Fine-Grained Code-Comment Semantic Interaction Analysis

Mingyang Geng
gengmingyang13@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Shangwen Wang
wangshangwen13@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Dezun Dong
dong@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Shanzhi Gu
gushanzhi@huishiwei.cn
Hunan Huishiwei Intelligent Technology Co., Ltd.
Changsha, China

Fang Peng
pengfang21@mails.ucas.ac.cn
University of Chinese Academy of Sciences
Shenzhen, China

Weijian Ruan
rweij@whu.edu.cn
Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences
Shenzhen, China

Xiangke Liao
xkliao@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

ABSTRACT

Code comment, i.e., the natural language text to describe code, is considered as a killer for program comprehension. Current literature approaches mainly focus on comment generation or comment update, and thus fall short on explaining which part of the code leads to a specific content in the comment. In this paper, we propose that addressing such a challenge can better facilitate code understanding. We propose Fosterer, which can build fine-grained semantic interactions between code statements and comment tokens. It not only leverages the advanced deep learning techniques like cross-modal learning and contrastive learning, but also borrows the weapon of pre-trained vision models. Specifically, it mimics the comprehension practice of developers, treating code statements as image patches and comments as texts, and uses contrastive learning to match the semantically-related part between the visual and textual information. Experiments on a large-scale manually-labelled dataset show that our approach can achieve an F1-score around 80%, and such a performance exceeds a heuristic-based baseline to a large extent. We also find that Fosterer can work with a high efficiency, i.e., it only needs 1.5 seconds for inferring the results for a code-comment pair. Furthermore, a user study demonstrates its usability: for 65% cases, its prediction results are considered as useful for improving code understanding. Therefore, our research sheds light on a promising direction for program comprehension.

CCS CONCEPTS

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

KEYWORDS

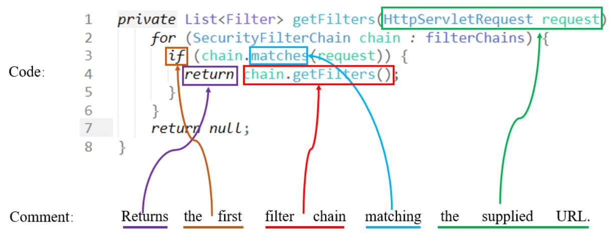
program comprehension, code comment, cross-modal learning, contrastive learning

ACM Reference Format:
Mingyang Geng, Shangwen Wang, Dezun Dong, Shanzhi Gu, Fang Peng, Weijian Ruan, and Xiangke Liao. 2022. Fine-Grained Code-Comment Semantic Interaction Analysis. In *30th International Conference on Program Comprehension (ICPC '22), May 16–17, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524610.3527887

† Shangwen Wang and Dezun Dong are the corresponding authors. This work was supported in part by Excellent Youth Foundation of Hunan Province under Grant No. 2021JJ10050.

1 INTRODUCTION

Program comprehension plays a critical role in software maintenance and evolution. This process, however, is widely known as labor-intensive since it can take up nearly 60% of developers' time, as reported by Xia *et al.* [60]. Code comment is considered as a killer to enhance the readability of code [3, 11, 58]. These comments, either written by the developers or generated by the automated approaches [10, 16], usually summarize the behaviour and/or design decisions of a code chunk (at the method level, under most conditions). Despite that these comments are shown to facilitate the comprehension process to some extent [40, 44], one thing remains challenging is the lack of fine-grained interpretation of the



```
1  private List<Filter> getFilters(HttpServletRequest request) {
2      for (SecurityFilterChain chain : filterChains) {
3          if (chain.matches(request)) {
4              return chain.getFilters();
5          }
6      }
7      return null;
8  }
```

Returns the first filter chain matching the supplied URL.

Figure 1: A motivating example from *spring-security* project.

code-comment relations. Specifically, developers still do not know which part of code corresponds to the specific contents (i.e., tokens) in the comment with only the plain comment on hand. Suppose that a developer is trying to understand a method. She finds something important for her comprehension in the comment and tries to locate the related code. Typically, she needs to browse tens of or even hundreds of lines of code for such localization, which is rather time-consuming. From this perspective, the program comprehension would be further boosted if the developers can be informed about which part of the code is related to a specific token in the comment.

The code and its corresponding comment share semantic interactions. A concrete example is shown in Figure 1. This piece of code is from the *spring-security* open-source project,[1] and the comment is written by its developers. From this example, we note that all the contents in the comment can semantically match with a part of the source code. For instance, the verb `Return` in the comment is related with the `return` statement in the code, since this statement reflects the main purpose of this method; the adjective `the first` is related with the `if` conditional statement, since the method will return a value the first time the condition in line 3 is satisfied; the attributive `matching` corresponds to the Application Programming Interface (API) called in line 3 (aka. `matches`); and so on. With these semantic interactions established, a number of program comprehension tasks can be improved, for example, comment update [21, 25]. Suppose that during the software evolution, line 3 is changed into `if (chain.contains(request)) {`, then, with the matching relations on hand, the developer would easily realize the suitable comment for the current code is “Returns the first filter chain **containing** the supplied URL” because the called API in line 3 is replaced and its name corresponds to the gerund in the comment as analyzed above. Code search [4, 28, 62] is another instance where after searching for the target code snippet with a natural language description, the interaction between the code and the description can be utilized to better help developers comprehend the search results. Therefore, the basic idea of this study is the code-comment semantic interaction analysis is beneficial for program comprehension.

In this paper, we propose Fosterer, a Fine-grained cOde-comment Semantic inTERaction analyzER, which can automatically analyze which part of code has semantic relation with a specific token in the comment and hence provide fine-grained information for code understanding. Theoretically, if this task is to be performed manually, a potential scenario would be that developers check each line of code in the method and match the line with specific tokens in the comment if they find semantic interactions between them. Our automated approach is designed to mimic this situation where we utilize a cross-modal deep neural network to learn the semantic interactions between each statement in the code and each token in the comment. Specifically, we treat each statement in the code as an image patch and use a vision Transformer to encode it, while we treat the comment as textual information and use a text Transformer for encoding. Transformer, the backbone of our approach, is a widely-used network architecture in recent years which is designed based on the *self-attention* mechanism [48]. The intuition of treating code as images is threefold: (1) by using the pre-trained model from large-scale image dataset like ImageNet [15], the knowledge hidden in the model can be leveraged for code-related tasks by fine-tuning while there lacks dataset of such large scale in source code; (2) the long-dependency problem is still a key bottleneck for sequential model and treating code like images has already achieved promising results on several code comprehension tasks [20, 24, 32]; and (3) it mimics human's practice on reading code: usually, humans understand code by reading each statement. If we rely on the token sequence of the code snippet or nodes from the corresponding AST to perform this task, we might build correlations between code tokens and comment tokens, which may be too fine-grained for developers. In contrast, by using code images, we can establish relations between code statements and comment tokens, which is a more user-friendly granularity. After embedding the code and comment respectively, we adopt contrastive learning [5] to model their semantic interactions. With contrastive learning, each image patch can be matched with the textual token that is most similar to it in the high-dimensional vector space, and similarly, each textual token is matched with its closest image patch. Therefore, when applied after training, Fosterer can bridge the interaction between a token in the comment and its related code statement(s), hence providing fine-grained comprehension information. Applying a pre-trained model on downstream tasks need tremendous training data since the pre-trained models usually have a large amount of parameters [51]. To address this challenge, we adopt data augmentation technique, which is commonly used to enrich the training dataset and make it as diverse as possible. Specifically, for code, we consider semantic-preserving transformation where semantically invariant code can be created with different implementation details. For comments, we use a *back-translation* strategy [61] to create comments with identical semantics.

