FIRA: Fine-Grained Graph-Based Code Change Representation for Automated Commit Message Generation

Jinhao Dong
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
School of Computer Science,
Peking University
Beijing, China
dongjinhao@stu.pku.edu.cn

Yiling Lou[*]
Department of Computer Science,
Purdue University
West Lafayette, IN, USA
lou47@purdue.edu

Qihao Zhu, Zeyu Sun, Zhilin Li
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
School of Computer Science,
Peking University
Beijing, China
{zhuqh,szy_,1700012439}@pku.edu.cn

Wenjie Zhang, Dan Hao[*]
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
School of Computer Science,
Peking University
Beijing, China
{zhang_wen_jie,haodan}@pku.edu.cn

ABSTRACT

Commit messages summarize code changes of each commit in natural language, which help developers understand code changes without digging into detailed implementations and play an essential role in comprehending software evolution. To alleviate human efforts in writing commit messages, researchers have proposed various automated techniques to generate commit messages, including template-based, information retrieval-based, and learning-based techniques. Although promising, previous techniques have limited effectiveness due to their *coarse-grained code change representations*.

This work proposes a novel commit message generation technique, FIRA, which first represents code changes via fine-grained graphs and then learns to generate commit messages automatically. Different from previous techniques, FIRA represents the code changes with fine-grained graphs, which explicitly describe the code edit operations between the old version and the new version, and code tokens at different granularities (i.e., sub-tokens and integral tokens). Based on the graph-based representation, FIRA generates commit messages by a generation model, which includes a graph-neural-network-based encoder and a transformer-based decoder. To make both sub-tokens and integral tokens as available ingredients for commit message generation, the decoder is further incorporated with a novel dual copy mechanism. We further perform an extensive study to evaluate the effectiveness of FIRA. Our quantitative results show that FIRA outperforms state-of-the-art techniques in terms of BLEU, ROUGE-L, and METEOR; and our ablation analysis further shows that major components in our technique both positively contribute to the effectiveness of FIRA. In addition, we further perform a human study to evaluate the quality of generated commit messages from the perspective of developers, and the results consistently show the effectiveness of FIRA over the compared techniques.

[*]Corresponding authors

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Commit Message Generation, Graph Neural Network, Code Change Representation

ACM Reference Format:
Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, and Wenjie Zhang, Dan Hao. 2022. FIRA: Fine-Grained Graph-Based Code Change Representation for Automated Commit Message Generation. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510069

1 INTRODUCTION

When developers commit changed code to a version control system, each commit is supposed to be documented with a *commit message*. Commit messages summarize code changes in natural language, and can help developers quickly understand the high-level intention of code changes without digging into detailed implementations. Therefore, commit messages are prevalent in software maintenance and play an essential role in comprehending software evolution [4].

However, manually writing commit messages can be very labor-intensive. High-quality commit messages should precisely describe the rationales of changed code, which often requires non-trivial manual efforts in practice. In addition, modern software has been evolving rapidly, and frequent commit submissions put a heavy burden on developers. Therefore, although commit messages are beneficial, they are often neglected by developers due to the time costs. As is reported, almost 14% commit messages in 23K Java projects are empty [10].

To alleviate manual efforts in writing commit messages, researchers have proposed various techniques to generate commit messages automatically. Given a code change, these techniques first represent the old-version and new-version code with specific formats, such as sequences of tokens or paths of abstract syntax tree (AST), and then generate commit messages based on the representations via different strategies such as template-based [4, 6], information retrieval-based [16, 29], and learning-based [19, 28, 32, 41, 44] techniques.

Although achieving promising performance, the effectiveness of previous techniques has been restricted by their *coarse-grained code change representations. First, existing commit message generation techniques represent the code changes by simply putting old-version and new-version code together without explicitly highlighting fine-grained edit operations.* For example, given an expression "`a = 1;`" modified into "`b = 1;`", token-based representations [19, 32, 41, 44] represent such code changes by concatenating both expressions into one flat sequence of tokens (i.e., "`- a = 1; + b = 1;`"), where -/+ denotes the old/new-version code; the AST-based representations [28] represent the code changes by concatenating old-version and new-version AST paths into one sequence (i.e., "`- assignment.variable.a.operator.=.literal.1; + assignment.variable.b.operator.=.literal.1;`"). Therefore, existing learning-based models have to compare the code representations of old and new versions so as to capture the subtle edit operation (i.e., the token "`a`" is changed into a new token "`b`") by themselves, which makes it more challenging to generate precise commit messages. *Second, existing code change representations mainly focus on coarse-grained tokens (i.e., integral tokens) in the code without explicitly and individually describing finer-grained tokens (i.e., sub-tokens of integral tokens).* In fact, it is prevalent that the commit messages may contain sub-tokens of the input code changes. For example, for a code change that contains an integral token "`setMinimumSize`", its relevant commit message contains three tokens "set", "minimum", and "size", which are exactly the sub-tokens of the integral token "`setMinimumSize`". However, most previous techniques [19, 32, 41] consider only integral tokens and ignore sub-tokens in their code change representations; while a few techniques [28, 44] represent all sub-tokens in a compound representation (e.g., one single embedding vector) without representing each sub-token individually. Such compound representations make it challenging to utilize each sub-token as available ingredients for commit message generation. Therefore, they exhibit a poor performance for the cases that commit messages contain sub-tokens of the input code.

To address the limitations above, in this work, we propose a novel commit message generation technique, **FIRA**, which first represents code changes via **fi**ne-grained g**ra**phs and then learns to generate commit messages automatically. Compared to previous code change representations, FIRA makes the first attempt to explicitly describe the edit operations between the old-version and new-version code, along with tokens at different granularities (i.e., integral tokens and sub-tokens). Based on the proposed graph-based representations, FIRA then learns to generate commit messages iteratively with an encoder-decoder model. In particular, FIRA incorporates the graph neural network in the encoder so as to directly encode the graph-structured inputs; and the decoder incorporates the transformer [39] and a novel dual copy mechanism, which can not only generate tokens from the vocabulary but also copy both integral tokens and sub-tokens from the input.

We perform an extensive evaluation to compare FIRA with six state-of-the-art commit message techniques on a widely-used benchmark [16, 19, 29, 32, 41, 44]. The results show that FIRA outperforms all compared techniques in terms of BLEU, ROUGE-L, and METEOR. We further analyze the effectiveness of each component in FIRA by an ablation study and case analysis. The results further confirm that major components (i.e., explicitly representing edit operations and copying sub-tokens) both positively contribute to the effectiveness of FIRA, and indeed help generate higher-quality commit messages than previous techniques. In addition, we further perform a human study to evaluate the quality of generated commit messages from the perspective of developers, which consistently shows the effectiveness of FIRA over compared techniques.

In summary, this paper makes the following contributions:

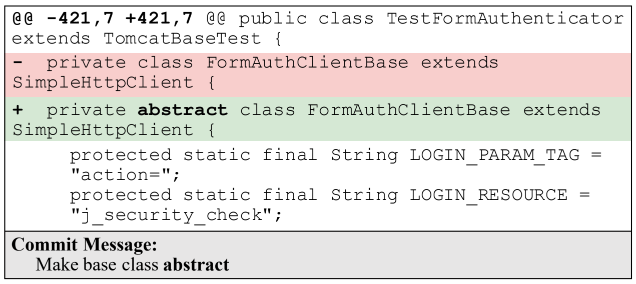
- **A fine-grained graph-based code change representation** for commit message generation, which explicitly describes code edit operations and tokens at different granularities.
- **A novel encoder-decoder model** for commit message generation, which leverages the graph neural network in the encoder to process the proposed graph-based representation, and leverages the transformer with a novel dual copy mechanism in the decoder to utilize both integral tokens and sub-tokens.
- **An extensive experiment** evaluating our approach against six state-of-the-art techniques on a widely-used benchmark, which suggests the effectiveness of our approach by the quantitative, qualitative, and ablation analysis.
- **A human study** on the quality of generated commit messages, which further shows the effectiveness of our approach from the perspective of developers.
- **A replication package** available at https://github.com/DJjjjhao/FIRA-ICSE.

2 MOTIVATION

Existing commit message generation techniques represent and utilize code changes in a coarse-grained way. First, they simply put old-version and new-version code together without explicitly describing fine-grained edit operations; second, they only focus on coarse-grained tokens (i.e., integral tokens) without representing sub-tokens individually. In this section, we further illustrate these limitations with several real-world examples.

