International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2021, 11(3): 357–378, doi: 10.21655/ijsi.1673-7288.00263

International Journal of Software and Informatics

Research Article

Structurally-Enhanced Approach for Automatic Code Transformation

Yingkui Cao (曹英魁)[1,2], Zeyu Sun (孙泽宇)[1,2], Yanzhen Zou (邹艳珍)[1,2], Bing Xie (谢冰)[1,2]

[1] (School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

[2] (Key Laboratory of High Confidence Software Technologies of Ministry of Education (Peking University), Beijing 100871, China)

Corresponding author: Bing Xie, xiebing@sei.pku.edu.cn

Abstract In software development, developers often need to change or update lots of similar codes. How to perform code transformation automatically has become a research hotspot in software engineering. An effective way is extracting the modification pattern from a set of similar code changes and applying it to automatic code transformation. In the related work, deep-learning-based approaches have achieved much progress, but they suffer from the problem of significant long-term dependency between the codes. To address this challenge, an automatic code transformation method is proposed, namely ExpTrans. Based on the graph-based representations of code changes, ExpTrans is enhanced with the structural information of code. It labels the dependency between variables in code parsing and adopts the graph convolutional network and Transformer structure to capture the long-term dependency between the code. ExpTrans is first compared with existing learning-based approaches to evaluate its effectiveness; the results show that ExpTrans gains 11.8%–30.8% precision increment. Then, it is compared with rule-based approaches and the results demonstrate that ExpTrans significantly improves the correct rate of the modified instances.

Keywords code change; software evolution; software maintenance; code generation

Citation Cao YK, Sun ZY, Zou YZ, Xie B. Structurally-enhanced approach for automatic code transformation, *International Journal of Software and Informatics*, 2021, 11(3): 357–378. http://www.ijsi.org/1673-7288/263.htm

1 Introduction

In software development, developers often need to change similar fragments of code. However, changing similar code is time-consuming and makes developers liable to introduce new mistakes when they finish the repeated tasks[1, 2]. Based on different version control systems, the commit histories of software projects have been fully documented. In these committed code changes, abundant similar code change scenarios and schemes can be found, from which change

This is the English version of the Chinese article "一种结构信息增强的代码修改自动转换方法. 软件学报, 2021, 32(4): 1006–1022. doi: 10.13328/j.cnki.jos.006227".

Funding items: National Science Fund for Distinguished Young Scholars (61525201); National Natural Science Foundation of China (61972006)

Received 2020-09-13; Revised 2020-10-26; Accepted 2020-12-19; IJSI published online 2021-09-25

patterns are extracted and used to automatically change similar code, called automatic code change transformation.

In previous work, automatic code change transformation was often carried out based on manual feature engineering[7–11], namely that researchers manually propose rules to represent change patterns and conduct transformation. However, these rules are often based on the researchers' expert knowledge of a particular programming language and require much time and effort for summary and refinement. Recently, learning-based transformation approaches[12, 13] have been emerging. A common practice is to use an end-to-end translation model, where the code to be modified is "translated" into the modified code. However, the existing work has not fully used the structural information of the modified code instances. On the one hand, some existing work utilizes translation models to translate the code to be modified directly into the modified code. However, in the absence of modified instances, trying to train a global translation model for code transformation is undoubtedly difficult. On the other hand, compared with natural language, code has significant long dependency between the information within it. As shown in Figure 1, the variable name *fis* in the function call *fis.close*() is the same as the one originally declared. However, existing approaches are often proposed based on recurrent models, which are not good at capturing the long dependency between code statements where variable *fis* is declared and used.

```
FileInputStream fis = new FileInputStream(new File(dir));
try {
    ...
    fis.close();
} catch(IOException e) {
    ...
}
```

Figure 1 Long dependency between variable names

To solve these problems, this paper proposes an approach, ExpTrans, which uses structural information to enhance code transformation. ExpTrans takes input as x and $x_\Delta \rightarrow y_\Delta$, where x is code which needs to be modified and the modified instance $x_\Delta \rightarrow y_\Delta$ consists of pre-modified code x_Δ and post-modified code y_Δ. ExpTrans outputs code y as the result of modifying x. On the one hand, ExpTrans parses x_Δ and y_Δ into ASTs (Abstract Syntax Trees) and looks for the correspondence between their nodes. Based on the correspondence, the given modified instances are represented by graphs which are used to enhance the ability of ExpTrans to capture the structural information of the modified instance. On the other hand, ExpTrans combines Graph Convolutional Network and Transformer structure[14] to enhance the model's ability to capture the dependency between code, especially the long-term dependency.

ExpTrans is based on an encoder-decoder architecture. The encoder encodes the information of x, $x_\Delta \rightarrow y_\Delta$ and internal state information. The decoder predicts the modified code y according to the encoder's result. To ensure the generated code y can be compiled, ExpTrans generates the code by predicting the rule sequence like the works of Yin *et al*.[15] and *et al*. [16] did. A rule is like $\alpha \rightarrow \beta_1\beta_2\cdots\beta_n$, which means to expand a node α on AST with n child nodes whose type is $\beta_1, \cdots, \beta_n$, respectively. Specifically, a abstract syntax tree representing the internal state will be maintained, and based on the next predicted rule, the current leftmost node to be expanded is processed until all non-leaf nodes are expanded. The code corresponding to the generated abstract syntax tree is the transformed one.

To verify the effectiveness of ExpTrans, this paper conducts a comparative experiment with two data sets. The first data set includes 111,724 C# code modifications[13] open to the public by Yin *et al*.[13]. The experimental results show that compared with the work of Yin (based on

deep learning)[13], ExpTrans gains an 11.8%–30.8% precision increment. The second data set comprises six groups of typical similar code modifications in the Java programming language collected from GitHub. ExpTrans, the GenPat approach[17], and the ARES approach[10] (the latter two based both on artificial rules) are used to automatically modify the code instances in the second data set. The experimental results show that there are instances correctly modified by ExpTrans in each group, and all the instances in each group can be correctly modified by ExpTrans. Compared with the results of GenPat and ARES, the results obtained by ExpTrans have been greatly improved in the correct rate of modified instances.

The contributions of this paper are mainly manifested in the following aspects:

(1) A graph-based representation approach for code modification is proposed. Compared with the representation by word sequences, the graph-based structure can more accurately represent the modification process, which has facilitated the capture of structural information in code modification.

(2) An automatic code transformation method based on the structural information enhancement of the Transformer structure is proposed. This approach uses a special copy rule to explicitly express the extensive dependency between variable declarations and usage different fragments of code, enhancing the ability of the model to capture the long-term dependency between the code.

(3) This paper carries out comparative experiments with two data sets and makes all the data public (`https://github.com/caoyingkui/ExpTrans`). Compared with the existing machine learning-based approaches, ExpTrans gains an 11.8%–30.8% precision increment. Moreover, it significantly improves the correct rate of modified instances, compared with the rule-based approaches.

In this paper, Section 2 introduces the existing related work. Section 3 illustrates the code generation model based on predicted rule sequences and the overall framework of the approach proposed in this paper. Section 4 shows the specific implementation details of the approach. Section 5 presents the experiments for verifying the effectiveness of the approach, the experimental settings, and results. Finally, Section 6 summarizes the work of this paper.

2 Related Work

As mentioned above, the related work of this paper is divided into two categories, i.e., the approaches based on artificial features and the automatic code transformation methods based on deep learning.

2.1 Rule-based approaches

The main idea of these approaches is as follows: Based on the rules of artificial extraction, a code transformation "script" is extracted from the given modified instances to explain and restrict the modification conditions, patterns, and processes in the instances. Moreover, according to the approach agreed in the script, this script is automatically matched to the qualified code area to complete code transformation. In the existing work, this script is presented in multiple forms, such as the code editing sequence[7], template[8], and Domain-Specific Language (DSL)[11].

In SYDIT proposed by Meng *et al*., an edit operation sequence is used to represent the process of code transformation[7]. The edit operations include four types: addition, deletion, update, and move of AST nodes. SYDIT leverages wildcards to generalize the class type and location in the operation. For example, !*config.inValid*() is represented as !*v*2.*m*5(). The operations in the sequence are performed successively to modify the code. LASE is another code transformation method proposed by Meng *et al*.[9]. It also uses a set of editing operation

sequences to represent the code transformation in the sample code. Different from SYDIT, LASE extracts the code transformation from multiple similar modified instances.

In addition, some existing works use templates to represent the patterns in modified instances. For example, spdiff extracts a term replacement patch from the modified instance to characterize the code modification in the instance and takes the longest common sub-patch in a group of modified instances as the code transformation[8]. Dotzler *et al*. proposed ARSE for code transformation based on multiple instances[10]. However, unlike LASE, ARSE uses a template to represent the pattern in the modified instances. Jiang *et al*. proposed GenPat for extracting the code transformation pattern from a single modified instance[17]. With this approach, the code transformation is finally represented as a tree structure. The information of each node in the tree includes the node *ID* and a set of attribute values. At the same time, the representation result of GenPat also contains a set of operations, each of which is a tuple $\langle id, id' \rangle$. id and id' respectively represent a node in the AST of the code before and after modification, namely that the node id is modified to the node id'.