To evaluate our approach, the first step is to create a benchmark with ground-truth (i.e., the semantic interactions between code and comment should be already labelled). To this end, we manually labelled 5,018 code-comment pairs from 10 open-source projects for evaluation. To our best knowledge, we are the first to target code-comment semantic interaction analysis. We therefore design a heuristic-based baseline approach which relies on analyzing the impacts of different statements on a code summarization model. Our experiment results show that Fosterer can significantly outperform the baseline. In concrete, Fosterer achieves an F1-score of nearly 80% and beats the heuristic-based baseline method by 6% on Precision metric, 5% on Recall metric and 5% on F1-score metric. We also investigate the efficiency of Fosterer and results show that once trained, it only takes 1.5 seconds for Fosterer to

[1]https://github.com/spring-projects/spring-security.

build the semantic interactions for a code-comment pair, which is an affordable time consumption. Moreover, a user study shows that the code-comment semantic interaction built by Fosterer can help participants better understand the code compared with the plain comment under most conditions (i.e., for 65% cases).

The main contributions of this paper are summarized as follows:

- We propose Fosterer, a code-comment semantic interaction analyzer built on top of cross-modal and contrastive learning techniques. Fosterer can provide fine-grained interpretation of the relationship between the code and the comment. To our best knowledge, we are the first to explore this direction in the literature.
- We build a dataset with over 5k code-comment pairs whose semantic interaction relationships have already been manually annotated. It is the first large dataset for this task. Based on it, we perform experiments to show the effectiveness of our approach.
- We open source our replication package at https://github.com/gmy2013/FOSTERER, including the dataset, the source code of Fosterer, and test results, for follow-up studies.

2 RELATED WORK

Our work is related to three research directions in the literature, including data augmentation, semantic representation learning, and multi-modality interaction mechanism.

2.1 Data Augmentation

Data augmentation aims to increase the data diversity and thus the generalization ability of the model by various transformation techniques. This approach is widely used in the computer vision domain [43, 57]. In recent years, researchers apply data augmentation to code data as well [34, 35, 38]. A series of studies are motivated by the fact that existing models are vulnerable to adversarial examples, and they design methods to expose the vulnerability of models and improve the robustness of models. In this work, we simply augments the data and feeds the augmented data into the model to improve our training process.

2.2 Semantic Representation Learning on Source Code

Deep learning techniques on program analysis have attracted much attention. Plenty of works [2, 9, 18, 22, 23, 39, 46, 47, 52–54, 59] have focused on code representation for facilitating diverse downstream software engineering tasks. These approaches mainly utilize textual or syntactic structures to model code. Bajracharya et al. [1] proposed a code search engine Sourcerer which can extract fine-grained structural information from source code. Lv et al. [29] proposed CodeHow, a code search technique which measures APIs and the queries based on text similarity, and applies an extended boolean model to retrieve code. Raychev et al. [39] adopted an RNN and n-gram model for code completion. To capture the structure information of code, Mou et al. [31] proposed a novel tree-based convolutional neural network (TBCNN) to represent the ASTs of source code. Wei et al. [56] proposed a framework (CDLH) incorporating an AST-based LSTM to exploit the lexical and syntactical information. More recently, Wan et al. [50] proposed MMAN to combine multiple semantic information of code, including tokens, AST, and graph information of source code. TBCCD (Tree-based Convolutional Clones Detection) [65] exploits tree-based convolution and position-aware character embedding technology to detect semantic clones, by capturing both the structural information of a code fragment from its AST and lexical information from code tokens. To fully capture the rich information in ASTs brought by the large size/depth, CAST [42] hierarchically splits and reconstructs ASTs. In concrete, CAST splits the AST of source code into several subtrees, embeds each subtree, and aggregates the information of subtrees back to form the full AST representation. FA-AST [55] builds a graph representation of programs called flow-augmented abstract syntax tree (FA-AST) by augmenting original ASTs with explicit control and data flow edges. IR2vec [49] exploits a concise and scalable encoding infrastructure to represent programs as a distributed embedding in continuous space. The distributed embedding is obtained by combining representation learning methods with flow information to capture the syntax as well as the semantics of the input programs. In our work, we complement literature approaches of source code representation via using computer vision and contrastive learning techniques.

2.3 Multi-Modality Interaction Mechanism

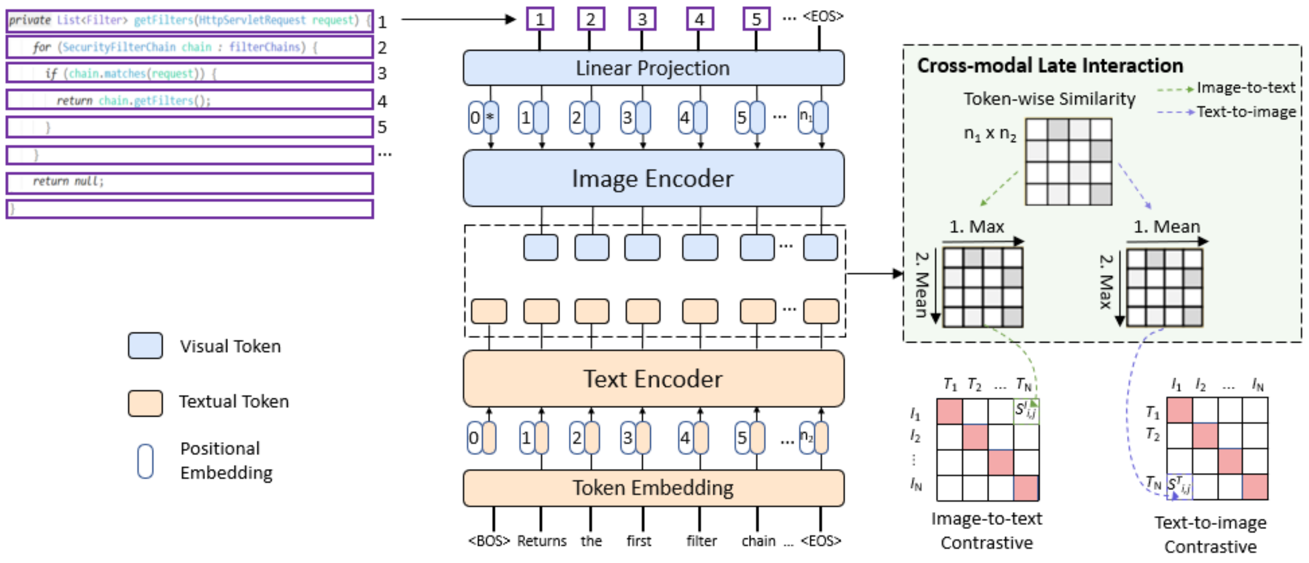
The core of vision-language pre-training models lies in modeling the interaction between the two modalities. There are mainly two types of cross-modal interaction architectures: single-stream and dual-stream models. Single-stream models like Visual-Bert [19] and ViLT [14] directly concatenate the patch-wise or regional visual features and textual embeddings, and then feed the fused vectors to the Transformer-based encoder. Dual-stream models such as ViLBERT [27] and CLIP [36] have separate encoders for different modalities. Such techniques illustrate flexible use of different models for different modalities, and efficient inference for downstream tasks like image-text retrieval, through the ability of decoupling the encoders and pre-compute image/text features offline. In this paper, we propose to exploit a new multi-model interaction mechanism [63] to capture the fine-grained representations between source code and comments.