2.1 Limitation 1: Edit Operations

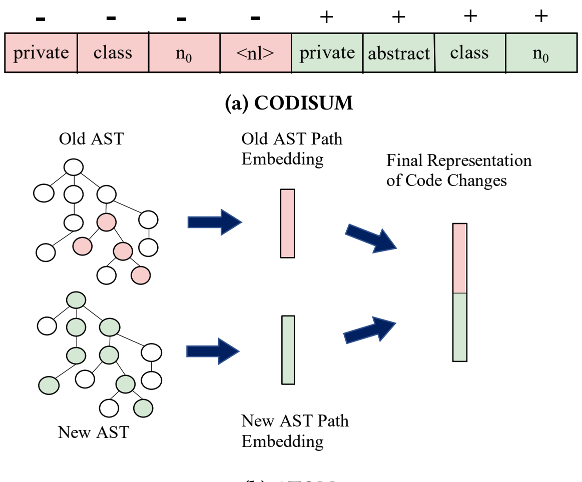
As shown by the example in Figure 1, we can observe that edit operations are highly relevant to developer-written commit messages. The developer makes the edit operation (i.e., adding one token "abstract") and the corresponding commit message indicates his/her intention of making the class abstract. However, existing techniques cannot always notice such edit operations, since their



```
@@ -421,7 +421,7 @@ public class TestFormAuthenticator
extends TomcatBaseTest {
-  private class FormAuthClientBase extends
SimpleHttpClient {
+  private abstract class FormAuthClientBase extends
SimpleHttpClient {
       protected static final String LOGIN_PARAM_TAG =
       "action=";
       protected static final String LOGIN_RESOURCE =
       "j_security_check";
```

Commit Message:
Make base class **abstract**

Figure 1: Motivating example: edit operations



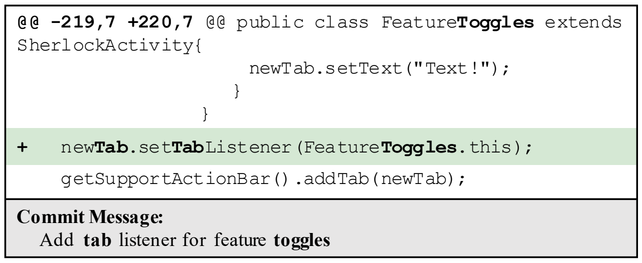
(a) CODISUM

(b) ATOM

Figure 2: Existing code change representations

code change representations put old-version and new-version code together without explicitly highlighting the differences. Therefore, they have to compare old-version and new-version code and then infer the edit operations by themselves. For example, as shown in Figure 2a, CODISUM [44], one of the learning-based approaches, represents the code change by concatenating the flat token sequences of old-version and new-version code into one sequence. Although each token in the sequence is annotated with old version (-) or new version (+), CODISUM has to learn to infer the specific edit operation (i.e., one token "abstract" is added) by itself. Similarly, in Figure 2b, another learning-based technique ATOM [28], represents the code changes by concatenating old-version and new-version AST paths together, and it also has to compare two paths by itself so as to capture the edit operations at AST level. Such coarse-grained code change representations are actually cumbersome, especially when code changes involve very minor edit operations with the majority of tokens unchanged (e.g., only one token is changed in the example). Learning-based techniques cannot always guarantee to precisely capture such subtle edit operations, which may further result in imprecise commit message generation. In fact, our experimental results also confirm that these techniques all fail to generate precise commit messages for this example.

To address this limitation, we propose to explicitly highlight edit operations in the code change representation, which can include more accurate information for commit message generation.



```
@@ -219,7 +220,7 @@ public class FeatureToggles extends
SherlockActivity{
                    newTab.setText("Text!");
                 }
             }
+    newTab.setTabListener(FeatureToggles.this);
     getSupportActionBar().addTab(newTab);
```

Commit Message:
Add **tab** listener for feature **toggles**

Figure 3: Motivating example: sub-tokens

2.2 Limitation 2: Sub-tokens

As shown by the example in Figure 3, we can observe that sub-tokens in the input code can provide very helpful hints for commit message generation. For example, the developer-written commit message "Add tab listener for feature toggles" consists of the sub-tokens "*tab*", "*listener*", "*feature*" and "*toggles*" in the input code. However, existing techniques focus on integral tokens and seldom treat sub-token as equally important as integral token. For example, most existing techniques [19, 32, 41] ignore sub-tokens in their code change representations, while a few techniques [28, 44] describe all sub-tokens in a compound representation (e.g., one single embedding vector) without representing sub-tokens explicitly and individually. Such compound representations restrict the utilization of sub-tokens. For example, with such representations, existing techniques can only generate the frequent sub-tokens that are included in the vocabulary, but often fail to generate those infrequent sub-tokens that are excluded in the vocabulary or seldom occur in the training set. Actually, for the generation tasks related to program code, such infrequent sub-tokens can be very prevalent since they are often project specific tokens (e.g., "*tab*" and "*toggles*" in the example). Therefore, existing coarse-grained code change representations make it challenging to generate commit messages containing such sub-tokens.

Therefore, to fully utilize sub-tokens in the code, we propose to treat all integral tokens and sub-tokens equally important and represent sub-tokens individually in the code change representation. In addition, to make both frequent and infrequent sub-tokens as ingredients of the commit message, we further leverage a novel dual copy mechanism additionally for sub-tokens in our model so that both integral tokens and sub-tokens can be either copied or generated from the vocabulary.

3 CODE CHANGE REPRESENTATION

This section presents our fine-grained graph-based code change representation, which explicitly includes edit operations and sub-tokens to enable more precise commit message generation. In particular, the graph construction consists of four steps, including (1) building chopped abstract syntax trees (Section 3.1), (2) adding sub-tokens (Section 3.2), (3) annotating edit operations (Section 3.3), and (4) incorporating additional sequential information (Section 3.4). We then introduce each step in detail and use the motivating example in Figure 1 for illustration.

3.1 Chopped Abstract Syntax Trees

A typical code change often includes the modified code and its surrounding context. For example, a code change in GitHub (e.g.,

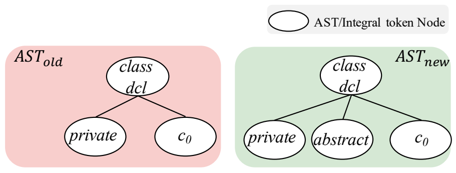


Figure 4: Graph$_{AST}$: chopped AST

Figure 1 and Figure 3) often contains multiple lines of code and each line starts with its change type. More specifically, "-" denotes the deleted old-version code, "+" denotes the added new-version code, and the empty character " " denotes the unchanged code both in old and new version. A code change may involve single or multiple hunks. Here, a *hunk* refers to the continuous lines with the same change type. Among commits in GitHub, some may modify only one token in a hunk, while some may modify hundreds of lines in multiple hunks. Existing learning-based techniques often represent all the changed code together, e.g., CODISUM [44] concatenates all deleted/added hunks as one sequence and ATOM [28] constructs AST of the entire code file even if the code changes occur in only several lines in the file. Representing the code change as a whole can obfuscate details among hunks and thus makes it more challenging for the commit message generation model to summarize essential features from such a coarse-grained representation. Therefore, in FIRA, we propose to construct AST at hunk level, so that more detailed information can be reserved when the code change involves multiple hunks.

More specifically, given a code change, we first separate it into several hunks according to their change types; then for each hunk, we parse it to construct its own abstract syntax tree, i.e., chopped AST. In this way, we obtain a set of chopped ASTs for the given code change, which are actually a set of graphs with basic semantic and syntactic information of each hunk. In particular, for the chopped AST of a deleted hunk (i.e., all its lines are deleted), we denote it as AST$_{old}$ since the relevant code only exists in the old version; for the chopped AST of an added hunk (i.e., all its lines are added), we denote it as AST$_{new}$ since the relevant code only exists in the new version. We denote the graph constructed in this phase as Graph$_{AST}$. Figure 4 presents the Graph$_{AST}$ of changed lines in the illustration example, where "*class dcl*" is the abbreviation of the AST node type "*class_declaration*", and "c_0" is the placeholder of the class name "*FormAuthClientBase*".