In the REFAZER proposed by Rolim *et al*., a special Domain-Specific Language (DSL) is defined[11]. Its main function is to define the operation of the AST and restrict the conditions for the AST that meets the specific modification as well as the modification location and type. Therefore, the goal is generating a piece of code based on DSL. The input of the generated code is a piece of code before modification, and its output is a piece of modified code.

2.2 Learning-based approaches

The main idea of learning-based approaches is giving the code to be modified and using the machine learning model to predict the modified code. Tufano *et al*. proposed an approach for code transformation based on the translation model[12]. Specifically, the model takes as input a given piece of code to be modified, and predicts a token sequence as the modified code. At the same time, the approach also explores types of code modification scenarios the translation model is suitable for, such as defect modification and code reconstruction. The function and compilation of code strictly depend on a specific token sequence of the code and the positional relationship between the tokens. However, the translation model cannot guarantee that the output of the model can be compiled. To overcome this problem, the approach proposed by Yin and Neubig generates code by predicting the rule sequence to ensure the grammatical correctness of the output code[15]. Based on this mechanism, Yin *et al*. proposed a learning-based code transformation model[13]. In addition to code to be modified, the model also input modification instance. That is, the code transformation is realized by learning the code transformation from the modified instance.

In summary, the approach proposed by Yin *et al*. is the most relevant approach with this paper in the existing work. They use the LSTM model to process the text information of code. Compared with natural language, the code has more significant long-term dependencies, but studies have shown that the time series model is not effective in dealing with long-term dependency[14]. Therefore, overcoming the long-term dependency of code is one of the main reasons for the approach proposed in this paper.

3 Approach Framework

To ensure the code generated by the approach can be compiled, this paper draws attention on the work of Yin and Neubig[15] and uses the approach of predicting rule sequences instead of term sequences to generate code. At the same time, our approach adopts special copy rules, so as to explicitly record the dependencies between the variables in the code. This section will first briefly describe code generation based on predicting rule sequence.

Code rules: In this paper, a rule takes form as $\alpha \rightarrow \beta_1\beta_2\cdots\beta_n$; α is a non-terminal symbol while β_i is a terminal symbol or a non-terminal symbol. In the abstract syntax tree of code, each non-leaf node corresponds to a rule $\alpha \rightarrow \beta_1\beta_2\cdots\beta_n$, where α is the syntax type of the node while β_i is the syntax type (or term) of its child nodes.

Acquisition of rule set: Assuming that the node v in a given AST has the syntax type of α and n child nodes with the syntax types (or terms) of $\beta_1, \cdots, \beta_n$, respectively, then the rule $r : \alpha \rightarrow \beta_1\cdots\beta_n$ can be obtained. All non-leaf nodes of the abstract syntax tree are traversed from top to bottom and from left to right to obtain the corresponding code rule sequence. At the same time, the rule set $\{r_1, \cdots, r_N\}$ can be obtained by traversing the abstract syntax tree of the code in the experimental data set.

Since developers can use any legal variable names and strings in code, the code has a more significant Out-Of-Vocabulary (OOV) problem than natural language. Therefore, our approach restricts and preprocesses the tokens in the rules to avoid the blow-up of the final rule set. Specifically, before acquiring the rule, our approach counts all variable names, strings, numeric constants, and character constants, as well as their number of occurrences in the code from the experimental data set. One token will be reserved only when the number of its occurrences exceeds a given threshold. When the number of occurrences of a token is lower than the threshold, our approach will replace it in the form of type_id, where *type* represents the tokens' type, namely variable name, string, numeric constant, or character constant; id indicates the serial number. In this way, it can be ensured that the size of the final rule set is within a limited range.

In addition, our approach includes a special copy rule in the rule set. In code, a clear dependency can be found between variable declaration and usage. Our approach uses the copy rule to explicitly record the dependency between variable declaration and usage. As shown in Figure 1, the variable name *fis* in the statement "*fis.close*();" is declared by the statement "*FileInputStream fis*=new *FileInputStream*(new *File*(*dir*));". Therefore, when our approach parses *fis.close*(), the copy rule is used to mark that the variable name *fis* is derived from copying the previously defined *fis*. There are two main advantages of using the copy rule. The first one is that the copy rule explicitly records the dependency between variables, which is conducive to enhancing the ability of subsequent models to capture the dependency between variables. The second is that the scope of the copy variable of the copy rule is the set of variable names defined by the code. Therefore, the prediction space faced by the approach in the subsequent prediction process is the set of variable names defined by the code. Compared with the entire term space, the copy rule narrows the prediction space when predicting variable names, thereby improving the correct rate of the approach.

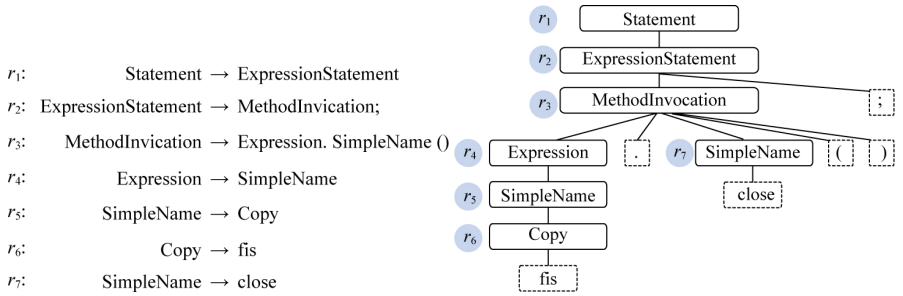
The code generation based on predicting rule sequence. Figure 2 shows the process of generating the code "*fis.close*();" based on the rule sequence $[r_1, \cdots, r_7]$. Given the rule sequence of Figure 2(a), an abstract syntax tree with only the root node is initiated, and its type is Statement. Based on the first rule, Statement → ExpressionStatement, a child node with the type of ExpressionStatement is added to the root node. As shown in Figure 2(b), when the first three rules are completed, the lowermost layer of the current abstract syntax tree will contain two nodes to be expanded, i.e. Expression and SimpleName. Because ".", "(", and ")" are terminal symbols, they will not be expanded in the subsequent process. At this time, the next rule r_4 is used to expand the leftmost node (non-terminal symbol) to be expanded, namely the Expression node on the left. According to the above approach, a complete abstract syntax tree will eventually be generated. The code is generated by obtaining all the leaf node content from left to right.

Based on the above approach, this paper will implement automatic code transformation by

predicting the rule sequence. The input are x and $x_\Delta \rightarrow y_\Delta$, where x is the code to be modified, and $x_\Delta \rightarrow y_\Delta$ represents a modified instance (x_Δ and y_Δ represent the code before and after modification, respectively). The final output, $\bar{y}$, represents the modified code generated by our approach. In the process of generating code $\bar{y}$, an abstract syntax tree that represents the internal state will be maintained. By predicting the next rule, the leftmost non-leaf node will be expanded until the final code is generated. The probability of generating code $\bar{y}$ are calculated as follows:

$$p(\bar{y}) = \prod p(r_i | x, x_\Delta \rightarrow y_\Delta, r_1, \cdots, r_{i-1}) \tag{1}$$

where $r_1, \cdots, r_i$ is the generated rule sequence.



(a) Instruction sequences of *fis.close*(); (b) Generation of *fis.close*();

Figure 2 Rule sequences and generation of *fis.close*();

Figure 3 shows the overall framework of ExpTrans as introduced in this paper.