3 METHODOLOGY

In this section, we introduce the architecture of Fosterer. We focus on the following 4 parts in detail: encoders for visual renderings of source code and comments, the contrastive learning mechanism, the cross-modal late interaction part, and the prediction part.

3.1 Encoding

Fosterer is a dual-stream model with Transformer-based image and text encoders. For the visual modality, the image encoder is a Vision Transformer, which takes the concatenation of extra <CLS> and <EOS> token embedding and linearly projected image patches as input. Our rendering is implemented using the pillow Python image drawing and manipulation library with a white background. A variation of the Plain text visualization consists in rendering the code text while highlighting syntax with colors, similarly to what is done in compilers. This rendering approach is implemented by first



```
private List<Filter> getFilters(HttpServletRequest request) {
    for (SecurityFilterChain chain : filterChains) {
        if (chain.matches(request)) {
            return chain.getFilters();
        }
    }
    return null;
}
```

Figure 2: The workflow of Fosterer.

generating an html page to highlight the code using the google code-prettify javascript library.[2] The web page is then saved as a PNG image using the imgkit, [3] which is a python wrapper for the Webkit web browser engine.[4] The syntax highlighting visual representation will highlight code structures for human programmers, which can make the semantic understanding task easier. In future, we will also explore other visualization method for source code renderings, e.g., using abstract syntax trees or graph representations.

In order to guarantee that the patching of the visual rendering or code snippet is not interrupted, we design a series of pre-processing rules. First, since the lengths of different visual rendering of source code are different, the traditional patching method of images [7] which divides the images into $N \times N$ squares cannot be directly applied because the code will lose its initial semantics once being split. Therefore, we decide to set the size of each patch to 24×1000. The height of each patch is calculated as the height of the whole visual rendering divided by the number of code lines. Our pre-study experiment shows the value is 24. By doing so, we can ensure that one code statement will not be split into different patches. The width is set to 1000 to guarantee that the token information of each line will not be discarded after patching. Furthermore, to guarantee that the width 1000 is long enough to involve each token in the line, we reformulate the indents to avoid the situation where 1000 pixels cannot accommodate the snippet line. We add an <EOS> token embedding to indicate the ending of the visual rendering since the length of the visual rendering is not fixed. In our study, the image encoder is pre-trained on the ImageNet dataset to efficiently produce embeddings that can exploit the powerful knowledge from the computer vision field.

For the textual modality, following [36], we use the lower-cased byte pair encoding (BPE) [41] with a vocabulary size of 49,408 to tokenize the text. Each text sequence starts with [BOS] token and ends with [EOS] token. After the word embedding layer, the token embeddings are fed into a modified decoder-only Transformer model as in [37]. On top of the visual rendering of source code and comment encoders, the representations of textual tokens and visual tokens are linearly projected to the multi-modal common space, and are separately L2-normalized. We do not model the cross-modal interaction via only the global features of the entire image and text sequence. Instead, we exploit a novel fine-grained contrastive learning objective equipped with cross-modal late interaction which takes into account the fine-grained interaction between image patches and textual tokens. Following the experiment setting in the previous study [63], for the image encoder, we exploit the transformers with 12 layers, 768 widths and 12 heads. For the text encoder, we exploit the transformers with 12 layers, 512 widths and 8 heads. The embedding dimension is set to 256.

[2]https://github.com/googlearchive/code-prettify.
[3]https://pypi.org/project/imgkit/.
[4]https://webkit.org/.

3.2 Contrastive Learning

Contrastive learning has recently been found to learn better representations than the predictive counterpart in both visual and vision-language cross-modal pre-training [45]. Under a general formulation of cross-modal contrastive learning, we want to learn encoders f_θ for image data $\mathcal{I}$ and g_ϕ for text data $\mathcal{T}$ such that, given an image $\boldsymbol{x}^I \in \mathcal{I}$ and a a text $\boldsymbol{x}^T \in \mathcal{T}$, the encoded representations $f_\theta\left(\boldsymbol{x}^I\right)$ and $g_\phi\left(\boldsymbol{x}^T\right)$ are close if they are related and far apart if not, under a distance metric. In each training batch, we sample b image-text pairs $\left\{\boldsymbol{x}_k^I, \boldsymbol{x}_k^T\right\}_{k=1}^b$, x_k^T. For image X in image-text pair, $\left\{\boldsymbol{x}_k^I, \boldsymbol{x}_k^T\right\}$ is its positive, while the other texts will be used as in-batch negatives. The image-to-text contrastive loss L_k^I for x_k^I can then be formulated as

$$\mathcal{L}_k^I\left(\boldsymbol{x}_k^I, \left\{\boldsymbol{x}_j^T\right\}_{j=1}^b\right) = -\frac{1}{b} \log \frac{\exp\left(s_{k,k}^I\right)}{\sum_j \exp\left(s_{k,j}^I\right)}, \tag{1}$$

where $s^I(k, j)$ denotes the similarity of the k-th image to the j-th text. Similarly, the text-to-image contrastive loss for x_k^T is

$$\mathcal{L}_k^T \left(x_k^T, \left\{ x_j^I \right\}_{j=1}^b \right) = -\frac{1}{b} \log \frac{\exp\left(s_{k,k}^T\right)}{\sum_j \exp\left(s_{j,k}^T\right)}. \quad (2)$$

The total loss of this mini-batch can be represented by

$$\mathcal{L} = \frac{1}{2} \sum_{k=1}^{b} \left(\mathcal{L}_k^I + \mathcal{L}_k^T \right). \quad (3)$$

3.3 Cross-modal Late Interaction

For the contrastive loss, the cross-modal interaction is reflected in how we compute the similarities $s_{i,j}^I$ and $s_{i,j}^T$ for the i-th image and j-th text. Instead of simply encoding each image or text separately to a global feature, we apply a cross-modal late interaction inspired by [13] to model the token-wise cross-modal interaction.

Specifically, denote n_1 and n_2 as the number of (non-padded) tokens of the i-th image and j-th text, respectively. The corresponding encoded features are $f_\theta\left(x_i^I\right) \in \mathbb{R}^{n_1 \times d}$ and $g_\phi\left(x_j^T\right) \in \mathbb{R}^{n_2 \times d}$. For the k-th visual token, we compute its similarities with all textual tokens of x_j^T, and use the largest one

$$\max_{0 \le r < n_2} \left[f_\theta\left(x_i^I\right) \right]_k^\top \left[g_\phi\left(x_j^T\right) \right]_r \quad (4)$$

as its token-wise maximum similarity with x_j^T. We then use the average token-wise maximum similarity of all non-padded tokens in the image as the similarity of an image to a text (resp. a text to an image). The similarity of the i-th image to the j-th text can thus be formulated as:

$$s_{i,j}^I\left(x_i^I, x_j^T\right) = \frac{1}{n_1} \sum_{k=1}^{n_1} \left[f_\theta\left(x_i^I\right) \right]_k^\top \left[g_\phi\left(x_j^T\right) \right]_{m_k^I}, \quad (5)$$

where $m_k^I = \arg\max_{0 \le r < n_2} \left[f_\theta\left(x_i^I\right) \right]_k^\top \left[g_\phi\left(x_j^T\right) \right]_r$. Similarly, the similarity of the j-th text to the i-th image is

$$s_{i,j}^T\left(x_i^I, x_j^T\right) = \frac{1}{n_2} \sum_{k=1}^{n_2} \left[f_\theta\left(x_i^I\right) \right]_{m_k^T}^\top \left[g_\phi\left(x_j^T\right) \right]_k, \quad (6)$$

where $m_k^T = \arg\max_{0 \le r < n_1} \left[f_\theta\left(x_i^I\right) \right]_r^\top \left[g_\phi\left(x_j^T\right) \right]_k$. Note that $s_{i,j}^I\left(x_i^I, x_j^T\right)$ in Equation X does not necessarily equals $s_{i,j}^T\left(x_i^I, x_j^T\right)$ in Equation (6).