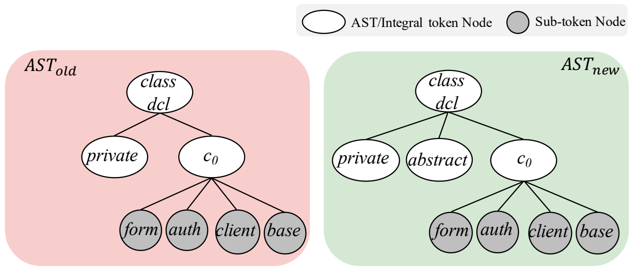


Figure 5: Graph$_{token}$: Graph$_{AST}$ extended with sub-tokens

3.2 Sub-tokens

As mentioned in Section 2, commit messages often contain coarse/fine-grained tokens (i.e., integral tokens and sub-tokens) in the input code. Such a phenomenon is prevalent, since it is a common practice for developers to name a function or a class with phrases. For example, given a method named as "deleteOldThreadDumps", this integral token consists of four sub-tokens "delete old thread dumps", which describe the functionality of the method and might be adopted in the commit message when code changes are relevant to this method. Therefore, in our representation, we consider not only the integral tokens but also their sub-tokens. More specifically, in each Graph$_{AST}$, for the node with an integral token, we split it into separated sub-tokens according to the widely-adopted naming convention (i.e., camel case and snake case), represent these sub-tokens as extra nodes in the graph, and then connect them with their belonging integral token nodes. In this way, the chopped AST is extended with nodes and edges relevant to sub-tokens, where integral tokens and sub-tokens are equally-important individuals and both can be directly utilized in the subsequent commit message generation. We denote the graph constructed in this phase as Graph$_{token}$. Figure 5 presents the Graph$_{token}$, which extends the Graph$_{AST}$ (i.e., Figure 4) with sub-token information.

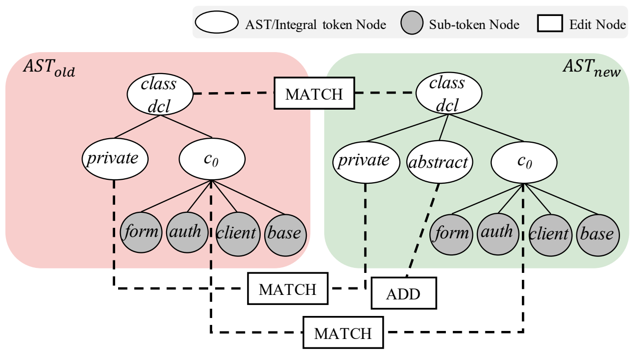


Figure 6: Graph$_{edition}$: Graph$_{token}$ extended with edit nodes

3.3 Edit Operations

So far, all nodes in the graph represent code elements, which are denoted as *code nodes* for distinction. Based on Graph$_{token}$, we further introduce *edit nodes* to explicitly represent fine-grained edit operations between AST$_{old}$ and AST$_{new}$. In particular, we consider five edit nodes, including v$_{ADD}$, v$_{DEL}$, v$_{MOVE}$, v$_{UPDATE}$, and v$_{MATCH}$.

- v$_{ADD}$. If the code node v exists in AST$_{new}$ but not in AST$_{old}$, v is newly-added and should be connected with an edit node v$_{ADD}$.
- v$_{DEL}$. If the code node v exists in AST$_{old}$ but not in AST$_{new}$, v is deleted and should be connected with an edit node v$_{DEL}$.
- v$_{MOVE}$. If the code node v exists in both AST$_{old}$ and AST$_{new}$ and the positions of v and its sub-tree are moved, the node v in both AST$_{old}$ and AST$_{new}$ should be connected with an edit node v$_{MOVE}$.
- v$_{UPDATE}$. If the code node v exists in both AST$_{old}$ and AST$_{new}$ and its value is updated, the nodes v in both AST$_{old}$ and AST$_{new}$ should be connected with an edit node v$_{UPDATE}$.
- v$_{MATCH}$. If a node v exists in both AST$_{old}$ and AST$_{new}$ and its value and position remain unchanged, the nodes v in both

AST$_{old}$ and AST$_{new}$ should be connected with an edit node v$_{MATCH}$.

According to the description above, we further insert edit nodes to the Graph$_{token}$ by connecting them with the original code nodes. We denote the graph constructed in this phase as Graph$_{edition}$. Figure 6 presents the Graph$_{edition}$, which further extends the Graph$_{token}$ (i.e., Figure 5) with edit nodes.

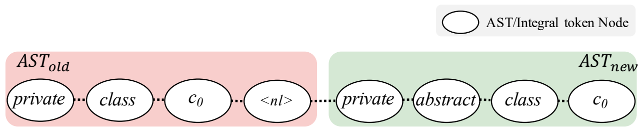


Figure 7: Graph$_{seq}$: one-line graph

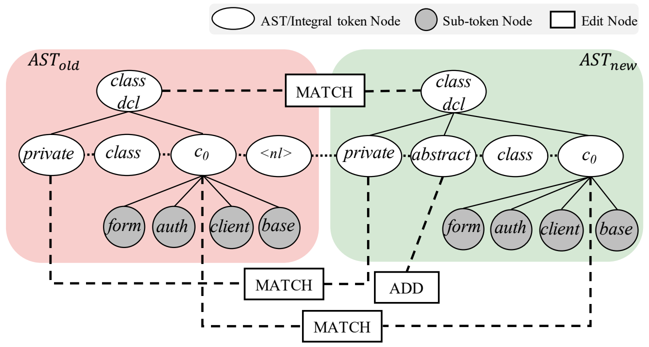


Figure 8: Graph$_{final}$: fine-grained representation for code changes

3.4 Additional Sequential Information

So far, the code change has mainly been represented based on the AST structure. As suggested by previous work [19, 44, 45], the sequential information (i.e., treating code as a flat token sequence) can also be helpful for commit message generation, since it can reserve the adjacent relationship and the order of the tokens. Therefore, we further extend Graph$_{edition}$ with additional sequential information so as to include more helpful information. In particular, we first build an extra one-line graph (denoted as Graph$_{seq}$) by regarding each token as a node and connecting every two adjacent nodes. Figure 7 presents the Graph$_{seq}$ of the code change hank. Then, we merge Graph$_{seq}$ with the Graph$_{edition}$ by using the nodes existing in both graphs as the anchor nodes. More specifically, for each node v_i in Graph$_{edition}$, if v_i matches with the node v_j in Graph$_{seq}$, v_i is removed from Graph$_{edition}$ and all its connected edges are re-connected with v_j. In this way, we combine the AST-based information (i.e., Graph$_{edition}$) and sequential information (i.e., Graph$_{seq}$) into one larger graph, i.e., as Graph$_{final}$. Figure 8 presents the Graph$_{final}$ for the illustration example, which further extends the Graph$_{edition}$ (i.e., Figure 6) with additional sequential information.

4 MODEL ARCHITECTURE

Figure 9 presents the overview of our model, whose input is the final graph-based code change representation and output is the generated commit message. Overall, the model is in an encoder-decoder architecture. In the encoder, we adopt graph neural networks (GNN) due to its strong capability of processing graph-structured data [12, 13, 22, 30, 34, 40, 43]. In the decoder, we leverage the transformer architecture [39], which is the state-of-the-art sequence to sequence model and is widely used in various generation tasks [1, 37, 46], to generate tokens in commit messages iteratively. When generating the next token, the decoder first performs self-attention between the current token and the previously generated tokens, and then performs cross-attention over input tokens embedded by the encoder. In addition, to fully utilize the integral tokens and sub-tokens in inputs, the decoder further incorporates a novel dual copy mechanism, which can copy both integral tokens and sub-tokens from the inputs. In other words, in each iteration, the model can choose an integral token or a sub-token with highest probability from the vocabulary or directly from the inputs. We then describe each component in detail.

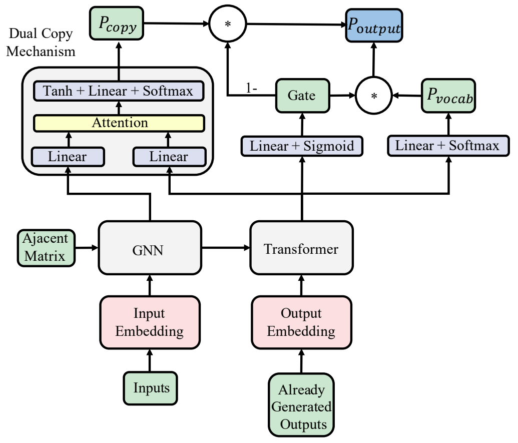


Figure 9: Architecture of the proposed model

4.1 Encoder

Given the final graph-based representation of code changes, i.e., Graph$_{final}$, the encoder first embeds the nodes with an embedding layer (i.e., in Section 4.1.1); then the graphs are represented by embedding vectors and an adjacency matrix, which can be further fed to a graph neural network layer (i.e., in Section 4.1.2); the final output of the encoder is learned representation vectors for each node, which can be further used by the decoder.