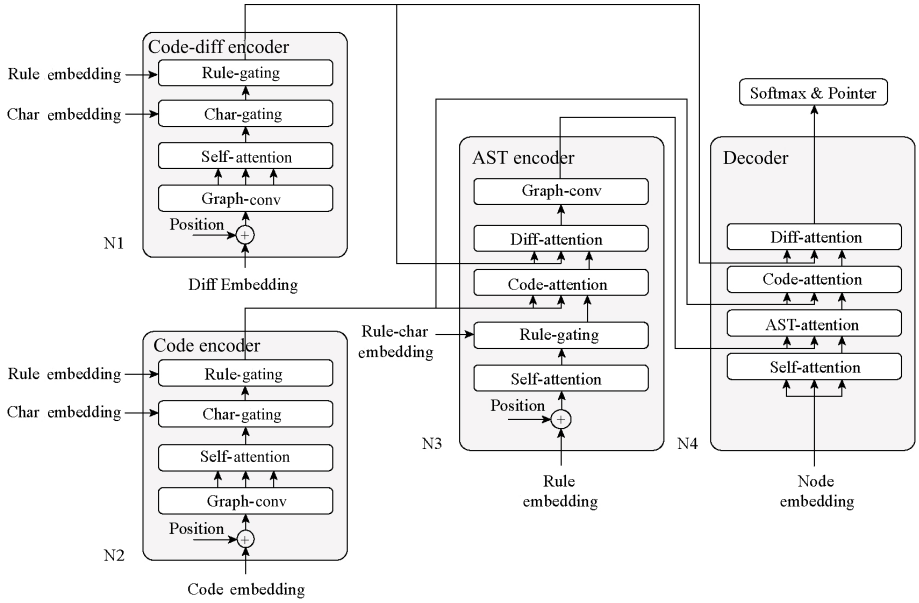


Figure 3 Neural network of ExpTrans

The approach follows the encoder-decoder architecture, which mainly includes four modules, i.e., a code-diff encoder, a code encoder, an AST encoder, and a decoder. These four modules all adopt the multi-block Transformer structure[14], namely that each module is

composed of multiple blocks with the same structure. For example, in the code-diff encoder, each block is composed of the same neural network, including the Graph-conv layer, the Self-attention layer, the Char-gating layer, and the Rule-gating layer, where the output of the previous block is taken as the input of the next for connection. Residual connection[18] is adopted in the internal structure of each block, and adjacent network layers (such as the Graph-conv layer and the Self-attention layer) are connected. It should be noted that Figure 3 only shows one block of each module. The main functions of the four modules are as follows:

- Code-diff encoder: It models the information of the input modified instance $x_\Delta \rightarrow y_\Delta$.
- Code encoder: It models the input information of code x to be modified.
- AST encoder: During code generation, an abstract syntax tree representing the internal state needs to be maintained. This module models the information of the abstract syntax tree to provide the global syntax tree information while predicting the code rules.
- Decoder: During code generation, the decoder will predict the next rule according to the node to be expanded in the generated abstract syntax tree. Specifically, the node information to be expanded is taken as a query and input in the decoder. Based on the input query, the decoder adopts an attention mechanism to combine the modeling information of the code encoder, code-diff encoder, and AST encoder. Then, according to the output of the decoder, the next rule is predicted by the proposed approach in combination with softmax and pointer network[19].

4 Approach

Given the code x to be modified and the modified instance $x_\Delta \rightarrow y_\Delta$, the goal of ExpTrans is to realize the code transformation $\bar{y} = \text{Trans}(x, x_\Delta \rightarrow y_\Delta)$, where $\bar{y}$ is code after modification predicted by ExpTrans. We will describe the details of the different parts of ExpTrans in this section.

4.1 Code-diff encoder

The code-diff encoder is used to model the modification and code structure information of $x_\Delta \rightarrow y_\Delta$. Given $x_\Delta \rightarrow y_\Delta$, the code before and after modification (x_Δ and y_Δ, respectively) is represented in the form of ASTs. The node sequences of these two ASTs are obtained from top to bottom and from left to right, and recorded as $[v_1^{(\text{ori})}, \cdots, v_{L^{(\text{ori})}}^{(\text{ori})}]$ and $[v_1^{(\text{mod})}, \cdots, v_{L^{(\text{mod})}}^{(\text{mod})}]$, separately. Then the two sequences are merged into one sequence, recorded as $V = [v_1, \cdots, v_{L^{(\text{ori})}}, \cdots, v_{L^{(\text{ori})}+L^{(\text{mod})}}, \cdots, v_L]$, where L is the pre-defined maximum length; the first $L^{(\text{ori})}$ nodes are the node sequence before modification and the following $L^{(\text{mod})}$ nodes are the node sequence after modification. When the lengths of node sequences before and after modification are less than L, a special placeholder symbol, < *EMPTY* >, is used for expansion. The code-diff encoder models the modified instance $x_\Delta \rightarrow y_\Delta$ as a node sequence $[v_1, \cdots, v_L]$, and the output of the code-diff encoder is $Y^{(\text{diff})} = [y_1^{(\text{diff})}; \cdots; y_L^{(\text{diff})}]^{\text{T}}$, where $y_i^{(\text{diff})} \in \mathbb{R}^H$ is the representation vector of node v_i, and H is the embedding size (128 in the approach).

4.1.1 *Graph-conv*

The modified instance $x_\Delta \rightarrow y_\Delta$ takes the form of separated fragments of code before and after modification. Separately modeling x_Δ and y_Δ will lose the relations between the pieces of code before and after modification and their structural information. For this, ExpTrans represents $x_\Delta \rightarrow y_\Delta$ as a unified graph $G = \langle V, E \rangle$, where the node set V is the set of abstract syntax tree nodes corresponding to x_Δ and y_Δ, and E is the set of linked edges between nodes.

To establish the linked edge between the x_Δ and y_Δ nodes, ExpTrans uses GumTree[20] to obtain the corresponding relationship between the nodes before and after modification. GumTree

is a code-difference extraction tool based on ASTs. It first obtains the abstract syntax trees, $tree_x$ and $tree_y$, corresponding to the code x_Δ and y_Δ before and after modification, respectively. Then, the syntax type and text information of the $tree_x$ and $tree_y$ nodes are compared in a bottom-top order and the similarity of the nodes is calculated accordingly to obtain the optimal matching relationship between $tree_x$ and $tree_y$ nodes. On this basis, the code modification process can be accurately deduced.

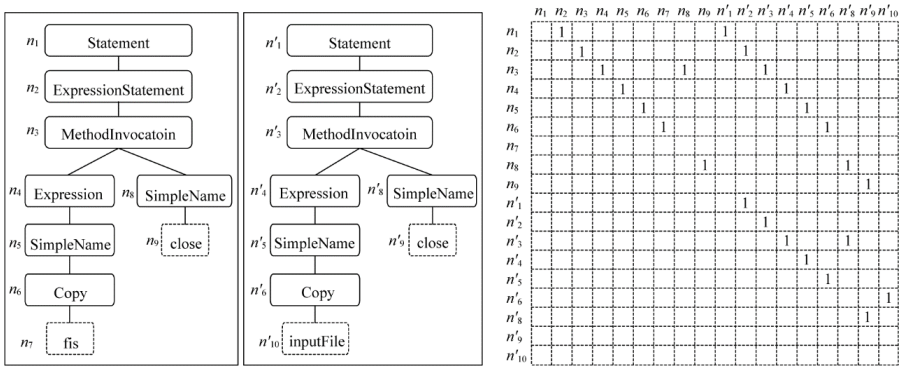
In this paper, if there is an edge from v_j to v_i, node v_j is called the parent node of node v_i. Further, the parent node of v_j is the grandparent node of v_i. According to the outcome of GumTree, when nodes v_i and v_j satisfy one of the following three relations, a linked edge from v_j to v_i is built, and the edge is added to the set E.

(1) Nodes v_i and v_j are both on $tree_x$, and v_j is the parent node of v_i.

(2) Nodes v_i and v_j are both on $tree_y$, and v_j is the parent node of v_i.

(3) Nodes v_i and v_j are on $tree_y$ and $tree_x$, respectively, and they have a correspondence according to the results of GumTree.

Based on the set E, the adjacency matrix M between nodes can be constructed. When v_j is the parent node of v_i, $M[i][j] = 1$; otherwise, $M[i][j] = 0$. For example, the code "*fis.close*();" is modified into "*inputFile.close*();", and Figure 4 shows a graph structure to present the process and results of code modification. As shown in Figure 4(a), the codes before and after modification are represented in the form of abstract syntax trees (symbols such as "." and "(" are omitted for ease of display). Based on the obtained abstract syntax trees, GumTree is used to obtain the correspondence between the abstract syntax tree nodes of the codes before and after modification. In Figure 4(a), nodes n_i and $n_i^{'}$ represent a correspondence between two nodes. Finally, the graph structure of code is represented as the adjacency matrix shown in Figure 4(b).



(a) Abstract syntax trees corresponding to "*fis.close*();" and "*inputFile.close*();"

(b) Adjacency matrix

Figure 4 Construction of the code-differential adjacency matrix

4.1.2 *Extraction of encoding information*

Before calculating $Y^{(\text{diff})}$, the code-diff encoder first obtains the initial information of different aspects of each node.

Word embedding: In the AST, each non-leaf node has a specific syntax type, and it corresponds to a token in the code. Based on the node set V of graph G, a word sequence $[w_1, \cdots, w_L]$ can be obtained. When v_i is a non-leaf node, w_i is the syntax type of v_i. When v_i is a leaf node, w_i is the token which v_i corresponds to in the node. Through embedding, each

word w_i has an initial representation vector, $y_i^{(w)} \in \mathbb{R}^H$; the word embedding corresponding to the node sequence $[v_1, \cdots, v_L]$ is $Y^{(w)} = [y_1^{(w)}; \cdots; y_L^{(w)}]$.