Intuitively, the token-wise maximum similarity in Equation (4) means that for each image patch, we find its most similar textual token. Similarly, for each textual token, we also find its closest image patch. By applying this to the similarity calculation in (5) and (6) for contrastive loss (3), FOSTERER will learn fine-grained alignment between image patches and textual tokens.

3.4 Prediction

Given a visual rendering of source code and the corresponding textual comment, we now introduce how to exploit FOSTERER to discover the fine-grained interactions. First, the source code is visualized as an image by highlighting the syntax with colors on the basic of plain texts. Then, the visual rendering of source code is patched into $N \times 24 \times 1000$ patches before fed to the visual encoder, where N represents the lines of the code snippet, 24 denotes the height pixels of each patching, and 1000 represents the width pixels of the patch, which also indicates the maximum length of each statement in the visual rendering. Similarly, the textual comment will be tokenized to a series of tokens using BPE before fed to the textual encoder.

After the patching and tokenization process, the image patches and the textual tokens will be fed to the trained vision encoder and the textual encoder respectively to get the embedding representation. The word-patch alignment is performed on the basis of the token-wise similarity between the embedding of image patches and textual tokens. In concrete, for the k-th textual token, the location index of image patch with the largest similarity with it (m_k^T in Equation (6)) is considered as its matching patch. Finally, we can acquire the matching patches for all textual tokens by calculating the location index of the image patch with the largest similarity.

4 EXPERIMENTS

In this section, we introduce the research questions in this study, the dataset preparation, the details in the evaluation, as well as the experimental results.

4.1 Research Questions

To assess the effectiveness of our FOSTERER, we propose to answer the following research questions:

- **RQ1:** ***How effective is* FOSTERER *in building semantic interactions between code and comment?*** In this RQ, we seek to assess the performance of FOSTERER on code-comment semantic interaction analysis. Since FOSTERER is the first approach to target this problem, we compare FOSTERER with a heuristic-based baseline proposed by ourselves.
- **RQ2:** ***How efficient is* FOSTERER*?*** The efficiency of FOSTERER is critical for its application in practice. If it needs a lot of time to infer its results, its usefulness would be drastically decreased since developers would like to obtain the results quickly when reading code. Therefore, except for the effectiveness, we assess the efficiency of FOSTERER in this RQ.
- **RQ3:** ***To what extent can* FOSTERER *help developers better understand programs?*** In this RQ, we aim to investigate whether the prediction results from FOSTERER can better help developers comprehend code than the pure comments. The answers will reflect the potential of FOSTERER in practice.

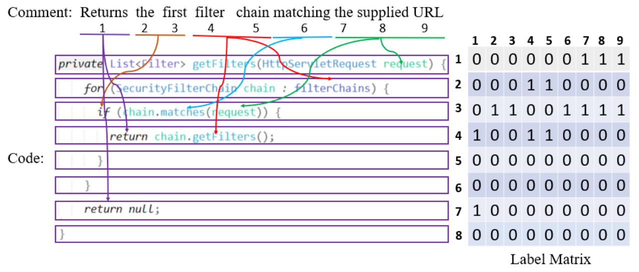
4.2 Dataset Preparation

In order to assess the effectiveness of FOSTERER on the fine-grained code-comment semantic interaction analysis ability, we need a dataset which labels the relationship between source code and comment tokens. As far as we know, there is no available dataset for this task. Therefore, we need to manually label a test set of Java programming language for our evaluation.

We first collect code and the corresponding comment from 10 randomly-selected GitHub repositories from the test set of Java part in CodeSearchNet. We first filter comments shorter than three tokens to ensure the comments in our study are descriptive. Then

Table 1: The details of our test data.

Repository	Number of Pairs
Unidata/thredds	1,656
oblac/jodd	1,042
wildfly/wildfly	432
orientechnologies/orientdb	421
rupertlssmith/lojix	389
streamsets/datacollector	344
tiefaces/TieFaces	288
ngageoint/geopackage-android	256
spring-projects/spring-security	138
ReactiveX/RxJava	52
All	5,018


Comment: Returns the first filter chain matching the supplied URL
1
2
3
4
5
6
7
8
9
Code:
Label Matrix

```
private List<Filter> getFilters(HttpServletRequest request) {
    for (SecurityFilterChain chain : filterChains) {
        if (chain.matches(request)) {
            return chain.getFilters();
        }
    }
    return null;
}
```

	1	2	3	4	5	6	7	8	9
1	0	0	0	0	0	0	1	1	1
2	0	0	0	1	1	0	0	0	0
3	0	1	1	0	0	1	1	1	1
4	1	0	0	1	1	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	1	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0

Figure 3: An example to illustrate how we label the fine-grained interactions between source code and comments.

we get in touch with our contact person in a top-tier international software company. This person helps us recruit five qualified participants, who are all software practitioners with more than five years of Java development experience. We ask each participant to label the fine-grained relationship between source code and comments in the following manner.

First, the visual renderings of source code are split into statements by line and comments are split into tokens by blanks. Then, each participant should label a 2-dimensional matrix, which indicates whether the token in the comment has semantic relations with the corresponding visual patches. The detailed example is shown in Figure 3 where the fine-grained matching relationship between the code and comment is marked in different colors. As for the 2-dimensional labeled matrix, the value in the i-th row and j-th column indicates whether the i-th image patch of source code has semantic relevance with the j-th token of the comment. Each statement can be semantic relevant with multiple tokens at the same time and vice versa. Besides, the cross validation process is implemented to eliminate the personal knowledge bias. In concrete, when two participants do not reach agreement on a sample pair, the third participant will attend and the voting mechanism will be applied to give a final judgement. Finally, we get 5,018 labels of the fine-grained interactions between source code and comments. The manually labeling process lasts for three weeks. Details of our test dataset are shown in Table 1.

4.3 Baseline Approach

Our approach is the first to address code-comment semantic interactions. Nonetheless, we design a simple heuristic-based baseline

Table 2: Statistics of the dataset used for training Fosterer.

Programming Language	Bimodal Data Pairs
Go	319,256
Java	500,754
JavaScript	143,252
PHP	662,907
Python	458,219
Ruby	52,905
All	2,137,293

approach for comparison. This baseline is based on an off-the-shelf Neural Comment Generation (NCG) model [17]. Generally speaking, given a code snippet, the model itself can generate a natural language description for it. To apply this model on our task, we remove one sentence in the code each time and use the model to generate comment according to the new code. Then we observe which part of the comment is changed (if any) and the changed part is matched with the deleted code statement. It should be noted that this process is performed for each statement so one comment token may correspond to multiple code statements like our approach.

