonstrate the superiority of our model over existing approaches for OJClone and GCJ. Our model is 5x faster compared to the state-of-the-art lightweight model and 563x faster compared to the BERT-based model with only marginal performance differences observed on BCB. In future work, we plan to incorporate more sequences as input for our model. As well as exploring more effective merge and post-merge techniques to better capture the structural information in our method. Our code and experimental data are publicly available at https://github.com/UnbSky/MRT-OAST

References

[1] Y. Yang, Z. Ren, X. Chen, and H. Jiang, "Structural function based code clone detection using a new hybrid technique," in *2018 IEEE 42nd annual computer software and applications conference (COMPSAC)*, vol. 1. IEEE, 2018, pp. 286–291.

[2] M. Singh and V. Sharma, "Detection of file level clone for high level cloning," *Procedia Computer Science*, vol. 57, pp. 915–922, 2015.

[3] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE transactions on software engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[4] V. Ciesielski, N. Wu, and S. Tahaghoghi, "Evolving similarity functions for code plagiarism detection," in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, 2008, pp. 1453–1460.

[5] L. Muxin, "Research on code plagiarism detection based on code clone detection technologies," in *2021 2nd International Conference on Big Data and Informatization Education (ICBDIE)*. IEEE, 2021, pp. 274–277.

[6] N. Ebrahimi, A. Trabelsi, M. S. Islam, A. Hamou-Lhadj, and K. Khanmohammadi, "An hmm-based approach for automatic detection and classification of duplicate bug reports," *Information and Software Technology*, vol. 113, pp. 98–109, 2019.

[7] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[8] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 117–125.

[9] A. Zhang, L. Fang, C. Ge, P. Li, and Z. Liu, "Efficient transformer with code token learner for code clone detection," *Journal of Systems and Software*, vol. 197, p. 111557, 2023.

[10] R. Tairas and J. Gray, "An information retrieval process to aid in the analysis of code clones," *Empirical Software Engineering*, vol. 14, pp. 33–56, 2009.

[11] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using ir methods for labeling source code artifacts: Is it worthwhile?" in *Proceedings of the 2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 2012, pp. 193–202.

[12] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 87–98.

[13] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code." in *IJCAI*, 2017, pp. 3034–3040.

[14] Y. Meng and L. Liu, "A deep learning approach for a source code detection model using self-attention," *Complexity*, vol. 2020, pp. 1–15, 2020.

[15] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 141–151.

[16] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[17] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 249–260.

[18] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 516–527.

[19] Y. Gao, Z. Wang, S. Liu, L. Yang, W. Sang, and Y. Cai, "Teccd: A tree embedding approach for code clone detection," in *Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. ieee, 2019, pp. 145–156.

[20] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, and K. A. Schneider, "Clcdsa: cross language code clone detection using syntactical features and api documentation," in *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1026–1037.

[21] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 354–365.

[22] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[23] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.

[24] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[25] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[26] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[27] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network," *Advances in neural information processing systems*, vol. 6, 1993.

[28] "Google Code Jam," https://code.google.com/codejam/contests.html, 2016, accessed: 2016-10-8.

[29] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 476–480.

[30] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 368–377.

[31] S. Paul and A. Prakash, "A framework for source code search using program patterns," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 463–475, 1994.

[32] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.

[33] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[34] M. Lan, J. Zhang, F. He, and L. Zhang, "Siamese network with interactive transformer for video object segmentation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 2, 2022, pp. 1228–1236.

[35] P. Yuan, Q. Zhao, X. Zhao, X. Wang, X. Long, and Y. Zheng, "A transformer-based siamese network and an open optical dataset for semantic change detection of remote sensing images," *International Journal of Digital Earth*, vol. 15, no. 1, pp. 1506–1525, 2022.

[36] W. G. C. Bandara and V. M. Patel, "A transformer-based siamese network for change detection," in *Proceedings of the IGARSS 2022-2022 IEEE International Geoscience and Remote Sensing Symposium*. IEEE, 2022, pp. 207–210.