Character embedding: Some semantically similar words have the same character sequence, such as the words derived from the same root. The above approach encoding the information of the word as a whole will discard the character information of the terms. To introduce the character information of the term into the representation vector of nodes, ExpTrans uses the approach proposed by Sun *et al*. to encode the character information of words[16]. Specifically, based on $[w_1, \cdots, w_L]$, each word w_i is split into a character sequence $[c_{i,1}, \cdots, c_{i,L'}]$, where L' is the pre-defined maximum length of the term preset by ExpTrans. Similarly, ExpTrans randomly obtains an initial representation vector $y_{i,j}^{(\text{char})} \in \mathbb{R}^H$, for each character $c_{i,j}$ and leverage the fully connected layer to calculate the character representation vector of term w_i according to Eq. (2):

$$y_i^{(\text{char})} = W^{(\text{char})}[y_{i,1}^{(\text{char})}; \cdots; y_{i,L'}^{(\text{char})}] \tag{2}$$

where $y_i^{(\text{char})} \in \mathbb{R}^H$, and $W^{(\text{char})}$ is the network parameter. Then the character embedding information corresponding to the node sequence $[v_1, \cdots, v_L]$ is $Y^{(\text{char})} = [y_1^{(\text{char})}; \cdots; y_L^{(\text{char})}]$.

Rule embedding: In the AST, each non-leaf node v_i corresponds to a rule r_i: $\alpha^{(i)} \rightarrow \beta_1^{(i)} \cdots \beta_{n_i}^{(i)}$, where $\alpha^{(i)}$ is the syntactic type of node v_i, and all $\beta^{(i)}$ are the syntax types or tokens of the child nodes of node v_i. Similarly to Eq. (2), each rule r_i: $\alpha_i \rightarrow \beta_1^{(i)} \cdots \beta_{n_i}^{(i)}$ is represented with a vector $y_i^{(\text{rule})} \in \mathbb{R}^H$. In addition, the rule corresponding to the leaf node is replaced by <*EMPTY_RULE*>. Therefore, the rule embedding information corresponding to the node sequence $[v_1, \cdots, v_L]$ is $Y^{(\text{rule})} = [y_1^{(\text{rule})}; \cdots; y_L^{(\text{rule})}]$.

Position embedding: In the Transformer structure, the sequential data (such as the representation vector of the node sequence of code) is packed into a vector matrix so that the model can be trained in parallel. However, it will lose the position information of the data. In this paper, the approach of Dehghani *et al*.[21] is used to artificially construct the position information of each node with the following formula:

$$p_{b,i}[2j] = \sin((i+b)/(10000^{2j/H})) \tag{3}$$

$$p_{b,i}[2j+1] = \cos((i+b)/(10000^{2j/H})) \tag{4}$$

where $p_{b,i} \in \mathbb{R}^H$ represents the location information of the i-th node in the b-th block. Then, the position information corresponding to the node sequence $[v_1, \cdots, v_L]$ is $P^{(\text{diff})} = [p_{b,1}; \cdots; p_{b,L}]$.

4.1.3 *Network structure of the code-diff encoder*

Graph-conv: After $x_\Delta \rightarrow y_\Delta$ is represented as a node sequence, ExpTrans uses a graph convolution layer to capture the structural information of the code.

Based on the graph $G = \langle V, E \rangle$ and the adjacency matrix M, the representation vector matrix of the current node is $F = [f_1; \cdots; f_L]$, and then the representation vector of the parent node of each node will be calculated according to Eq. (5):

$$[f_1^{(\text{par})}; \cdots; f_L^{(\text{par})}] = [f_1; \cdots; f_L]M \tag{5}$$

where $f_i^{(\text{par})} \in \mathbb{R}^H$ is the representation vector of the parent node of the i-th node; $[f_1; \cdots; f_L]M^2$ is the representation vector of the grandparent node of the node, and so on. Convolution

is performed as follows:

$$Conv(F, M) = f(W^{(\text{conv})}[F; FM; \cdots; FM^{K-1}]) \tag{6}$$

where K is the window size, $W^{(\text{conv})}$ is the convolution parameter, and f is the ReLU activation function.

The input of this layer is the sum of the representation vector and the position embedding of the node, namely $I = [x_{b,1} + p_{b,1}; \cdots; x_{b,L} + p_{b,L}]$; the output of this layer is

$$Y^{(\text{conv})} = Conv(I, M) \tag{7}$$

where when $b = 1$ (the first block), $x_{b,i}$ represents the word embedding ($y_i^{(w)}$) of the corresponding node; for other blocks, $x_{b,i}$ is the output of the previous block.

Self-attention: This layer uses the self-attention mechanism in *Transformer* to capture long-term dependency in code[14].

In *Transformer*, the attention mechanism is expressed as the process of mapping query Q and key-value pair K and V to the output, where Q, K, and V are all vector matrixes. The output is the result of the weighted summation of the components in V. The corresponding weight of each value in V will be given according to the degree of matching between Q and the corresponding keyword K; the result of the weighted summation is

$$Attention(Q, K, V) = softmax\left(\frac{QK^{\mathrm{T}}}{\sqrt{d_k}}\right)V \tag{8}$$

where d_k denotes the length of each features vector. At the same time, *Transformer* uses a multi-head attention mechanism, which enables the model to notice information at different locations from various perspectives of the representation space, namely

$$Multihead(Q, K, V) = concat(head_1, \cdots, head_h)W^O \tag{9}$$

where $head_i = Attention(QW_i^{(Q)}, KW_i^{(K)}, VW_i^{(V)})$, h is the number of heads, and W^O, $W_i^{(Q)}$, $W_i^{(K)}$, and $W_i^{(V)}$ are all model parameters.

When Q, K, and V are the same, namely in the form of *Multihead* (Q, Q, Q), it is called self-attention. This layer implements the self-attention mechanism; the output of this layer is

$$Y^{(\text{self})} = Multihead(Y^{(\text{conv})}, Y^{(\text{conv})}, Y^{(\text{conv})}) \tag{10}$$

Char-gating: Before this layer, the node sequence is represented as a vector matrix $Y^{(\text{self})}$. To capture character information, a gating mechanism is used in this layer so that the representation vector matrix of the node is updated to the weighted sum of $Y^{(\text{self})}$ and $Y^{(\text{char})}$.

Given the representation vectors f_1 and f_2, the goal of the gating mechanism is to perform the weighted summation of f_1 and f_2 to obtain the representation vector $f^{(\text{gate})}$, where the weights of f_1 and f_2 are calculated according to Eq. (11):

$$[\gamma_1, \gamma_2] = weight(f_1, f_2) = softmax\{f_1, f_2\} \tag{11}$$

Then the weighted result $f^{(\text{gate})}$ is

$$f^{(\text{gate})} = gate(f_1, f_2) = [\gamma_1, \gamma_2] \times [f_1, f_2]^{\mathrm{T}} \tag{12}$$

Similarly, given the vector matrixes $F_1 = [f_{1,1}; \cdots; f_{1,|F_1|}]$ and $F_2 = [f_{2,1}, \cdots, f_{2,|F_2|}]$, which satisfy $|F_1| = |F_2|$, the weighted result $Gate(F_1, F_2)$ of the two by the gating mechanism is

$$Gate(F_1, F_2) = [gate(f_{1,1}, f_{2,1}); gate(f_{1,2}, f_{2,2}); \cdots] \tag{13}$$

To ensure that $Y^{(\text{self})}$ is still the main information of the weighted result, the control vector $C^{(\text{self})}$ is obtained through the linear transformation of $Y^{(\text{self})}$. Then the weighted results of $Y^{(\text{self})}$ and $Y^{(\text{char})}$ are calculated according to Eq. (14) as the output of this layer:

$$Y^{(\text{char-gate})} = Gate(Y^{(\text{self})^{\mathrm{T}}} C^{(\text{self})}, Y^{(\text{char})^{\mathrm{T}}} C^{(\text{self})}) \tag{14}$$

Rule-gating: Similarly, this layer also adopts a gating mechanism to capture rule embedding information. The output of this layer is

$$Y^{(\text{diff})} = Gate(Y^{(\text{char-gate})^{\mathrm{T}}} C^{(\text{char-gate})}, Y^{(\text{rule})^{\mathrm{T}}} C^{(\text{char-gate})}) \tag{15}$$

where $C^{(\text{char})}$ is the control vector obtained by the linear transformation of $Y^{(\text{char-gate})}$.

Finally, the output of the code-diff encoder is $Y^{(\text{diff})}$.

4.2 Code encoder

ExpTrans uses a code encoder to model the information of code x. As shown in Figure 3, the code encoder and the code-diff encoder have the same neural network while different ways of obtaining the input of this layer.