Note that the prerequisite for applying this baseline is that this code summarization model can generate oracle comments (i.e., those identical to the human-written ones). We carefully checked our dataset and ensured that this condition is satisfied for 3,608 instances in the test set. Consequently, our evaluation is performed on these 3,608 code-comment pairs.

4.4 Training Details

4.4.1 Training dataset. Since we use constractive learning, we do not need explicit code-comment semantic interaction information in the training set. Any bimodal (i.e., containing code-comment pairs) dataset can be used. We exploit a recent large dataset provided by Husain *et al.* [12], which includes 2.1M bimodal datapoints across six programming languages (Python, Java, JavaScript, PHP, Ruby, and Go) from GitHub repositories. Data statistics are shown in Table 2.

The data comes from popular GitHub repositories and are filtered with a set of constraints and rules. Namely, (1) each project should be used by at least one other project, (2) each documentation is truncated to the first paragraph, (3) documentations shorter than three tokens are removed, (4) functions shorter than three tokens are removed, (4) functions shorter than three lines are removed, and (5) function names with substring "test" are removed.

4.4.2 Data augmentation. To obtain better generalization and data-efficiency of the model, we perform data augmentation on both visual rendering of code snippets and texts during the training phase to construct more code-comment pairs. To ensure the augmented data is semantically similar as the original one, we design nine data augmentation strategies for code snippets because the traditional data augmentation strategies (i.e., flipped or cropped) for images [43] cannot be directly applied to source code renderings. The details are shown in Table 3. In concrete, the nine strategies are the semantic-preserving transformations which could be applied to the source code. The strategies will change the lexical or structural of the initial source code while preserving the initial semantics. For

Table 3: The data augmentation techniques used for source code.

Number	Data augmentation technique
1	Changing the code format and adding blank lines
2	Renaming identifiers
3	Adjusting code statements order
4	Replacing constants
5	Changing data types
6	Substituting equivalent operators
7	Adding redundant statements
8	Substituting equivalent control structures
9	Substituting equivalent APIs

example, as for the sixth data augmentation strategy, we can change the operator "++" to "+=1". As for the eighth data augmentation strategy, we can change the "while" loop function to "for" loop function. As for the ninth data augmentation strategy, we can change the "cout" API to "printf" API. Note that these data augmentation techniques are widely used code transformation approaches. Due to the space limit, we cannot enumerate them here and readers are referred to [34, 51, 66] for more details.

For text augmentation, inspired by the text augmentation strategies in natural language processing (NLP) area, we augment the texts by exploiting back-translation strategy [61]. In concrete, the English comments are first translated to another language (Chinese, in this study) and then translated back to English. Furthermore, we manually re-write 51,224 comments by replacing the keywords with the semantic-relevant ones to increase the diversity of the comments. For example, the comment "verify that any buffers acquired by the test have been released" will be re-written as "check/inspect that any buffers gained/obtained by the test have been delivered/set free". The synonym information is obtained through using the *WordNet* database.[5] When feeding a batch of code-comment pairs during the training process, the visual rendering of source code and the comment are randomly sampled from the semantic-relevant candidates. The data augmentation process can enrich the diversity of the samples and help Fosterer strengthen the ability to capture the fine-grained interactions between the source code and comments.

4.4.3 Implementation details. To save memory and scale up the batch size, automatic mixed-precision [30] and gradient checkpoint [8] are used. Each patch of the input images is resized to 24× 1000 resolution during training and the maximum length of the image patches is limited to 30 with a <BOS> ending flag. The maximum length of the comment is set to 30 tokens according to the distribution of the training set. The training is mainly conducted on Nvidia V100 GPUs. We train Fosterer using the LAMB optimizer [64] and cosine learning rate schedule [26] with a linear warmup. Weight decay regularization is applied to all parameters except bias, layer normalization, token embedding, positional embedding and temperature in contrastive loss.

For the implementation of the contrastive loss, we also set the temperature in the softmax function to be a learnable parameter and initialize it as 0.07. For the training, we use the LAMB optimizer implemented by the cybertroniai's open-source repository.[6] For

[5]https://wordnet.princeton.edu/.
[6]https://github.com/cybertronai/pytorch-lamb

Table 4: Hyper-parameters for training Fosterer.

Hyperparameter	Value
Vocabulary Size	49408
Initial temperature	0.07
LAMB β_1	0.9
LAMB β_2	0.999
LAMB ϵ	0.0001
Warm-up iters	3000
Training epochs	30
Batch size	1024×8
Base LR	6×10^{-3}
Weight decay	3e-2

the learning rate scheduler, we first assign a base learning rate and then linearly warm it up to the peak learning rate according to the effective total batch size by a square root strategy. We note that a large weight decay is crucial to stabilize training and improve generalization. Specifically, we found that the training stability is a challenging issue when applying mix-precision training to large-scale models, i.e., the training is extremely unstable and the NaN loss easily happens. The base learning rate and weight decay are selected manually via observing the performance at the early training stage. Table 4 summarizes the hyperparameters for Fosterer's training.

4.5 Evaluation Metrics

To evaluate the effectiveness of Fosterer, we use the widely adopted precision, recall, and F1-score metrics. Precision measures to what extent the <code statement, comment token> interaction pairs predicted by Fosterer are real, and Recall measures to what extent the real <code statement, comment token> interaction pairs can be captured by Fosterer. The interaction pairs which are semantic relevant found by Fosterer are denoted as $Found_{Interaction}$, while the interaction pairs which are actually semantic relevant in our test set are denoted as $Actual_{Interaction}$. Then, the Precision, Recall and F1-score can be calculated as follows:

$$Precision = \frac{Found_{Interaction} \cap Actual_{Interaction}}{Found_{Interaction}} \tag{7}$$

$$Recall = \frac{Found_{Interaction} \cap Actual_{Interaction}}{Actual_{Interaction}} \tag{8}$$

Since precision and recall are a pair of trade-off metrics, we also introduce F1-Score taking both metrics into account.

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{9}$$

4.6 User Study

To answer RQ3, we invite 4 Ph.D students in our college majoring in software engineering to help us assess the usability of the prediction results of Fosterer on program comprehension. For each code-comment pair, we show them the semantic interaction results from Fosterer and ask them if they think the result is useful for them to better understand the code snippet compared with the pure comment. The comprehension is reflected by the understanding of the intention of each variable and statement. They assess the usefulness of each case on a 5 point Likert scale (Very Unuseful, Unuseful, Neutral, Useful, and Very Useful). Apart from scoring, they are welcomed to provide the rationale for each case's score.

We randomly select 20 code-comment pairs from our test set. Each participant assesses the prediction results of FOSTERER for 10 of them. To alleviate the bias induced by subjectivity, each case is assessed by two participants, and if their scores are different, they discuss with each other to reach a final score.

4.7 Experiment Results

In this section, we introduce our experiment results to answer the proposed research questions.

4.7.1 RQ1: Effectiveness of FOSTERER. We report the experiment results in Table 5. From the results, we note that FOSTERER achieves relatively high precision and recall. Specifically, the overall value of precision reaches 82%, which means the semantic interaction pairs reported by FOSTERER are of high probability to be correct; and the recall reaches 77%, indicating that most of the real semantic interaction pairs are identified by FOSTERER. The reason that the recall value is relatively lower than the precision value is that FOSTERER usually misses the one-to-many situation where one token may match several code statements at the same time.