4.1.1 Embedding Layer. Formally, the final graph-based representation of code changes Graph$_{final}$ can be defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the nodes and $\mathcal{E}$ denotes the edges in the graph. As mentioned above, $\mathcal{V}$ contains two types of nodes, i.e., code nodes and edit nodes. We establish a lookup table for both nodes and covert them to embedding vectors based on the table. In particular, the embedding vectors of code nodes can be denoted as $[\boldsymbol{c}_1, \boldsymbol{c}_2, ..., \boldsymbol{c}_{N_c}]$ and the embedding vectors of edit nodes can be denoted as $[\boldsymbol{e}_1, \boldsymbol{e}_2, ..., \boldsymbol{e}_{N_e}]$, where N_c and N_e denote the numbers of code nodes and edit nodes, respectively. Therefore, the embedding vectors E for the graph can be represented by $[\boldsymbol{c}_1, \boldsymbol{c}_2, ..., \boldsymbol{c}_{N_c}, \boldsymbol{e}_1, \boldsymbol{e}_2, ...$

$, \boldsymbol{e}_{N_e}]$, where $E \in \mathbb{R}^{d_x \times N}$, $N = N_c + N_e$, and d_x denotes the embedding size. Note that embedding vectors are learnable and are initialized randomly.

4.1.2 *Graph Neural Network.*

The embedding vectors are further fed into a graph convolution network (GCN) layer. GCN [22] is a variant of the graph neural network (GNN) and it leverages the first-order approximation of Chebyshev Spectral CNN (ChebNet) [7] to aggregate the feature information among all neighbor nodes [43].

Here we use an adjacency matrix and the embedding vectors E to identically represent the input graph $\mathcal{G}$, so that GCN can directly process the input. For an adjacency matrix A of $\mathcal{G}$ ($A \in \{0,1\}^{N \times N}$), $A_{i,j}$ means whether there exists an edge between the node v_i and the node v_j in $\mathcal{G}$. In order to preserve the information of each node itself, we further include self-connections to each node in the graph and obtain an enhanced adjacency matrix $\widetilde{A}$ with self-connections. In addition, to avoid gradient explosion caused by accumulated degrees, we apply symmetric normalization to $\widetilde{A}$ and get the normalized adjacency matrix $\hat{A}$, as shown in Equation 1. $\widetilde{D}$ denotes the degree matrix of $\widetilde{A}$, which can be computed by $\widetilde{D}_{ii} = \sum_j \widetilde{A}_{ij}$.

$$\hat{A} = \widetilde{D}^{-\frac{1}{2}} \widetilde{A} \widetilde{D}^{-\frac{1}{2}}. \tag{1}$$

The output of GCN in the lth iteration can be computed as Equation 2. $W_g \in \mathbb{R}^{d_x \times d_x}$ is the trainable parameters. X^{l-1} denotes the embedding vectors of the nodes in the last iteration, and initially X^0 is the embedding vectors of all nodes (i.e., E). In addition, to boost the learning process, we employ residual connection [15] and layer normalization [2] similar to the transformer architecture [39].

$$X^l = W_g X^{l-1} \hat{A}, \tag{2}$$

After L iterations, the final representation of the nodes can be denoted as X^L, i.e., $X^L \in \mathbb{R}^{d_x \times N}$.

4.2 Decoder

The decoder is built on top of a transformer architecture with a novel dual copy mechanism for both integral tokens and sub-tokens.

4.2.1 *Transformer Layer.*

Here we use the decoder part of transformer [39], which is stacked by multi-head self-attention, multi-head attention over the output of the encoder, and a fully-connected feed-forward network.

For better illustration, we denote the output of the encoder as X_e, i.e., $X_e = X^L$. The decoder decides each token in the commit message iteratively, which is based on both the output of the encoder X_e and the currently generated tokens. When generating the kth token in the commit message, we denote the output of the decoder as $\boldsymbol{x}_d^k$, (i.e., $\boldsymbol{x}_d^k \in \mathbb{R}^{d_x}$), which can be computed as Equation 3. For better illustration, we use X_d^{k-1} to represent the already generated output $[\boldsymbol{x}_d^1, \boldsymbol{x}_d^2, ..., \boldsymbol{x}_d^{k-1}]$ of the decoder.

$$\boldsymbol{x}_d^k = \text{Transformer}(X_e, X_d^{k-1}) \tag{3}$$

Next, we introduce the detailed computation process of transformer. First, transformer computes multi-head self-attention (i.e., $\boldsymbol{a}_d^k$). $\boldsymbol{a}_d^k$ is the concatenation of multiple single attention $\boldsymbol{a}_d^k(i)$, which is the weighted sum of the already generated output X_d^{k-1}, as shown in Equation 4 and Equation 5. $W_Q(i) \in \mathbb{R}^{d_x \times d_x}$, $W_K(i) \in \mathbb{R}^{d_x \times d_x}$, $W_V(i) \in \mathbb{R}^{d_x \times d_x}$, $W_O \in \mathbb{R}^{d_x \times hd_x}$ denote the projection parameters, and h is the number of heads.

$$\boldsymbol{a}_d^k(i) = W_V(i) X_d^{k-1} \cdot \text{softmax}\left(\frac{\left(X_d^{k-1}\right)^T W_K(i)^T \cdot W_Q(i) \boldsymbol{x}_d^{k-1}}{\sqrt{d_x}} \right) \tag{4}$$

$$\boldsymbol{a}_d^k = W_O[\boldsymbol{a}_d^k(1); \boldsymbol{a}_d^k(2); ...; \boldsymbol{a}_d^k(h)] \tag{5}$$

Second, transformer computes multi-head attention between a_d^k and the output of the encoder X_e, which is denoted as a_e^k and shown in Equation 6 and Equation 7.

$$\boldsymbol{a}_e^k(i) = W_V(i) X_e \cdot \text{softmax}\left(\frac{X_e{}^T W_K(i)^T \cdot W_Q(i) \boldsymbol{a}_d^k}{\sqrt{d_x}} \right) \tag{6}$$

$$\boldsymbol{a}_e^k = W_O[\boldsymbol{a}_e^k(1); \boldsymbol{a}_e^k(2); ...; \boldsymbol{a}_e^k(h)] \tag{7}$$

Third, a_e^k passes a fully connected feed-forward network to get the output $\boldsymbol{x}_d^k$, as shown in Equation 8. $W_1 \in \mathbb{R}^{d_x \times d_x}$, $W_2 \in \mathbb{R}^{d_x \times d_x}$, $b_1 \in \mathbb{R}^{d_x}$, $b_2 \in \mathbb{R}^{d_x}$ are trainable parameters.

$$\boldsymbol{x}_d^k = W_2 \cdot \max(0, W_1 \boldsymbol{a}_e^k + b_1) + b_2 \tag{8}$$

$\boldsymbol{x}_d^k$ is then fed to a linear layer and transformed into a $|V|$-dimension vector $\boldsymbol{o}_v^k$ as shown in Equation 9. $|V|$ denotes the size of the vocabulary and $W_v \in \mathbb{R}^{|V| \times d_x}$ is a trainable parameter.

$$\boldsymbol{o}_v^k = W_v \boldsymbol{x}_d^k \tag{9}$$

At last, for each token in the vocabulary, the decoder calculates its probability of being selected as the next token by passing $\boldsymbol{o}_v^k$ to a *softmax* layer. $\boldsymbol{p}_v^k$ denotes the probability distribution across the vocabulary, and $\boldsymbol{p}_v^k(i)$ denotes the probability of the ith token being selected, which is computed as Equation 10.

$$\boldsymbol{p}_v^k(i) = \frac{exp\{\boldsymbol{o}_v^k(i)\}}{\sum_{j=1}^{|V|} exp\{\boldsymbol{o}_v^k(j)\}} \tag{10}$$

4.2.2 *Dual Copy Mechanism.*

To fully utilize both integral tokens and sub-tokens during commit message generation, we propose and include a novel dual copy mechanism in the decoder. In this way, when generating each token in the commit message, the candidate tokens can be selected not only from the vocabulary but also from the integral tokens or sub-tokens in the input.

More specifically, in the kth iteration, the probability of each input token being copied is computed according to the current output of the decoder (i.e., $\boldsymbol{x}_d^k$). In FIRA, we consider the input token which is the most similar to $\boldsymbol{x}_d^k$ with the highest probability of being copied. Given an input token (i.e., a code node v_j in $\mathcal{G}$), its similarity to $\boldsymbol{x}_d^k$ can be computed by the sum of its embedding vector $\boldsymbol{x}_e^j$ and the output of the decoder $\boldsymbol{x}_d^k$, as shown in Equation 11. $W_1 \in \mathbb{R}^{d_x \times d_x}$, $W_2 \in \mathbb{R}^{d_x \times d_x}$, $\boldsymbol{v} \in \mathbb{R}^{d_x}$ are learnable parameters.

$$s_k(j) = \boldsymbol{v}^T \tanh(W_1 \boldsymbol{x}_d^k + W_2 \boldsymbol{x}_e^j) \tag{11}$$

The similarity of kth token $\boldsymbol{s_k}$ is further fed to a *softmax* layer, which generates the probability of each input token being copied, i.e., $\boldsymbol{p}_c^k = softmax(\boldsymbol{s}_k)$.