The code x is parsed into an AST $tree_x$ and the node sequence $V^{(x)} = [v_1^{(x)}, \cdots, v_L^{(x)}]$ is obtained in a top-to-bottom, left-to-right manner. Based on $tree_x$, the graph $G^{(x)} = \langle V, E \rangle$ is constructed, where $V = V^{(x)}$; E is the set of linked edges of nodes in $tree_x$. In addition, the adjacency matrix $M^{(x)}$ between nodes is constructed.

The code encoder adopts the same preprocessing approach of data as the code-diff encoder to obtain the word embedding, character embedding, rule embedding, and position embedding corresponding to the node sequence $V^{(x)}$. Then, the information passes through the Graph-conv layer, Self-attention layer, Char-gating layer, and Rule-gating layer in turn. The final output of the code encoder is $Y^{(x)}$.

4.3 AST encoder

During code generation, an internal AST is maintained and used to track the generation process. The leftmost non-leaf node (to be expanded) of the current abstract syntax tree is taken as a query to predict the next rule and the predicted rule is used to expand this node. Therefore, it is necessary to encode the information of the abstract syntax tree during code generation to provide a global view of the AST while predicting the next rule. In the AST encoder, the generated rule sequence $[r_1, \cdots, r_p]$ represents the generated abstract syntax tree. The goal of the AST encoder is to calculate the representation $[y_1^{(\text{ast})}; \cdots; y_P^{(\text{ast})}]$, for rule sequence, where $y_i^{(\text{ast})}$ is the representation vector of rule r_i, and P is the pre-defined maximum length of the rule sequence.

4.3.1 *Extraction of the encoding information*

Initial embedding: With the embedding approach, each rule r has an initial vector $\bar{r}$. Therefore, given the rule sequence $[r_1, \cdots, r_R]$, the initial representation vector matrix of the rule sequence is obtained by table lookup, namely $R^{(\text{init})} = [r_1^{(\text{init})}; \cdots; r_P^{(\text{init})}]$.

Character embedding: In the rule representation form as $\alpha \rightarrow \beta_1\beta_2\cdots$, α and β_i are very important to express the semantic information of the rule. For example, when the type of the node to be expanded is Statement, the type of the non-terminal symbol α of the next predicted rule must also be Statement. However, the above approach of encoding the rule will ignore the character information of the rule. To capture character information of rules, ExpTrans adopts the approach of Sun *et at*.[16]. To be specific, given the rule r_i:$\alpha \rightarrow \beta_1\beta_2\cdots$, α and all β_j are words in the standard word set. Through table lookup, the representation vector $\alpha^{(w)}$ or $\beta_j^{(w)}$ is obtained. Similarly to Eq. (2), the fully connected layer is used; $\alpha^{(w)}$ and all $\beta_j^{(w)}$ are taken as the input; the output is recorded as $r_i^{(\text{char})}$. Finally, the fully connected layer is adopted again to calculate the character embedding as follows:

$$r_i^{(\text{rule})} = W^{(\text{rule})}[r_i^{(\text{init})}, r_i^{(\text{char})}, \bar{\alpha}] \tag{16}$$

where $W^{(\text{rule})}$ is the network parameter; the character embedding corresponding to the rule sequence $[r_1, \cdots, r_p]$ is $R^{(\text{rule})} = [r_1^{(\text{rule})}; \cdots; r_P^{(\text{rule})}]$.

Position embedding: Similarly, ExpTrans uses Eq. (3) and Eq. (4) to calculate the position embedding of the rule sequence, $P^{(\text{ast})} = [p_{b,1}^{(\text{ast})}, p_{b,2}^{(\text{ast})}, \cdots]$, where $p_{b,i}^{(\text{ast})}$ is the position embedding of the i-th rule in the b-th block.

4.3.2 *Network structure of the AST encoder*

Self-attention: A Self-attention layer is used to capture the dependency between rules. The structure of this layer is consistent with that in the code-diff encoder. The input of this layer is the sum of the rule representation vector and the position embedding of rules, namely $I^{(\text{ast})} = [r_{b,1} + p_{b,1}^{(\text{ast})}; \cdots; r_{b,P} + p_{b,P}^{(\text{ast})}]$. The output of the layer is

$$R^{(\text{self})} = \mathit{Multihead}(I^{(\text{ast})}, I^{(\text{ast})}, I^{(\text{ast})}) \tag{17}$$

When $b = 1$ (the first block), $r_{b,i}$ represents the initial vector of the rule r_i ($r_i^{(\text{init})}$); in other blocks, $r_{b,i}$ is the output of the previous block.

Rule-gating: The AST encoder uses this layer to capture the character information of rules. Similar to the previous gating layer in the code-diff encoder, a control matrix $C^{(\text{r-self})}$ is first calculated based on $R^{(\text{self})}$. The output of this layer is

$$R^{(\text{gate})} = \mathit{Gate}(R^{(\text{self})^{\mathrm{T}}} C^{(\text{r-self})}, R^{(\text{rule})^{\mathrm{T}} C^{(\text{r-self})}}) \tag{18}$$

Code-attention: The code transformation needs to be conducted by the code to be modified, so the model needs to capture the embedding information of x in the subsequent prediction process of code rules. This layer is used to introduce the information of the code to be modified, $Y^{(x)}$, into the encoding result of the rule sequence. The output of this layer is

$$R^{(x)} = \mathit{Multihead}(Y^{(x)}, R^{(\text{gate})}, R^{(\text{gate})}) \tag{19}$$

Diff-attention: When code x is modified, the modification depends on the given modified instance $x_\Delta \rightarrow y_\Delta$. Therefore, the model also needs to capture the modification information. This layer is used to introduce the information of $x_\Delta \rightarrow y_\Delta$, $Y^{(\text{diff})}$, into the encoding result of the rule sequence. The output of this layer is

$$R^{(\text{diff})} = \mathit{Multihead}(Y^{(\text{diff})}, R^{(x)}, R^{(x)}) \tag{20}$$

Graph-conv: An internal abstract syntax tree can be constructed from the generated code rules; the predicted rules correspond to specific nodes. Therefore, there is a meaningful structural relationship between the rules (nodes). However, representing the generated abstract syntax tree with the rule sequence will lose the structural information between rules. Therefore, a Graph-conv layer is adopted to capture the structural information of the AST to enhance the encoding information of rules.

According to the adjacency between the corresponding nodes of rules, an adjacency matrix M_{p*p} between rules can be obtained. When $M[i][j] = 1$, it means that the rule r_j (its corresponding node) is the parent node of rule r_i (its corresponding node). Based on this, the output of this layer is

$$R^{(\text{ast})} = \mathit{Conv}(R^{(\text{diff})}, M) \tag{21}$$

The final output of the AST encoder is $R^{(\text{ast})}$.

4.4 Decoder

The decoder takes the current non-leaf node to be expanded as its input, namely $Q^{(d)} = [q_1; \cdots; q_R]$, where q_i is the representation vector of each node to be expanded. Since the decoder still follows the multi-block design, q_1 is the syntax type or word information corresponding to the node in the first block; q_i is the output of the previous block for other blocks. The goal of the decoder is to generate a query matrix, $D^{(\text{query})} = [d_1^{(\text{query})}; \cdots; d_R^{(\text{query})}]$, where $d_i^{(\text{query})}$ is the query vector corresponding to the i-th node. The i-th rule will be predicted based on $d_i^{(\text{query})}$. The decoder first uses the Self-attention layer to obtain the dependency between the nodes to be expanded. Then, the data stream will pass through the AST-attention layer, the Code-attention layer, and the Diff-attention layer in turn, which is used to obtain the abstract syntax tree, the code to be modified, and the information of modified instances respectively.

Self-attention: In the above query matrix, each query component q_i represents the information of a node to be expanded in the AST. A self-attention layer is used to capture the dependency betweeng queries (namely the dependency between nodes), and the output of this layer is

$$D^{(\text{self})} = \mathit{Multihead}(Q^{(\text{d})}, Q^{(\text{d})}, Q^{(\text{d})}) \tag{22}$$

AST-attention: This layer is used to enhance the query information with the information of the generated AST ($Y^{(\text{ast})}$). In this layer, $Q = D^{(\text{self})}$, $K = Y^{(\text{ast})}$, $V = Y^{(\text{ast})}$, and the output is

$$D^{(\text{ast})} = \mathit{Multihead}(D^{(\text{self})}, Y^{(\text{ast})}, Y^{(\text{ast})}) \tag{23}$$

Code-attention: This layer is used to capture the information of code x ($Y^{(x)}$) to enhance the query information. In this layer, $Q = D^{(\text{ast})}$, $K = Y^{(x)}$, $V = Y^{(xt)}$, and the output is

$$D^{(x)} = \mathit{Multihead}(D^{(\text{ast})}, Y^{(x)}, Y^{(x)}) \tag{24}$$

Diff-attention: This layer is used to enhance the query information with the information of the modified instance $x_\Delta \rightarrow y_\Delta$ ($Y^{(\text{diff})}$). In this layer, $Q = D^{(x)}$, $K = Y^{(\text{diff})}$, $V = Y^{(\text{diff})}$, and the output is

$$D^{(\text{query})} = \mathit{Multihead}(D^{(x)}, Y^{(\text{diff})}, Y^{(\text{diff})}) \tag{25}$$

Finally, the output of the decoder is $D^{(\text{query})}$.