Moreover, the F1-scores of FOSTERER consistently exceed those of the baseline approach on all the projects except for the *lojix* project where both approaches achieve an F1-score of 75%. Totally, the precision, recall, and F1-score of FOSTERER outperform those of the baseline by 6%, 5%, and 5% respectively. We also perform an ablation study and results show that the F1-score of the approach decreases 2.4% without the data augmentation technique. This means that such a technique, which is used to increase the generalization of the model, contributes slightly to the final effectiveness.

The F1-score of FOSTERER *nearly reaches 80%, exceeding that of the baseline which is 74% to a large extent.*

4.7.2 RQ2: EFFICIENCY OF FOSTERER. FOSTERER exploits the fine-grained matching interactions between image patches and textual comments by updating only contrastive loss, while simultaneously acquiring the ability to pre-compute source code and comment representations offline at inference, keeping both the training and inference efficient. In order to optimize the time for inference, we implement a series of operations to save the communication and computation time. First, we reduce the embedding size from 512 to 256 to decrease the number of parameters of FOSTERER. Second, we reduce the precision of the last-layer features of the image encoder and the text encoder from fp32 to fp16 before node communication and perform the multiplication in Equation (5) and Equation (6) under the reduced precision. Finally, in the similarity calculation process, for a token in comments, we select 50% image patches with the highest token-wise maximum similarity score in Equation (4) among all patches.

The results of the time consumption of FOSTERER under diverse configurations are shown in Table 6. We can see that all the three strategies can optimize the efficiency of FOSTERER to a large degree. For instance, if we only reduce the embedding dimension from 512 to 256, the inference time for each item (i.e., the code-comment pair) will be dropped from 2.81 to 2.62, a decrease of around 7%. When all the three strategies are applied together, the training time and inference time are optimized to 1.61 and 1.57 second per code-comment pair, respectively. Once trained, our model can be applied for multiple times, thus it will take a little time (i.e., around 1.5 seconds) for developers to obtain the prediction results in practice, which is an affordable time consumption.

With a number of lightweight strategies, the time consumption of FOSTERER *to make prediction for a code-comment pair is about 1.5 seconds, which is quite efficient for applying in practice.*

Table 5: The effectiveness of semantic interaction analysis of FOSTERER compared with that of the baseline.

Project	Tool	Precision	Recall	F1-score
Unidata/thredds	Baseline	74%	72%	73%
	FOSTERER	81%	77%	79%
oblac/jodd	Baseline	73%	71%	72%
	FOSTERER	80%	76%	78%
wildfly/wildfly	Baseline	78%	74%	76%
	FOSTERER	82%	78%	80%
orientechnologies/orientdb	Baseline	76%	72%	74%
	FOSTERER	81%	73%	77%
rupertlssmith/lojix	Baseline	77%	73%	75%
	FOSTERER	77%	74%	75%
streamsets/datacollector	Baseline	73%	71%	72%
	FOSTERER	83%	74%	77%
tiefaces/TieFaces	Baseline	79%	75%	77%
	FOSTERER	83%	79%	78%
ngageoint/geopackage-android	Baseline	74%	68%	71%
	FOSTERER	85%	79%	82%
spring-projects/spring-security	Baseline	75%	73%	74%
	FOSTERER	84%	78%	81%
ReactiveX/RxJava	Baseline	79%	73%	76%
	FOSTERER	82%	78%	80%
Total	Baseline	76%	72%	74%
	FOSTERER	**82%**	**77%**	**79%**

Table 6: Efficiency study of FOSTERER. We report the training time and inference time respectively. The training process is exploited on 8 V100 GPUs, with a batch size of 512 per GPU.

Embed dim	Embed precision	Token (%)	Training time (sec/item)	Inference time (sec/item)
512	fp32	100%	2.85	2.81
512	fp16	100%	2.67	2.62
256	fp16	100%	2.32	2.29
256	**fp16**	**50%**	**1.61**	**1.57**

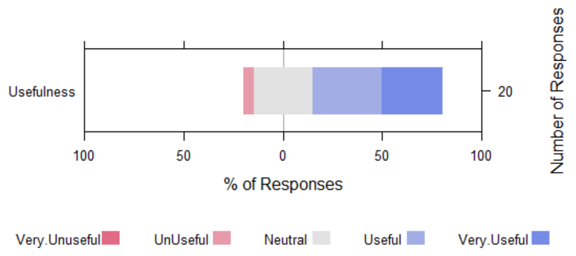
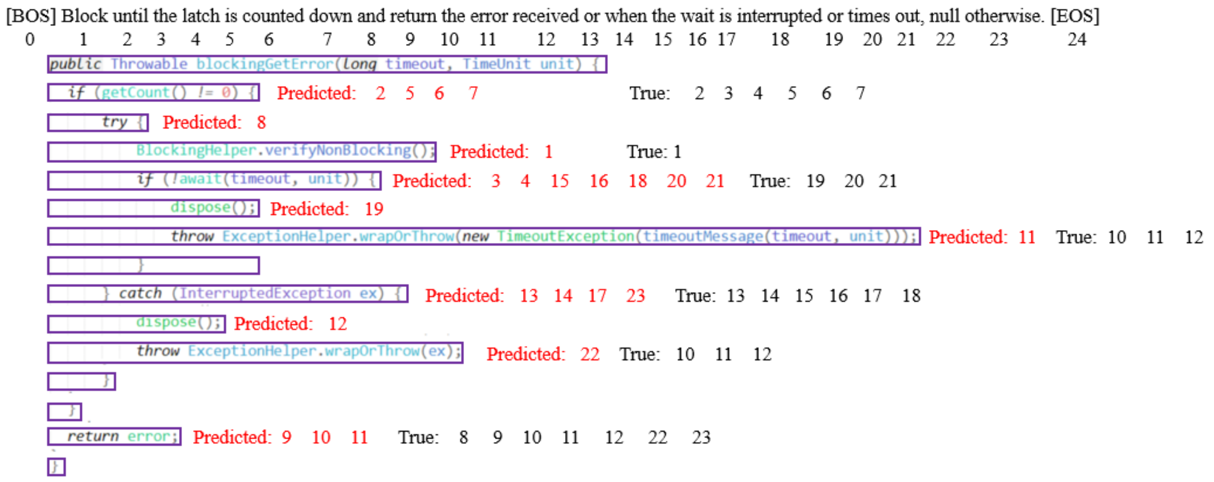