At the end of the iteration, we combine the probability distribution across the vocabulary tokens (i.e., $\boldsymbol{p}_v^k$) and the probability distribution across input tokens (i.e., $\boldsymbol{p}_c^k$) as Equation 13. g are learned according to the output of the decoder, as shown in Equation 12. $\boldsymbol{w} \in \mathbb{R}^{1\times d_x}$ is the learnable parameter. In this way, the kth token to be selected in the commit message would be a token from the vocabulary or copied from inputs.

$$g = \frac{1}{1 + exp\{\boldsymbol{w}\boldsymbol{x}_d^k\}} \tag{12}$$

$$\boldsymbol{p}^k = [g * \boldsymbol{p}_v^k; (1-g) * \boldsymbol{p}_c^k] \tag{13}$$

5 EXPERIMENTAL SETUP

5.1 Research Question

- **RQ1: Overall effectiveness.** How does FIRA perform compared to the state-of-the-art commit message generation techniques?
- **RQ2: Ablation analysis.** How does each component of FIRA contribute to the effectiveness?
- **RQ3: Human evaluation.** How does FIRA perform from the perspective of developers?

5.2 Dataset

Our experiments are evaluated on the well-established benchmark [20, 44], which has been widely used in previous commit message generation techniques [16, 19, 29, 32, 41, 44]. The dataset is based on the commits from top 1,000 popular Java projects in GitHub, excluding rollback/merge commits and duplicated code changes. For each commit, it includes the first sentence of the relevant commit message. In total, the dataset contains 90,661 pairs of commits and the relevant commit messages. Following existing work [44], we randomly select 75,000 commits as the training set, 8,000 commits as the validation set, and the remaining 7,661 commits as the testing set.

5.3 Compared Techniques

We compare FIRA with six state-of-the-art commit message generation techniques as follows.

Information retrieval-based techniques leverage information retrieval (IR) to adopt existing commit messages from similar code changes. We consider two representative IR-based techniques NNGen [29] and LogGen [16] for comparison.

Learning-based techniques leverage neural machine translation (NMT) models to generate commit messages automatically. We consider four state-of-the-art learning-based techniques, i.e., CODISUM [44], ATOM [28], CoreGen [32], and CoRec [41] for comparison.

5.4 Implementation

Representations. FIRA applies GumTree [11] to map ASTs of old-version and new-version code and then to identify edit operations. GumTree [11] is a representative AST mapping algorithm and has been widely adopted in various tasks [5, 14, 18, 24, 26, 31].

Model. In the encoder, we set the size of the input graphs (i.e., the maximum number of nodes) up to 650, containing up to 370 code nodes and 280 edit nodes, which is more than the number of the graph nodes of each training data so that the largest graphs in the training set can be included. In the decoder, we set the maximum length of each commit message as 30, which is longer than the length of all commit messages in the training set. For the hyper-parameters, we configure the six-layer GNN with 0.20 dropout rate [36], and the six-layer eight-head transformer with 0.10 dropout rate and 256-dimension hidden states. In the training phase, we adopt the cross-entropy loss function and the Adam optimizer [21] with 0.0001 learning rate. We tune these hyper-parameters and select the best performing model in the validation set.

Compared techniques. We directly reuse the implementations of the compared techniques from their reproducible packages, if their packages are available and executable [28, 29, 32, 41]; otherwise, we re-implement the techniques strictly following the description in their papers.

Environment. The experiments are performed on a Dell workstation with Intel Xeon CPU E5-2680 v4 @ 2.40GHz, running Ubuntu 16.04.6 LTS. The models are trained on two 24G GPUs of GeForce RTX 3090 and two 24G GPUs of NVIDIA TITAN RTX.

5.5 Evaluation Metrics

We use the commit messages (i.e., manually written by developers) in the dataset as the ground truth. In particular, given a code change, we compare the similarity between the generated commit message with the ground truth. Following previous work on commit message generation [16, 19, 28, 29, 32, 41, 44], we use the widely-used metrics, BLEU, ROUGE-L, and METEOR to measure the similarity. Their computation details are presented as follows.

BLEU measures the precision of generated sequences by calculating its average of the modified n-gram precision (i.e., 1-gram, 2-grams, 3-grams and 4-grams for BLEU-4) [33]. The modified n-gram precision refers to the ratio of the number of matched n-grams to the number of all the n-grams in the generated sequence. So far, researchers have proposed several variants of BLEU. According to a recent human study [38], the B-Norm BLEU exhibits the most consistently with human judgements on the quality of commit messages. Therefore, in this paper, we use B-Norm BLEU as one of the metrics.

ROUGE-L calculates the F-score of precision and recall based on the longest common sub-sequences (LCS) between the generated sequence and the ground truth [25]. A longer LCS indicates the higher similarity between two sentences.

METEOR calculates the harmonic mean of 1-gram precision and 1-gram recall of the generated sequence against the ground truth [3]. It also includes a penalty mechanism when the matched tokens are not adjacent.

6 RESULTS AND ANALYSIS

In this section, we first present the overall results of FIRA (RQ1) in Section 6.1, the results of the ablation study (RQ2) in Section 6.2 and human evaluation (RQ3) in Section 6.3.

Table 1: Overall commit message generation results

Model	BLEU	ROUGE-L	METEOR
LogGen [16]	8.95	10.50	8.34
NNGen [29]	9.16	11.24	9.53
CoreGen [32]	14.15	18.22	12.90
CODISUM [44]	16.55	19.73	12.83
ATOM [28]	8.35	10.17	8.73
CoRec [41]	13.03	15.47	12.04
FIRA	**17.67**	**21.58**	**14.93**

6.1 RQ1: Overall Effectiveness

Table 1 presents average of BLEU, ROUGE-L and METEOR on all the commit messages generated by FIRA and compared techniques. Figure 10 further shows the distribution of ROUGE-L in a box plot.

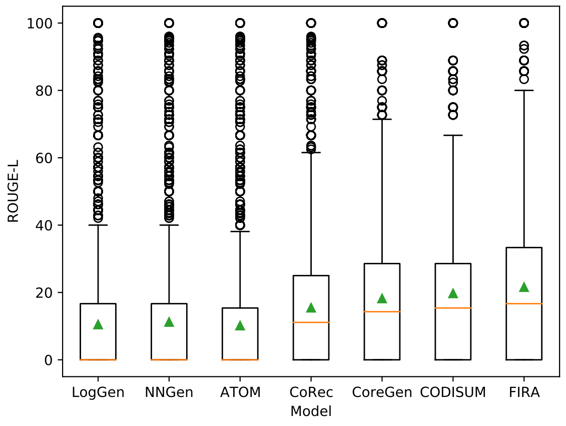


Figure 10: Box plot of ROUGE-L

As shown by the table and the figure, FIRA outperforms all the compared techniques including the best IR-based technique NNGen and the best learning-based technique CODISUM on all metrics, indicating effectiveness both in precision (i.e., BLEU, ROUGE-L and METEOR) and recall (i.e., ROUGE-L and METEOR). For example, the improvements achieved by FIRA range from 7% to 112%, 9% to 112%, and 16% to 79% in BLEU, ROUGE-L, and METEOR, respectively. In addition, it is notable that the IR-based approaches LogGen and NNGen perform significantly worse. A potential reason is that these techniques can only retrieve existing messages from the retrieved database instead of generating new commit messages. In other words, they are no longer effective, once there exists no similar code change as the given code change.

Table 2: Penalty-BLEU of all approaches

Model	LogGen	NNGen	CoreGen	CODISUM	ATOM	CoRec	FIRA
Penalty-BLEU	7.15	8.07	11.15	12.07	7.42	10.49	**13.30**

BLEU may overrate the precision of the cases that the actual number of the matched n-grams is small but the length of the commit message is even shorter, which may result in biased average of all commit messages. Therefore, for those short commit messages, we introduce a penalty mechanism by multiplying their original BLEU with a penalty factor, to reduce their impact on the final average score. The penalty factor f_i of the i-th commit message can be computed as Equation 14, which is the ratio of the length of the i-th ground truth commit message to the total length of all ground truth commit messages.

$$f_i = len(msg_i) / \sum_j len(msg_j) \tag{14}$$

We denote the BLEU with an enhanced penalty as penalty-BLEU for distinction. Table 2 presents the penalty-BLEU of all approaches. From the table, we can find that our approach also outperforms other approaches in terms of the penalty-BLEU, indicating FIRA is consistently effective on generating commit messaging of different lengths.