4.5 Rule prediction

When the next rule is predicted, the prediction range of the model will be based on two types of rules. First, when the data is processed, the standard data rule set $R = \{r_1, r_2, \cdots, r_N\}$ is obtained, which can meet the needs for normal code generation. However, there is a dependency between the definition and the usage of variable names in code. The rule $copy(n)$ is added to capture this dependency, which means that the variable name in the n-th rule of the generated rule is copied. When the i-th rule is predicted, in addition to calculating the probability $p(r_j)$ of rule r_j, the probability of $p(copy(t))$ will be calculated according to the pointer network. Therefore, the final prediction result of the i-th rule needs to be chosen from the rules r_j and $copy(t)$. The gating mechanism shown in Eq. (26) will be adopted in this paper to filter the rules of the two types:

$$p(op) = \begin{cases} g \times p(r_j), & op \text{ is instruction } r_j \\ (1-g) \times p(copy(t)), & op \text{ is } copy(t) \end{cases} \tag{26}$$

where g represents the probability that the current predicted rule belongs to R, namely that the probability of the copy rule is $1 - g$. The final predicted rule is $op = argmax_{op} p(op)$.

The probability $p(r_j)$ of the next rule r_j is calculated according to Eq. (27):

$$p(r_j) = softmax(d_i^{(\text{query})} W)[j] \tag{27}$$

When the i-th rule is predicted, the variable name copied by the copy rule is the variable declared in the previous $i - 1$ rules. Therefore, $p(copy(t))$ will be calculated according to Eq. (28) and Eq. (29):

$$\xi_t = v^{\mathrm{T}} \tanh(W^{(\text{query})} d_i^{(\text{query})} + W^{(\text{rule})} r_t^{(\text{init})}) \tag{28}$$

$$p(copy(t)) = \frac{\exp \xi_t}{\sum_1^{i-1} \xi_j} \tag{29}$$

where $1 \le t \le i-1, r_t^{(\text{init})}$ is the representation vector of the generated t-th rule; $W^{(\text{query})}$ and $W^{(\text{rule})}$ are network parameters.

5 Evaluation

Two experiments are carried out to compare ExpTrans with the existing deep-learning-based and rule-based approaches to verify the effectiveness of our approach.

5.1 Experiment 1

This experiment compares the approach presented given in this paper with the existing deep-learning-based methods to verify the effectiveness of ExpTrans.

5.1.1 *Data set*

The experiment is conducted on the data set of Yin *et al.*[13]. In the work of Yin *et al.*, 54 C# open source projects were collected from GitHub, and then 111,724 C# code modifications were extracted and screened out from the submission history of these software projects, of which 91,372/10,176/10,176 pieces of data were used for training/validation/test, respectively. In this data set, data examples can be expressed as <x_i, y_i>, where x_i is the code before modification, and y_i is the code after modification. The example <x_i, y_i> was transformed into <$x_i, x_i \to y_i, y_i$>, and the preprocessing was conducted following the model input requirements.

Specifically, x_i and y_i were parsed into the form of abstract syntax trees first, and then the node sequences were obtained in a top-to-bottom, left-to-right order, i.e., $[v_1^{(\text{ori})}, \cdots, v_{L^{(\text{ori})}}^{(\text{ori})}]$ and $[v_1^{(\text{mod})}, \cdots, v_{L^{(\text{mod})}}^{(\text{mod})}]$. In addition, according to the abstract syntax tree corresponding to y_i, the rule sequence $[r_1, \cdots, r_R]$ was obtained. The node sequence of the code before modification is taken as the input of the code encoder, with a length of $L^{(\text{ori})}$; the node sequences of the pieces of code before and after modification are combined as the input of the code-diff encoder, with a length of $L^{(\text{ori})} + L^{(\text{mod})}$; the rule sequence is the prediction target, with a length of P. The distribution of the four lengths of the data is shown in Table 1. As shown in the second row of Table 1, 92.2% of the instances have the length $L^{(\text{ori})}$ less than 100; 7.7% of the instances have the length $L^{(\text{ori})}$ between 101 and 200; 0.1% of the instances have the length $L^{(\text{ori})}$ between 201 and 300; the maximum length is 239. In addition, the terms contained in the preprocessed data are counted. A total of 2,931 unique terms are found, of which the maximum character length was 70; the character length of 80.3% of the terms was less than 20.

5.1.2 *Comparison approach*

The work of Yin *et al.* is currently the latest work most relevant to the work of this paper[13]. They represented the modification approach in the instances to guide code transformation. In addition, the authors tried to represent the code and modified instances in several ways, to verify the performance by combining different models.

Table 1 Data length distribution in the first experiment

Data length	≤ 100	101–200	201–300	301–400	>400	Maximum length
Code ($L^{(\text{ori})}$)	102,949 (92.2%)	8,574 (7.7%)	142 (0.1%)	—	—	239
Code difference ($L^{(\text{ori})} + L^{(\text{mod})}$)	55,960 (50.1%)	46,839 (41.9%)	7,070 (6.3%)	1,672 (1.5%)	124 (0.1%)	464
Rule	108,929 (97.5%)	2,736 (2.5%)	—	—	—	185
Character	≤ 5 437 (15.0%)	6–10 836 (28.5%)	11–15 687 (23.4%)	15–20 404 (13.8%)	≥ 21 567 (19.3%)	Maximum 70

5.1.3 *Parameter settings*

In order to find the optimal number of blocks in the Transformer structure, the number of blocks has been ranged over 4, 6, and 8. The experimental results show that when the number of blocks is 6, the performance of the model is optimal (the specific results are shown in Table 3). Therefore, the number of blocks ($N1/N2/N3/N4$) for the code-diff encoder, code encoder, AST encoder, and decoder is set to 6. In addition, as shown in Table 1, because all code lengths are less than 300 and more than 98% of the code difference lengths are less than 300, the maximum input length L of the code-diff encoder and the code encoder is set to 300. At the same time, the rule length of all data instances is less than 200, with the maximum length being 185. Therefore, the maximum rule length P allowed by the model is set to 200. In addition, there are more than 80% of the terms with the number of characters at most 20, where the maximum length is 70. Therefore, the maximum term length L' allowed by the model is set at 20 to ensure that this value can cover most words while avoiding the introduction of too many placeholders that affect the model performance.

5.1.4 *Experimental process*

During the comparison with the work of Yin *et al.* [13], their experimental settings are also adopted in this paper, and the training and validation sets of their data set are used to train our

approach. Then, the test set is used to test our approach, namely whether the approach can correctly transform the code x_i before modification into y_i.

5.1.5 *Evaluation criterion*

Correct rate is used in this paper to quantify the performance of the approach. In a piece of data $< x, x_\Delta \rightarrow y_\Delta, y >$, x is the code before modification; $x_\Delta \rightarrow y_\Delta$ is the modified instance; and y is the modified code; if the prediction result $\bar{y}$ given by the approach is the same as y, then x is modified correctly. The correct rate of the approach is defined in this paper as

$$Acc = \frac{\text{Correctly modified instances}}{\text{All instances}}$$

5.1.6 *Experimental results*

The comparative results with the work of Yin *et al*. [13] are shown in Table 2. Compared with the different models proposed by Yin *et al*. [13], ExpTrans gains an 11.8%–30.8% precision increment. In the work of Yin *et al*. [13], two different sub-modules were used to encode the information of the code to be modified and that of the modified instance. At the same time, they also tried two different ideas to model the information, namely the term-sequence-based approach and the graph-structure-based approach, so they explored the actual effect of different combinations of information modeling approaches. From the results of Yin *et al*. [13], the results of term-sequence-based models (such as Seq2Seq-Seq) are better than those of graph-structure-based models (such as Graph2Tree-Graph). The reason for this result is the particularity of the experimental data. Because the modified instance of the model input contains the modification result of the code to be modified, it is more conducive to the prediction of modification results by term-sequence-based models.

Table 2 Comparative results with the work of Reference [13]

Method	Model	Acc@1 (%)
Work of Reference [13]	Seq2Seq-Bag-of-Edits Encoder	44.05
	Seq2Seq-Seq Edit Encoder	59.63
	Graph2Tree-Bag-of-Edits Encoder	40.66
	Graph2Tree-Seq Edits Encoder	57.49
	Graph2Tree-Graph Edit Encoder	48.05
The approach in this paper	ExpTrans	71.45

According to the results of ExpTrans, if ExpTrans is compared with the graph-structure-based model in the work of Yin *et al*. it gains a 23.4% precision increment. This demonstrates that ExpTrans using graphs to represent modified instances and combined with convolutional neural networks can enhance the ability of the model to capture the structural information of code so that the correct rate of ExpTrans is greatly improved. Compared with the term-sequence-based approach by Yin *et al*., ExpTrans also has an improvement of at least 11.8%. These experimental results show that ExpTrans is effective.