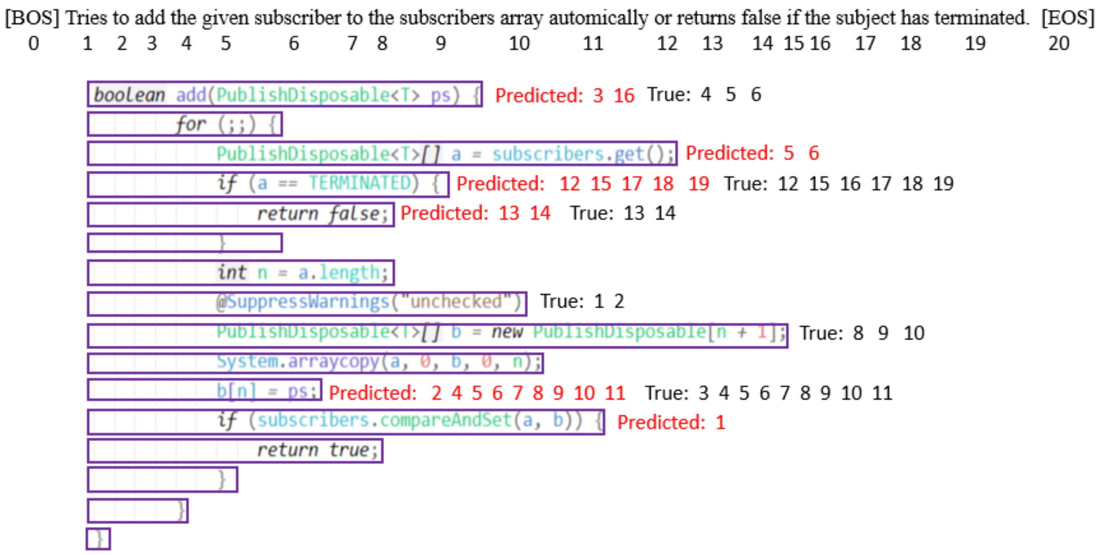
Figure 4: The results of our user study.



```
[BOS] Block until the latch is counted down and return the error received or when the wait is interrupted or times out, null otherwise. [EOS]
  0     1    2  3  4  5   6     7   8   9  10  11     12  13  14  15 16 17   18    19  20 21  22   23      24

public Throwable blockingGetError(long timeout, TimeUnit unit) {
    if (getCount() != 0) {                    Predicted: 2 5 6 7              True: 2 3 4 5 6 7
        try {                                 Predicted: 8
            BlockingHelper.verifyNonBlocking();    Predicted: 1               True: 1
            if (!await(timeout, unit)) {      Predicted: 3 4 15 16 18 20 21   True: 19 20 21
                dispose();                    Predicted: 19
                throw ExceptionHelper.wrapOrThrow(new TimeoutException(timeoutMessage(timeout, unit)));  Predicted: 11  True: 10 11 12
            }
        } catch (InterruptedException ex) {   Predicted: 13 14 17 23          True: 13 14 15 16 17 18
            dispose();                        Predicted: 12
            throw ExceptionHelper.wrapOrThrow(ex);   Predicted: 22            True: 10 11 12
        }
    }
    return error;                             Predicted: 9 10 11              True: 8 9 10 11 12 22 23
}
```

Figure 5: Visualizations on a sample selected from GitHub repository *RxJava*. The first line represents the comment. For each token in comments, the matched patch label predicted by FOSTERER is shown in red. The true label is shown in black. The visualization rendering of code snippets after patching by line is shown below.



```
[BOS] Tries to add the given subscriber to the subscribers array automically or returns false if the subject has terminated. [EOS]
  0     1   2  3   4    5        6       7  8      9        10        11    12   13   14 15 16  17   18    19      20

boolean add(PublishDisposable<T> ps) {        Predicted: 3 16   True: 4 5 6
    for (;;) {
        PublishDisposable<T>[] a = subscribers.get();   Predicted: 5 6
        if (a == TERMINATED) {                 Predicted: 12 15 17 18 19   True: 12 15 16 17 18 19
            return false;                      Predicted: 13 14   True: 13 14
        }
        int n = a.length;
        @SuppressWarnings("unchecked")                             True: 1 2
        PublishDisposable<T>[] b = new PublishDisposable[n + 1];   True: 8 9 10
        System.arraycopy(a, 0, b, 0, n);
        b[n] = ps;                             Predicted: 2 4 5 6 7 8 9 10 11   True: 3 4 5 6 7 8 9 10 11
        if (subscribers.compareAndSet(a, b)) { Predicted: 1
            return true;
        }
    }
}
```

Figure 6: Another sample from GitHub repository *RxJava*. The information is illustrated in the same way as Fig. 5.

4.7.3 RQ3: User study. Results of our user study are shown in Fig. 4. For 6 out of the 20 code-comment pairs, the prediction results of FOSTERER are considered as *Very Useful* to help programmers understand the code, and the number of *Useful* is 7. Such results also demonstrate the rationale of the basic point of our study that building accurate code-comment semantic interactions can boost program comprehension. Our participants only find one code-comment instance where the semantic interaction information from FOSTERER hinders the comprehension process. In such a case, one noun in the comment should be matched with a parameter in the method signature, while FOSTERER fails to establish such a relationship. In the contrary, it matches this token with a statement in the method body. Such a result may cast side effect since it misses a number of critical information such as what type is the parameter and how it is operated from the beginning. Thus, our participants consider the prediction results as *Unuseful.*

In our user study, the prediction results of FOSTERER *are considered as useful or very useful for 65% (13/20) cases.*

5 DISCUSSION

5.1 Qualitative Analysis

In RQ1, we have analyzed the effectiveness of FOSTERER quantitatively. In this section, we provide qualitative case study to investigate what is captured by FOSTERER and what is missed. We randomly select 100 code-comment pairs from the test set and manually compare the matching relations built by FOSTERER with the oracles to identify the strengths and weakness of FOSTERER.

We mainly find that FOSTERER is good at capturing the semantics inherent in the control statements like `If` conditional statements. An example from our test set is shown in Fig. 5. In this case, FOSTERER make good predictions for the tokens corresponding to the control statements. For instance, it matches the conditional statement `if (getCount() != 0)` with the condition in the comment (i.e., **is counted down**), which captures the semantic of this `If` statement accurately. In another case, it successfully captures the relation between the bound condition in the comment (i.e., **or when the wait is interrupted**) and the `Catch` statement in the code `catch (InterruptedException ex)`.

Nonetheless, FOSTERER does have some limitations. The most significant one is, as pointed out in our user study, it tends to miss semantic information from method signatures. Specifically, a number of comment tokens should have semantic relations with the method parameters but FOSTERER does not make such predictions. This situation becomes worse if the parameter is not named as the token in the comment. A concrete example is shown in Fig. 6. In this example, FOSTERER also demonstrates its capability of capturing the semantic of Control statements. For instance, it precisely matches the description **or return false if the subject has terminated** with the conditional statement `if (a == TERMINATED)`. It, however, fails to match the **given subscriber** with the parameter of this method. The assignment operation in the method (i.e., `b[n] = ps`) is related with the parameter *ps* and thus our annotators believe *ps* refers to *the given subscriber* in the comment and build this relationship. In contrast, FOSTERER assigns these tokens to another statement where the token *subscriber* occurs, which is a mismatching. Our analysis shows the strengths and weaknesses of FOSTERER, which indicates that there is still improvement space for future works.

5.2 Threats to Validity

Internal threats. Any manually-created dataset may face the bias from subjectivity. Thus there may exist bias in our labelled code-comment semantic interaction dataset. This threat, however, is alleviated considering that (1) the participants for dataset labelling are all experienced developers; and (2) cross-checking is applied in the labelling process which can ensure one code-comment pair is checked by at least two participants. Furthermore, we open source the largest dataset on this task so far in our replication package for the community's review.

Another threat is that in our user study, we randomly select 20 cases as the experiment objects. This process is suffered from randomness. However, we cannot burden our participants with too much workload and we checked that FOSTERER achieves an average F1-score of 78% on these cases, a performance that is similar to its overall result on the whole dataset.