In summary, our quantitative results show that FIRA outperforms all six compared techniques in terms of all studied metrics; meanwhile FIRA is consistently effective on generating commit messages of different lengths.

6.2 RQ2: Ablation Study

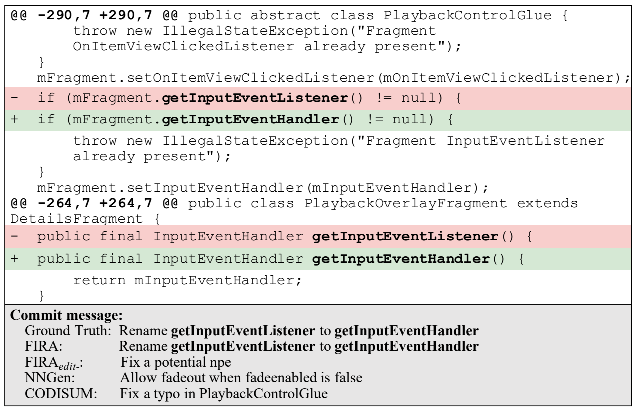
In this section, we further perform an ablation study to investigate the effectiveness of each component in FIRA. The major novelty of FIRA is explicitly including and analyzing (1) edit operations between old and new versions, and (2) copying sub-tokens with a dual copy mechanism. Therefore, to investigate their contribution, we further build two variants of FIRA by (1) removing the edit operations from the code change representation graph (i.e., denoted as FIRA$_{edit-}$), and (2) degrading the dual copy mechanism into single copy mechanism for integral tokens which cannot copy sub-tokens anymore (i.e., denoted as FIRA$_{sub-}$). In addition, we build a naive model by removing both components (i.e., denoted as FIRA$_{both-}$) for comparison. Table 3 presents the effectiveness of the default FIRA and variants. In the following sections, we then analyze the contribution of each component quantitatively and qualitatively.

Table 3: Results of the ablation study

Model	BLEU	ROUGE-L	METEOR
FIRA$_{edit-}$	17.39	21.15	14.54
FIRA$_{sub-}$	17.36	20.97	14.09
FIRA$_{both-}$	16.82	20.15	13.42
FIRA	**17.67**	**21.58**	**14.93**

6.2.1 Contribution of edit operations. As shown in Table 3, the effectiveness of FIRA$_{edit-}$ becomes worse in terms of all three metrics, indicating that including edit operations is helpful for commit message generation.

We further look into some cases that FIRA$_{edit-}$ exhibits less effective than the default FIRA in terms of these metrics. Figure 11 presents such a real-world case, which includes the code changes, the ground truth, the commit messages generated by the default FIRA, FIRA$_{edit-}$, and other compared techniques, including the best IR-based technique NNGen and the best learning-based technique CODISUM. In this example, developers rename the method from "`getInputEventListener`" to "`getInputEventHandler`" in



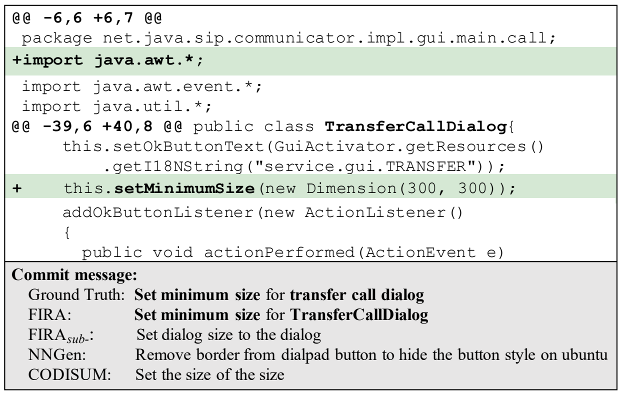
```
@@ -290,7 +290,7 @@ public abstract class PlaybackControlGlue {
        throw new IllegalStateException("Fragment
        OnItemViewClickedListener already present");
    }
    mFragment.setOnItemViewClickedListener(mOnItemViewClickedListener);
-   if (mFragment.getInputEventListener() != null) {
+   if (mFragment.getInputEventHandler() != null) {
        throw new IllegalStateException("Fragment InputEventListener
        already present");
    }
    mFragment.setInputEventHandler(mInputEventHandler);
@@ -264,7 +264,7 @@ public class PlaybackOverlayFragment extends
DetailsFragment {
-   public final InputEventHandler getInputEventListener() {
+   public final InputEventHandler getInputEventHandler() {
        return mInputEventHandler;
    }
```

Commit message:

Ground Truth:	Rename **getInputEventListener** to **getInputEventHandler**
FIRA:	Rename **getInputEventListener** to **getInputEventHandler**
FIRA$_{edit-}$:	Fix a potential npe
NNGen:	Allow fadeout when fadeenabled is false
CODISUM:	Fix a typo in PlaybackControlGlue

Figure 11: Case analysis: edit operations

two involving files, and the manual commit message exactly describes such edit operations. As shown in the figure, the default FIRA can precisely generate the exactly same message as developers, whereas after removing explicit representation of edit operations FIRA$_{edit-}$ fails to generate such correct message. In addition, we can observe that other compared techniques cannot generate the precise commit message neither, since none of them represents edit operations explicitly. Such observations further confirm our intuition that representing edit operations explicitly can help the model to capture the fine-grained code changes, enabling more precise commit message generation. On the contrary, if the old-version and new-version code are represented in combination without highlighting their differences, the model has to learn to capture such edit operations by itself, which can be challenging especially when there are only a few tokens changed.



```
@@ -6,6 +6,7 @@
 package net.java.sip.communicator.impl.gui.main.call;
+import java.awt.*;
 import java.awt.event.*;
 import java.util.*;
@@ -39,6 +40,8 @@ public class TransferCallDialog{
     this.setOkButtonText(GuiActivator.getResources()
         .getI18NString("service.gui.TRANSFER"));
+    this.setMinimumSize(new Dimension(300, 300));
     addOkButtonListener(new ActionListener()
     {
       public void actionPerformed(ActionEvent e)
```

Commit message:

Ground Truth:	**Set minimum size** for **transfer call dialog**
FIRA:	**Set minimum size** for **TransferCallDialog**
FIRA$_{sub-}$:	Set dialog size to the dialog
NNGen:	Remove border from dialpad button to hide the button style on ubuntu
CODISUM:	Set the size of the size

Figure 12: Case analysis: copying sub-tokens

6.2.2 *Contribution of copying sub-tokens.*

As shown in Table 3, the performance of FIRA$_{sub-}$ declines, indicating the dual copy mechanism for sub-tokens indeed boosts commit message generation.

We further look into some cases that FIRA$_{sub-}$ exhibits less effective than the default FIRA in terms of these metrics. Figure 12 presents such a real-world case in our dataset, including the code changes, the ground truth, the commit messages generated by the default FIRA, FIRA$_{sub-}$, and other compared techniques. In the example, the developer commit message contains several sub-tokens in the newly-added code (i.e., `setMinimumSize`) and the integral token in its belonging class name "`TransferCallDialog`". As shown in the figure, the default FIRA can effectively utilize sub-tokens in the input code, while the FIRA$_{sub-}$ without the dual copy mechanism is incapable of copying the infrequent sub-token (i.e., `minimum`) to the commit message. In addition, other compared techniques fail to generate precise commit messages neither. For example, NNGen generates a completely irrelevant commit message. For CODISUM, it includes only two sub-tokens in the generated commit message and generates the commit message with poor readability. Since CODISUM only leverages a single copy mechanism for integral token, it can only generate the frequent sub-tokens from the vocabulary (e.g., two successfully generated sub-tokens `set` and `size`) but fails to generate the infrequent sub-token (e.g., `minimum`) that is excluded in the vocabulary or seldom occurs in the training set.