[37] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[38] Q. Wang, B. Li, T. Xiao, J. Zhu, C. Li, D. F. Wong, and L. S. Chao, "Learning deep transformer models for machine translation," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2019.

[39] Y. Kang, S. Kim, and S. Choi, "Deep learning to hash with multiple representations," in *Proceedings of the 2012 IEEE 12th International Conference on Data Mining*. IEEE, 2012, pp. 930–935.

[40] C. Lin, F. Lee, L. Xie, J. Cai, H. Chen, L. Liu, and Q. Chen, "Scene recognition using multiple representation network," *Applied Soft Computing*, vol. 118, p. 108530, 2022.

[41] H. Su, S. Maji, E. Kalogerakis, and E. Learned-Miller, "Multi-view convolutional neural networks for 3d shape recognition," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 945–953.

[42] X. Zhai, Y. Peng, and J. Xiao, "Heterogeneous metric learning with joint graph regularization for cross-media retrieval," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 27, no. 1, 2013, pp. 1198–1204.

[43] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 542–553.

[44] D. Yu, Q. Yang, X. Chen, J. Chen, and Y. Xu, "Graph-based code semantics learning for efficient semantic code clone detection," *Information and Software Technology*, vol. 156, p. 107130, 2023.

[45] Y. Wu, D. Zou, S. Dou, S. Yang, W. Yang, F. Cheng, H. Liang, and H. Jin, "Scdetector: Software functional clone detection based on semantic tokens analysis," in *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, 2020, pp. 821–833.

[46] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th conference on program comprehension*, 2018, pp. 200–210.

[47] C. Lattner, "Llvm and clang: Next generation compiler technology," in *Proceedings of the BSD conference*, vol. 5, 2008, pp. 1–20.

[48] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[49] T. Hu, Z. Xu, Y. Fang, Y. Wu, B. Yuan, D. Zou, and H. Jin, "Fine-grained code clone detection with block-based splitting of abstract syntax tree," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 89–100.

[50] A. Sheneamer and J. Kalita, "Semantic clone detection using machine learning," in *2016 15th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2016, pp. 1024–1028.

[51] S. Feng, W. Suo, Y. Wu, D. Zou, Y. Liu, and H. Jin, "Machine learning is all you need: A simple token-based approach for effective code clone detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[52] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 96–105.

[53] B. BAKER, "A program for identifying duplicated code," *Computing Science and Statistics*, vol. 24, pp. 49–57, 1992.

[54] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1157–1168.

[55] Z. Xu, S. Qiang, D. Song, M. Zhou, H. Wan, X. Zhao, P. Luo, and H. Zhang, "Dsfm: Enhancing functional code clone detection with deep subtree interactions," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 1005–1005.

[56] H. Li, S. Wang, W. Quan, X. Gong, H. Su, and J. Zhang, "Prism: Decomposing program semantics for code clone detection through compilation," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 1001–1001.

[57] L. Yuan, J. Wang, L.-C. Yu, and X. Zhang, "Encoding syntactic information into transformers for aspect-based sentiment triplet extraction," *IEEE Transactions on Affective Computing*, 2023.

[58] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 2023, pp. 31–53.

[59] B. Berabi, J. He, V. Raychev, and M. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *Proceedings of the International Conference on Machine Learning*. PMLR, 2021, pp. 780–791.

[60] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. Santos, "An empirical study of code smells in transformer-based code generation techniques," in *Proceedings of the 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2022, pp. 71–82.

[61] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[62] L. Cao, Y. Cai, J. Wang, H. He, and H. Huang, "Beyond code: Evaluate thought steps for complex code generation," in *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, 2024, pp. 2296–2306.

[63] J. Wang, L. Cao, X. Luo, Z. Zhou, J. Xie, A. Jatowt, and Y. Cai, "Enhancing large language models for secure code generation: A dataset-driven study on vulnerability mitigation," *arXiv preprint arXiv:2310.16263*, 2023.

[64] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[65] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 4998–5007.

[66] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 05, 2020, pp. 8984–8991.