In addition, in ExpTrans, the number of blocks of the Transformer structure, copy rules, and different modules all influence the code transformation. Therefore, based on ExpTrans, this paper changes the architecture or parameter settings of the model to produce different variants, thereby exploring the effectiveness of the above factors during the code transformation made by ExpTrans. Settings and performances of different variations are listed in Table 3.

First of all, for a better setting for the number of Transformer structure layers in the model, this paper draws on the work of Vaswani *et al*.[14]. The number of layers in the model is set to 4, 6, and 8 successively. The specific experimental results are shown in line (A) of Table 3. From the results, when the number of layers is set to 4, the correct rate of the approach is 67.37%,

which is 4.08% lower than that when the number of layers is 6. When the number of layers is set to 8, the correct rate is 64.37%. Given the above results, the number of Transformer structure layers in ExpTrans is finally set to 6.

At the same time, ExpTrans is applied again to the data set of the work of Yin *et al*. to verify the effectiveness of the copy rule. The difference is that the copy rule is canceled when the data set is processed. The specific experimental results are shown in line (B) of Table 3. When the rule set does not contain the copy rule, the correct rate of the model drops to 60.12%, which means that the additional copy rule can improve the ability of the model to capture the long-term dependency between fragments of code and the correct rate of the approach.

Table 3 Performances of different variations on ExpTrans

Line	$N1/N2/N3/N4$	Whether the copy rule is contained in the rule set?	Whether the code-diff encoder is contained	Whether the conv layer is contained in code-diff encoder	Whether the Graph-conv layer is contained in code encoder	Acc@1 (%)
ExpTrans	6	Y	Y	Y	Y	**71.45**
(A)	4	—	—	—	—	67.37
	8	—	—	—	—	64.37
(B)	—	N	—	—	—	60.12
(C)	—	—	N	—	Y	6.36
	—	—	Y	N	Y	68.01
	—	—	Y	Y	N	65.69

Finally, we proposes another three variant models by removing the code-diff encoder and the Graph-conv layer in the code-diff encoder and the code encoder respectively. The results are shown in row (C) in Table 3. When the model remove the code-diff encoder, the accuracy drops to 6.36%. The results illustrate that modifcation instances can improve the performance of code transformation, and it also illustrates the effectiveness of the code-diff encoder in ExpTrans. In addition, when remove the Graph-conv layer in the code-diff encoder and the code encoder, the accuracy drops to 68.01% and 65.69%, respectively. Without Graph-conv layer, the model is actually modeling a linear sequence of tokens when modeling the input information. It will ignore the structural information of the code. These results show that ExpTrans can effectively capture the structural information of the code by using graph convolution and improve the performance of code transformation.

5.2 Experiment 2

This experiment compares ExpTrans with the existing rule-based approaches to verify the effectiveness of our approach.

5.2.1 *Data set*

Because the rule-based approach is designed for the Java language, the code-modification data set in the Java language is collected. Before data collection, the additional features of the Java language in different versions are checked, of which five features are selected, and six possible modification patterns are summarized. The corresponding query sentences and modification patterns of different Java features are shown in Table 4. Based on each feature, the corresponding query statement is constructed, and related commits are searched on GitHub. According to the search results, 10 code modifications with corresponding modification patterns are manually filtered out for each query, namely similar code modifications.

5.2.2 *Comparison approaches*

In this experiment, two rule-based code transformation methods, GenPat and ARES, are adopted for comparison.

(1) GenPat[17]: It is a single-example-based code transformation method and generalizes or limits the attributes of specific nodes on the abstract syntax tree by parsing a given single modified instance into an abstract syntax tree. Moreover, it marks the change process of the nodes of the abstract syntax tree. During the code transformation, GenPat will match codes according to the extracted abstract syntax tree and use the change process of the marked nodes to transform the code.

Table 4 Corresponding query statements and modification patterns of different Java features

Version	Feature	Query statement or keyword	Modification pattern example

Java 5	EnhancedFor	for loop replaced with enhanced for loop	– for (int i = 0; i < *names.length*; ++i) {
			– String *val* = *getAttributeValueString*(*names*[i]);
			+ for (String *name*:*names*) {
			+ String*val* = *getAttributeValueString*(*name*);
			…
			}

		replaced while iterators with for iterators	– Iterator it = *names.iterator*();
			– While(*it.hasNext*()) {
			– String *query* = *it.next*();
			+ for (String *name:names*) {
			+ String *query* = *name*;
			…
			}

	Generic type	replace explicit types with diamond operator	– List ⟨*GroundItem*⟩ *check* = new *ArrayList* ⟨*GroundItem*⟩ (); + List ⟨*GroundItem*⟩ *check* = new *ArrayList*⟨⟩();
	ValueOf	replace new *Double* with *Double.valueOf*	– *retrydelay* = new *Double(childval)*; + *retrydelay* = *Double.valueOf*(*childval*);
	StringBuilder	replaced String co ncatenation with StringBuilder	– String *score* = ""; – *score* += "Love"; + StringBuilder *score* = new *StringBuilder*(); + *score.append*("Love");

Java 7	Try-With-resource	use try-with-resources	– *FileInputStreamfis* = new *FileInputStream*(new *File*(*dir*));
			– try {
			+ try (*FileInputStreamfis* = new *FileInputStream*(new *File*(*dir*))) {
			…
			– *fis.close*();
			}

(2) ARES[10]: It is a multiple-examples-based code transformation method and uses a template to represent the modification pattern in modified instances. The common part of modified instances is retained in the template, and wildcards are used to generalize different parts. During code transformation, ARES will match the code according to the extracted template and use the pattern described by the template to modify the matched code.

5.2.3 *Experimental process*

Experiments are conducted with the six sets of similar code modifications that have been collected. Specifically, a group of similar modification sets $P = \{p_1, \cdots, p_{10}\}$ is given, and code modification is $p_i = \langle x_i, y_i \rangle$, where x_i is the code before modification, and y_i is the code after modification. To train the approach proposed in this paper, we construct the following equation based on the set P:

$$P' = \{p_{i,j} | p_{i,j} = \langle x_i, x_j \rightarrow y_j, y_i \rangle\},$$

where, for $1 \le i, j \le 10$, $p_{i,j}$ means that using the modification pattern of p_j to modify the code x_i; P' is divided as follows:

Training set: $\{p_{i,j}\}, 1 \le i \le 10, 1 \le j \le 8$;

Validation set: $\{p_{i,j}\}, 1 \le i \le 10, j = 9$;

Test set: $\{p_{i,j}\}, 1 \le i \le 10, j = 10$.

In the experiment, three approaches were adopted to modify the instances in the test set. Because ARES is for multiple-examples-based code transformation, we provide ARES with P as a set of similar code changes to extract the required modification pattern.

5.2.4 *Parameter settings*

In this experiment, we use the same parameter setting as in Experiment 1.

5.2.5 *Experimental results*

The output results are divided into four types:

✓ The output result of the representation approach, $\bar{y}$, is the same as the expected result y.

○ The approach cannot extract a unified modification pattern from the given modified instances, resulting in no output.

⊘ The extracted modification pattern cannot be applied to the code to be modified, resulting in no output.

⊠ The output result $\bar{y}$ of the approach is inconsistent with the expected result y.

The comparative results on six groups of data are listed in Table 5. The results show that ExpTrans is significantly better than the other two approaches. ExpTrans has correctly modified instances in all groups. Particularly, it can modify all code instances successfully in the second group of data. We manually checked the instances modified incorrectly by ExpTrans and found they mainly occurred when predicting multiple parameters of the same function. The reason may be that ExpTrans lacks the sequence location information of parameters while predicting the parameters of a function, leading to incorrect results. For example, an expected result is *byteBufferReadCheck*(*in*, *buf*, 11);, but the prediction result of ExpTrans is *byteBufferReadCheck*(*in*, **11**, 11), where **11** is the parameter incorrectly predicted.

Further, we checked the output results of GenPat and ARES to explore the reasons for the unsatisfactory results of the two approaches.

The reason why GenPat fails lies in the following points: (1) It relies on the similarity of the structure and syntactic type between the modified instance code and the code to be modified. For example, due to the different syntactic types, the modification approach in "*return new Double*(0.0);"→"*return new Double.valueOf*(0.0);" cannot be used to modify the code "*val=new Double*(0.0);". However, similar code modifications cannot be guaranteed of a similar structure and syntax features in the data obtained in this experiment. Therefore, the modification pattern extracted from instances by GenPat cannot be adapted to the code to be modified (Type ⊘). (2) GenPat needs to use the definition information of the variables in the code. However, only the modified code is extracted when code modifications are obtained, so the extracted code modifications may not include the definitions of all the variables involved. Because GenPat cannot obtain enough information, an incorrect match occurs when the extracted pattern is matched to the code to be modified, leading to incorrect modifications (Type ⊠).