Table 4: Results for copying sub-tokens

Model	Copy Ratio (%)	#Different Sub-tokens	Occurrence Frequency
NNGen	10.53	436	689
CODISUM	3.77	115	1097
FIRA$_{sub-}$	5.40	159	1118
FIRA	**11.95**	**454**	**643**

To confirm the explanation above, we further investigate whether dual copy mechanism has correctly copied sub-tokens into commit messages. In particular, we denote the sub-token appears both in the input code change and the commit message as a *copy token*. We then compute the ratio of the number of correctly-copied *copy tokens* to the number of all *copy tokens* in our testing set. A higher ratio indicates the technique is more effective in copying sub-tokens. We also present the number of different correctly-copied sub-tokens. In addition, we further present the average number of times of the correctly-copied sub-tokens occurring in the training messages, which can reflect the occurrence frequency of the sub-tokens. Table 4 presents the results. Based on the table, we can notice that FIRA can correctly copy more sub-tokens than other techniques, and FIRA can copy more infrequent sub-tokens. Furthermore, we notice that only FIRA can copy the sub-tokens never occurring in the training set. Without the dual copy mechanism, the performance of FIRA$_{sub-}$declines a lot. We can notice that NNGen performs well in terms of copying sub-token, because instead of generating new commit messages, IR-based techniques select existing messages based on the similarity of the code changes and similar code changes may have common sub-tokens. However, note that the overall performance of IR-based techniques (i.e., as shown in Table 1) is much worse than FIRA (i.e., 9.16 v.s 17.67 in BLEU). In summary, the results further indicate FIRA can copy sub-tokens effectively.

6.3 RQ3: Human Evaluation

To further study the quality of generated commit messages from the perspective of developers, we perform a human study to evaluate the commit messages generated by FIRA and compared techniques. We compare FIRA with the best retrieval-based technique NNGen and the best learning-based technique CODISUM. We invite six developers[1] to participate in this study, who have industrial experience in Java programming language ranging from 3 to 5 years.

Table 5: Scoring criterion

Score	Definition
0	Neither relevant in semantic nor having shared tokens.
1	Irrelevant in semantic but with some shared tokens.
2	Partially similar in semantic, but each contains exclusive information.
3	Highly similar but not identical in semantic.
4	Identical in semantic.

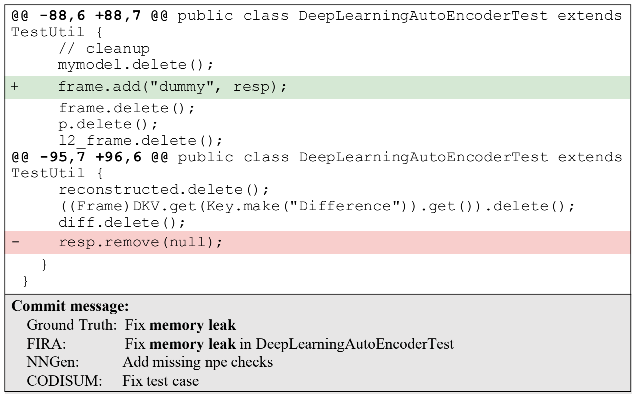
6.3.1 Study Design. Following previous work [28, 41], we randomly select 100 commits from the testing set and design a questionnaire for manual evaluation. For each commit, the questionnaire includes the code change, the ground truth commit message, and the commit messages generated by FIRA as well as the compared techniques (i.e., the best IR-based technique NNGen and the best learning-based technique CODISUM). Each invited participant is asked to score the commit messages generated by three techniques (i.e., FIRA, CODISUM, and NNGen) based on the code changes and the ground truth commit message. The score ranges from 0 to 4, and a higher score indicates a higher similarity between the generated commit message and the ground truth. We follow the existing scoring criterion [28, 29], and detailed definition is shown in Table 5. To avoid bias, all three techniques are anonymous in the questionnaire and each participant fills in the questionnaire separately.

Table 6: Results of the human evaluation

Model	Low (%)	Medium (%)	High (%)	Average Score
NNGen	71.3	13.2	15.5	0.98
CODISUM	38.0	19.8	42.2	2.06
FIRA	**35.5**	**20.3**	**44.2**	**2.15**

6.3.2 Results. For each technique, we measure the quality of its generated commit message based on the average scores of six participants on that commit message. In particular, in line with previous work [28, 29], we regard the commit messages scored 0 and 1 as low-quality, scored 2 as medium-quality, and scored 3 and 4 as high-quality. Table 6 presents the ratio of commit messages of different quality. As shown in the table, a large proportion (i.e., 44.2%) of commit messages generated by FIRA are considered as high-quality by the participants. In addition, FIRA exhibits the largest ratio of high-quality commit messages while the lowest ratio of low-quality commit messages. The average score also indicates the out-performance of FIRA over compared techniques. To confirm our observations, we further conducted a Wilcoxon signed-rank test [42] between the scores of FIRA and the other techniques. The results further confirm that difference between the scores of FIRA and NNGen/CODISUM is statistically significant at the confidence level of 95%.

6.3.3 Successful cases. We further present two cases that FIRA achieves higher scores in Figure 13 and Figure 14. Each figure includes the code change, the ground truth, and the commit messages generated by FIRA and the compared techniques, i.e., NNGen and CODISUM.

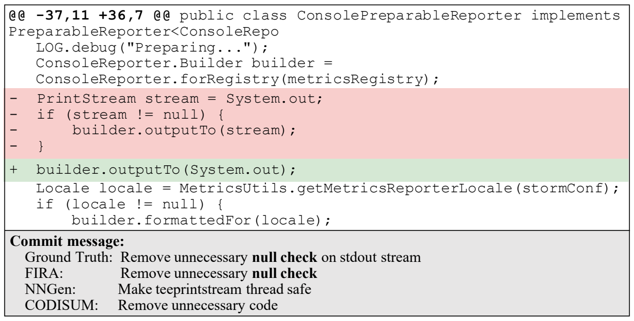


```
@@ -88,6 +88,7 @@ public class DeepLearningAutoEncoderTest extends
TestUtil {
      // cleanup
      mymodel.delete();
+     frame.add("dummy", resp);
      frame.delete();
      p.delete();
      l2_frame.delete();
@@ -95,7 +96,6 @@ public class DeepLearningAutoEncoderTest extends
TestUtil {
      reconstructed.delete();
      ((Frame)DKV.get(Key.make("Difference")).get()).delete();
      diff.delete();
-     resp.remove(null);
    }
  }
```

Commit message:
Ground Truth: Fix **memory leak**
FIRA: Fix **memory leak** in DeepLearningAutoEncoderTest
NNGen: Add missing npe checks
CODISUM: Fix test case

Figure 13: Example of fixing memory leak

The first example in Figure 13 shows the code changes for fixing a memory-leak bug. In particular, the old-version code "`resp.remove (null);`" fails to delete the object "`resp`", which results in a memory leak; while the new-version code puts "`resp`" as a member of "`frame`", which can successfully delete "`resp`" once "`frame`" is deleted. As shown in the figure, FIRA successfully predicts the intention of fixing memory leak and also the location of where memory leak occurs, which we consider as a precise commit message; whereas, other approaches fail to generate such commit messages.



```
@@ -37,11 +36,7 @@ public class ConsolePreparableReporter implements
PreparableReporter<ConsoleRepo
   LOG.debug("Preparing...");
   ConsoleReporter.Builder builder =
   ConsoleReporter.forRegistry(metricsRegistry);
-  PrintStream stream = System.out;
-  if (stream != null) {
-      builder.outputTo(stream);
-  }
+  builder.outputTo(System.out);
   Locale locale = MetricsUtils.getMetricsReporterLocale(stormConf);
   if (locale != null) {
       builder.formattedFor(locale);
```

Commit message:
Ground Truth: Remove unnecessary **null check** on stdout stream
FIRA: Remove unnecessary **null check**
NNGen: Make teeprintstream thread safe
CODISUM: Remove unnecessary code

Figure 14: Example of removing unnecessary null check

The second example in Figure 14 shows the code changes for removing an unnecessary null check, since the object "`stream`" never becomes null. As shown in the figure, FIRA appears to capture the functionality of changed code (i.e., null check) and thus generates similar message with the ground truth. In contrast, the compared approaches fail to generate proper descriptions for the given code change: NNGen generates completely irrelevant message while CODISUM generates over-general and uninformative message.

7 DISCUSSION

7.1 Threats to Validity

The internal threat to validity lies in the implementation of compared techniques and our approach. To reduce this threat, we directly reuse the implementation of the compared techniques from their reproducible packages, if their packages are available and executable [28, 29, 32, 41]; otherwise, we re-implement the techniques

[1] None of them are co-authors of this paper.

strictly following their papers. We also build our approach based on existing mature tools/libraries, such as GumTree [11]. In addition to code review, we also sampled 100 data items from our dataset and manually ensure there is no violation case where camel case or snake case is not applicable.

The external threat to validity lies in the dataset used in the experiment. To mitigate this threat, we use a well-established dataset, which has been constructed on popular Java projects from GitHub and well-cleaned by previous work [20, 44].

The construct threat lies in the metrics used in evaluation. To reduce this threat, we adopt several metrics that have been widely used by prior work on commit message generation [16, 19, 28, 29, 32, 41, 44]. In addition, we further perform a human evaluation to evaluate the effectiveness from the perspective of developers. We strictly follow the procedure of previous work [28, 41] and invite experienced developers, so as to reduce the threats in human evaluation (e.g., the limited number of participants [23]).