ARES is a code transformation method based on multiple examples. Therefore, when a group of similar code-modified instances is given, it needs to generalize the different parts of the modified instances so that the modification pattern can fit the given modified instances. However, when the given modified instances have similar modification semantics but different structures, ARES may fail to extract a unified modification pattern. For example, ARES cannot extract a unified modification pattern from Groups 2, 4, and 6, so it is impossible for ARES to modify the

corresponding code (Type ○). In addition, there are some incorrect examples because ARES retains the unique variable names of the modified instances in the extracted pattern. Therefore, when the extracted pattern is adapted to the code to be modified, these variable names are introduced into the modified result, causing unsuccessful modifications (Type ⊠).

Table 5 Comparative results with GenPat and ARES

x	Group 1			Group 2			Group 3		
	GenPat	ARES	ExpTrans	GenPat	ARES	ExpTrans	GenPat	ARES	ExpTrans
x_1	⊠	⊠	✓	⊠	✓	✓	⊠	○	⊠
x_2	⊠	⊠	⊠	⊠	✓	✓	⊠	○	⊠
x_3	⊘	⊠	⊠	⊠	⊠	✓	⊠	○	✓
x_4	⊠	⊠	✓	⊠	✓	✓	✓	○	✓
x_5	⊠	⊠	✓	⊠	⊠	✓	⊠	○	✓
x_6	⊠	⊠	✓	⊠	⊠	✓	⊠	○	✓
x_7	⊠	⊠	✓	⊠	⊠	✓	⊠	○	✓
x_8	⊘	⊠	✓	⊠	✓	✓	⊠	○	⊠
x_9	⊘	⊠	⊠	⊠	⊠	✓	⊠	○	✓
x_{10}	⊠	⊠	✓	⊠	⊠	✓	✓	○	✓
x	Group 4			Group 5			Group 6		
	GenPat	ARES	ExpTrans	GenPat	ARES	ExpTrans	GenPat	ARES	ExpTrans
x_1	⊠	○	✓	⊠	○	✓	✓	⊠	⊠
x_2	⊠	○	⊠	⊠	○	⊠	⊠	⊠	⊠
x_3	⊠	○	⊠	⊠	○	⊠	⊠	⊠	✓
x_4	⊠	○	✓	⊠	○	⊠	⊠	⊠	⊠
x_5	⊠	○	✓	⊠	○	✓	⊠	⊠	⊠
x_6	⊠	○	✓	⊠	○	⊠	✓	⊠	⊠
x_7	⊠	○	✓	⊠	○	⊠	⊠	⊠	⊠
x_8	⊠	○	✓	⊠	○	⊠	⊠	⊠	⊠
x_9	⊠	○	⊠	⊠	○	✓	⊠	⊠	⊠
x_{10}	✓	○	✓	⊠	○	⊠	⊠	⊠	⊠

5.3 Discussion

Application scope of the approach: Based on the approach presented in this paper, the maximum length for the input code x to be modified and the modified instance $x_\Delta \rightarrow y_\Delta$ is preset. When the code length exceeds the preset length, the code content exceeding the preset length will be intercepted. This affects the code modification scenarios which are available for the approach to a certain extent. However, in some similar modification tasks with frequent occurrences, such as API version migration, the modified code is often partial and short. Therefore, the preset maximum length does not seriously restrict the practicability of our approach. Nevertheless, in future work, we still need to try to propose different network models to reduce the impact of modifying the code length on the approach.

Data size: The collection of Java data relies on manual screening of search results, which limits the size and efficiency of data collection and thus the greater potential of ExpTrans. In future work, automatic methods will be used to collect data on a large scale, and the effectiveness of the approach will be improved at a lower labor cost.

6 Conclusion

In this paper, we proposed a deep-learning-based code transformation method. By representing modified instances with a graph structure and combining convolutional network and Transformer structure, we enhanced the ability of the approach to capture code structural information. The experimental results show that compared with the existing deep-learning-based and rule-based approaches, the approach developed in this paper gains a remarkable

improvement. For factors that may affect the effectiveness of the approach, we will propose optimization models and automated data collection methods to reduce the effects of these factors in future work.

References

[1] Hunter A, Eastwood JD. Does state boredom cause failures of attention? Examining the relations between trait boredom, state boredom, and sustained attention. Experimental Brain Research, 2016, 1–10. [doi: 10.1007/s00221-016-4749-7]

[2] Ko AJ, Myers BA. A framework and methodology for studying the causes of software errors in programming systems. Journal of Visual Languages and Computing, 2005, 16(1-2): 41–84. [doi: 10.1016/j.jvlc.2004.08.003]

[3] Barr ET, Brun Y, Devanbu P, Harman M, Sarro F. The plastic surgery hypothesis. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. ACM, 2014. 306–317.

[4] Nguyen HA, Nguyen AT, Nguyen TT, *et al*. A study of repetitiveness of code changes in software evolution. Proc. of the 28th IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE Press, 2013. 180–190.

[5] Nguyen TT, Nguyen HA, Pham NH, *et al*. Recurring bug fixes in object-oriented programs. Proc. of the 32nd ACM/IEEE Int'l Conf. on Software Engineering, Volume 1. ACM, 2010. 315–324.

[6] Ray B, Nagappan M, Bird C, *et al*. The uniqueness of changes: Characteristics and applications. Proc. of the 12th Working Conf. on Mining Software Repositories. IEEE Press, 2015. 34–44.

[7] Meng N, Kim M, McKinley KS. Sydit: Creating and applying a program transformation from an example. Proc. of the 19th ACM SIGSOFT Symp. and the 13th European Conf. on Foundations of Software Engineering. ACM, 2011. 440–443.

[8] Andersen J, Nguyen AC, Lo D, *et al*. Semantic patch inference. Proc. of the 27th IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE, 2012. 382–385.

[9] Meng N, Kim M, McKinley KS. LASE: Locating and applying systematic edits by learning from examples. Proc. of the 2013 Int'l Conf. on Software Engineering. 2013. 502–511.

[10] Dotzler G, Kamp M, Kreutzer P, *et al*. More accurate recommendations for method-level changes. Proc. of the 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017. 798–808.

[11] Rolim R, Soares G, D'Antoni L, *et al*. Learning syntactic program transformations from examples. Proc. of the 39th Int'l Conf. on Software Engineering. IEEE Press, 2017. 404–415.

[12] Tufano M, Pantiuchina J, Watson C, *et al*. On learning meaningful code changes via neural machine translation. Proc. of the 41st Int'l Conf. on Software Engineering. IEEE Press, 2019. 25–36.

[13] Yin P, Neubig G, Allamanis M, Brockschmidt M, Gaunt AL. Learning to represent edits. arXiv: 1810.13337. 2018.

[14] Vaswani A, Shazeer N, Parmar N, *et al*. Attention is all you need. In: Advances in Neural Information Processing Systems. 2017. 5998–6008.

[15] Yin PC, Neubig G. A syntactic neural model for general-purpose code generation. arXiv: 1704.01696. 2017.

[16] Sun Z, Zhu Q, Xiong Y, *et al*. Treegen: A tree-based transformer architecture for code generation. Proc. of the AAAI. 2020.

[17] Jiang J, Ren L, Xiong Y, *et al*. Inferring program transformations from singular examples via big code. Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2019. 255–266.

[18] He K, Zhang X, Ren S, *et al*. Deep residual learning for image recognition. Proc. of the CVPR. 2016. 770–778.

[19] See A, Liu PJ, Manning CD. Get to the point: Summarization with pointer-generator networks. Proc. of the ACL. 2017. 1073–1083.

[20] Falleri JR, Morandat F, Blanc X, *et al*. Fine-grained and accurate source code differencing. Proc. of the 29th ACM/IEEE Int'l Conf. on Automated Software Engineering. ACM, 2014. 313–324.

[21] Dehghani M, Gouws S, Vinyals O, *et al*. Universal transformers. arXiv: 1807.03819. 2018.

Yingkui Cao Ph.D. His research interests include software engineering, software reuse, and program generation.

Yanzhen Zou Ph.D., associate professor, and professional member of CCF. Her research interests include software engineering, software reuse, knowledge graph, and intelligent software development.

Zeyu Sun Ph.D., student member of CCF. His research interests include software engineering, software analysis and testing, and program generation.

Bing Xie Ph.D., professor, Ph.D. supervisor, and senior member of CCF. His research interests include software engineering, formal approaches, software reuse, and intelligent software development.