7.2 Limitations

The section discusses the limitations in FIRA. First, our approach would be less effective when the code change cannot be parsed into valid AST. In this case, FIRA would utilize only sub-token identifiers and sequential information during learning. Such cases are actually not observed in our dataset according to our manual inspection, which however are still possible in practice. Second, when the training set contains highly-similar code changes as the given one and the frequency of these similar data items is quite low (e.g., only once), FIRA is less effective than the retrieval-based approaches. It is a common drawback for learning-based techniques, since retrieval-based approaches can inherently retrieve the correct commit message for the similar inputs from the training set. Third, when the commit message contains tokens absent from both vocabulary and the input code change, FIRA would fail to generate these tokens in the commit message.

8 RELATED WORK

The existing work on commit message generation can be categorized as template-based, information retrieval-based, and learning-based techniques.

The template-based techniques [4, 6, 35] analyze code changes and generate commit messages with pre-defined patterns. For example, Buse and Weimer [4] design pre-defined templates based on path predicates, while Cortés-Coy et al. [6] propose templates based on method stereotypes [9] and commit stereotypes [8]. In general, the template-based techniques tend to describe what is changed but has weak capability of capturing the rationales and purposes of code changes. In addition, they are effective only when the cases perfectly fit with the pre-defined rules, but cannot be general due to the diversity of commit messages.

The information retrieval-based approaches [16, 17, 29] leverage IR techniques to adopt existing commit messages from similar code changes. For example, given a code change as a query, Liu et al. [29] leverage cosine similarity and BLEU to select a most similar code change from the training set; similarly, Huang et al. [17] use both syntax similarity and semantic similarity as the similarity metric. However, IR-based techniques are no longer effective once there is no similar code change in the retrieved database and they can only output existing commit messages instead of generating new ones.

More recently, researchers propose to leverage advanced learning techniques in commit message generation. The learning-based techniques [19, 27, 28, 32, 41, 44] regard commit message generation as a translation problem, and adopt neural machine translation (NMT) models to generate commit message for the given code change. Existing learning-based techniques first represent the old-version and new-version code with specific formats respectively, such as sequences of tokens [32] or paths of abstract syntax tree (AST) [28], concatenate both representations, and generate commit messages via different learning models. The code representations in existing learning-based techniques are coarse-grained, since (1) they represent the code changes by simply putting old-version and new-version code together, and thus edit operations have to be learned by models, and (2) they only focus on integral tokens without individually describing sub-tokens, and thus commit message with infrequent sub-tokens cannot be generated. To address these limitations, we propose a fine-grained graph-based representation for code changes to enable more powerful commit message generation. In addition to the code change representation, we propose a novel model that is different from prevision work. In particular, we leverage a graph neural network in the encoder so as to directly encode the graph-structured inputs; and we equip the decoder with the transformer and a novel dual copy mechanism, which can not only generate tokens from the vocabulary but also directly copy both integral tokens and sub-tokens from the input.

9 CONCLUSION

In this work, we propose a novel commit message generation technique, FIRA, which first represents code changes via fine-grained graphs and then learns to generate commit messages automatically. Compared to previous code change representations, FIRA explicitly describes the edit operations between the old-version and new-version code, along with tokens at different granularities. Based on the proposed graph-based representations, FIRA generates commit messages by a generation model. FIRA incorporates the graph neural network in the encoder so as to directly encode the graph-structured inputs; and the decoder incorporates the transformer and a novel dual copy mechanism, which can not only generate tokens from the vocabulary but also directly copy both integral tokens and sub-tokens from the input. We perform an extensive evaluation to compare FIRA with six commit message techniques on a widely-used benchmark. The results show that FIRA outperforms all compared techniques in terms of BLEU, ROUGE-L, and METEOR. We further analyze the effectiveness of each component in FIRA by an ablation study and case analysis. The results further confirm that major components both positively contribute to the effectiveness of FIRA. In addition, we further perform a human study to evaluate the quality of generated commit messages from the perspective of developers, which consistently shows the effectiveness of FIRA.

ACKNOWLEDGMENTS

This work was supported by National Natural Science Foundation of China under Grant No. 61872008. We are grateful for Jiashuo Liang for his help on the implementation of the transformer.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653* (2020).

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[3] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 65–72.

[4] Raymond PL Buse and Westley R Weimer. 2010. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 33–42.

[5] Eduardo C Campos and Marcelo de A Maia. 2019. Discovering common bug-fix patterns: A large-scale observational study. *Journal of Software: Evolution and Process* 31, 7 (2019), e2173.

[6] Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On automatically generating commit messages via summarization of source code changes. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 275–284.

[7] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. *arXiv preprint arXiv:1606.09375* (2016).

[8] Natalia Dragan, Michael L Collard, Maen Hammad, and Jonathan I Maletic. 2011. Using stereotypes to help characterize commits. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 520–523.

[9] Natalia Dragan, Michael L Collard, and Jonathan I Maletic. 2006. Reverse engineering method stereotypes. In *2006 22nd IEEE International Conference on Software Maintenance*. IEEE, 24–34.

[10] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 422–431.

[11] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 313–324.

[12] Marco Gori, Gabriele Monfardini, and Franco Scarselli. 2005. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, Vol. 2. IEEE, 729–734.

[13] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 1025–1035.

[14] Quinn Hanam, Fernando S de M Brito, and Ali Mesbah. 2016. Discovering bug patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 144–156.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[16] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 518–529.

[17] Yuan Huang, Nan Jia, Hao-Jie Zhou, Xiang-Ping Chen, Zi-Bin Zheng, and Ming-Dong Tang. 2020. Learning Human-Written Commit Messages to Document Code Changes. *Journal of Computer Science and Technology* 35, 6 (2020), 1258–1277.

[18] Md Rakibul Islam and Minhaz F Zibran. 2020. How bugs are fixed: exposing bug-fix patterns with edits and nesting levels. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 1523–1531.

[19] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 135–146.

[20] Siyuan Jiang and Collin McMillan. 2017. Towards automatic generation of short summaries of commits. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 320–323.

[21] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[22] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[23] Russell V Lenth. 2001. Some practical guidelines for effective sample size determination. *The American Statistician* 55, 3 (2001), 187–193.

[24] Shanshan Li, Xu Niu, Zhouyang Jia, Ji Wang, Haochen He, and Teng Wang. 2018. Logtracker: Learning log revision behaviors proactively from software evolution history. In *Proceedings of the 26th Conference on Program Comprehension*. 178–188.

[25] Chin-Yew Lin and Franz Josef Och. 2004. Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*. 605–612.

[26] Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F Bissyandé, and Yves Le Traon. 2018. A closer look at real-world patches. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 275–286.

[27] Qin Liu, Zihe Liu, Hongming Zhu, Hongfei Fan, Bowen Du, and Yu Qian. 2019. Generating commit messages from diffs using pointer-generator network. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 299–309.

[28] Shangqing Liu, Cuiyun Gao, Sen Chen, Nie Lun Yiu, and Yang Liu. 2020. ATOM: Commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software Engineering* (2020).

[29] Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 373–384.

[30] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 664–676.

[31] Zhen Ni, Bin Li, Xiaobing Sun, Tianhao Chen, Ben Tang, and Xinchen Shi. 2020. Analyzing bug fix for automatic bug cause classification. *Journal of Systems and Software* 163 (2020), 110538.

[32] Lun Yiu Nie, Cuiyun Gao, Zhicong Zhong, Wai Lam, Yang Liu, and Zenglin Xu. 2021. CoreGen: Contextualized Code Representation Learning for Commit Message Generation. *Neurocomputing* 459 (2021), 97–107.

[33] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[34] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.

[35] Jinfeng Shen, Xiaobing Sun, Bin Li, Hui Yang, and Jiajun Hu. 2016. On automatic summarization of what and why information in source code changes. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 103–112.

[36] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[37] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991.

[38] Wei Tao, Yanlin Wang, Ensheng Shi, Lun Du, Hongyu Zhang, Dongmei Zhang, and Wenqiang Zhang. 2021. On the Evaluation of Commit Message Generation Models: An Experimental Study. *arXiv preprint arXiv:2107.05373* (2021).

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[40] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[41] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. 2021. Context-aware Retrieval-based Deep Commit Message Generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–30.

[42] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.

[43] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* (2020).

[44] Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. 2019. Commit message generation for source code changes. In *IJCAI*.

[45] Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L Gaunt. 2018. Learning to represent edits. *arXiv preprint arXiv:1810.13337* (2018).

[46] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 341–353.