In our user study, the initial target is to judge if our motivation for this study holds in practice, i.e., if providing developers with the code-comment semantic interactions can help them comprehend programs. We do not provide participants with results from our baseline approach since we are afraid that if we provide them with the results from two tools, they may focus more on comparing the differences between the results rather than assessing whether the results from a single tool are helpful. The qualitative comparison with the baseline is left as our future work.

FOSTERER is the first to tackle the ambition of code-comment semantic-interaction analysis. Apart from our designed baseline, other approaches could also be proposed to address this task, among which the traditional traceability approach [6] is a possible one. Such comparisons are left in future.

External threats. External threats to validity mainly relate to the representativeness of our dataset. Our dataset collects Java code-comment pairs from 10 open-source highly-ranked repositories, and they are also included in another widely-known benchmark, CodeSearchNet [12]. Considering that the labelling process is quite time-consuming, enlarging the scale of our dataset is considered as our future work. All comments used in our evaluation are about functionality description (the category A in the previous study [33]). We believe this is the only suitable type for performing the code-comment semantic interaction analysis task, for example, we cannot expect to build relations between the ToDo comments and the code.

In RQ1, to fairly compare against the baseline, we only evaluate the effectiveness of FOSTERER on a part of cases in our dataset (those where the code summarization model can generate oracle comments). We also evaluate the effectiveness of FOSTERER on the whole dataset and results reveal it can achieve 76% for precision, 73% for recall, and 75% for F1-score. Such a performance experiences slightly decrease compared with that shown in Table 5. From this perspective, FOSTERER and automated code summarization tools seem to face the similar challenges.

6 CONCLUSION AND FUTURE WORK

In this paper, we propose FOSTERER, a tool that can automatically analyze which part of code has semantic relation with a specific token in the comment. The core idea is that program comprehension can be enhanced by establishing fine-grained interaction relationship between code statements and comment tokens. The approach is based on cross-modal learning and contrastive learning, and also borrows the weapon of pre-trained models. Experiments have shown that (1) FOSTERER can effectively capture the fine-grained semantic interactions between visual renderings of source code and comments compared with a heuristic-based baseline approach; (2) FOSTERER can be applied in practice with an affordable time consumption; and (3) programmers consider the prediction results of FOSTERER as useful for better understanding the code for most conditions. According to the results of our user study, we believe this study points out a promising direction for boosting program comprehension for future studies.

In future, we will further enlarge the scale of our labeled dataset to perform more comprehensive evaluation. Besides, we will explore other data augmentation and AutoML technologies and design a more advanced image encoder as well as an interaction layer to further improve the results of fine-grained code-comment semantic interaction analysis. Finally, we will design pipelines which integrate our FOSTERER for better performing downstream tasks such as automatic comment update.

REFERENCES

[1] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 681–682.

[2] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. 2016. Learning python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307* (2016).

[3] Arianna Blasi and Alessandra Gorla. 2018. Replicomment: identifying clones in code comments. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 320–323.

[4] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 964–974.

[5] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR, 1597–1607.

[6] Barthélémy Dagenais and Martin P Robillard. 2014. Using traceability links to recommend adaptive changes for documentation evolution. *IEEE Transactions on Software Engineering* 40, 11 (2014), 1126–1146.

[7] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[8] Andreas Griewank and Andrea Walther. 2000. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)* 26, 1 (2000), 19–45.

[9] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[10] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 200–20010.

[11] Yuan Huang, Shaohao Huang, Huanchao Chen, Xiangping Chen, Zibin Zheng, Xiapu Luo, Nan Jia, Xinyu Hu, and Xiaocong Zhou. 2020. Towards automatically generating block comments for code snippets. *Information and Software Technology* 127 (2020), 106373.

[12] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[13] Omar Khattab and Matei Zaharia. 2020. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 39–48.

[14] Wonjae Kim, Bokyung Son, and Ildoo Kim. 2021. Vilt: Vision-and-language transformer without convolution or region supervision. *arXiv preprint arXiv:2102.03334* (2021).

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012), 1097–1105.

[16] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension*. 184–195.

[17] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 795–806.

[18] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2017. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573* (2017).

[19] Liunian Harold Li, Mark Yatskar, Da Yin, Cho-Jui Hsieh, and Kai-Wei Chang. 2019. Visualbert: A simple and performant baseline for vision and language. *arXiv preprint arXiv:1908.03557* (2019).

[20] Zheng Li, Yonghao Wu, Bin Peng, Xiang Chen, Zeyu Sun, Yong Liu, and Deli Yu. 2021. SeCNN: A semantic CNN parser for code comment generation. *Journal of Systems and Software* 181 (2021), 111036.

[21] Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F. Bissyandé. 2021. Automated Comment Update: How Far are We?. In *2021 29th International Conference on Program Comprehension (ICPC)*.

[22] Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. 2022. Context-aware code change embedding for better patch correctness assessment. *ACM Transactions on Software Engineering and Methodology* (2022).

[23] Chang Liu, Xin Wang, Richard Shin, Joseph E Gonzalez, and Dawn Song. 2016. Neural code completion. (2016).

[24] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1–12.

[25] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. 2020. Automating Just-In-Time Comment Updating. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM.

[26] Ilya Loshchilov and Frank Hutter. 2016. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983* (2016).

[27] Jiasen Lu, Dhruv Batra, Devi Parikh, and Stefan Lee. 2019. Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks. *arXiv preprint arXiv:1908.02265* (2019).

[28] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28.

[29] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 260–270.

[30] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. 2017. Mixed precision training. *arXiv preprint arXiv:1710.03740* (2017).

[31] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2014. Convolutional neural networks over tree structures for programming language processing. *arXiv preprint arXiv:1409.5718* (2014).

[32] Jordan Ott, Abigail Atchison, Paul Harnack, Natalie Best, Haley Anderson, Cristiano Firmani, and Erik Linstead. 2018. Learning lexical features of programming languages from imagery using convolutional neural networks. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 336–3363.

[33] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. 2019. Classifying code comments in Java software systems. *Empirical Software Engineering* 24, 3 (2019), 1499–1537.

[34] Erwin Quiring, Alwin Maier, and Konrad Rieck. 2019. Misleading authorship attribution of source code using adversarial learning. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 479–496.

[35] Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology* 135 (2021), 106552.

[36] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. *arXiv preprint arXiv:2103.00020* (2021).

[37] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[38] Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2020. Semantic robustness of models of source code. *arXiv preprint arXiv:2002.03043* (2020).

[39] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 419–428.

[40] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing automatic evaluation metrics for code summarization tasks. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1105–1116.

[41] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909* (2015).

[42] Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. CAST: Enhancing Code Summarization with Hierarchical Splitting and Reconstruction of Abstract Syntax Trees. *arXiv preprint arXiv:2108.12987* (2021).

[43] Connor Shorten and Taghi M Khoshgoftaar. 2019. A survey on image data augmentation for deep learning. *Journal of Big Data* 6, 1 (2019), 1–48.

[44] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*. 2–13.

[45] Yonglong Tian, Dilip Krishnan, and Phillip Isola. 2020. Contrastive multiview coding. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XI 16*. Springer, 776–794.

[46] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 269–280.

[47] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 542–553.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[49] S VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and YN Srikant. 2020. Ir2vec: Llvm ir based scalable program embeddings. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 4 (2020), 1–27.

[50] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 13–25.

[51] Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. 2022. Bridging Pre-trained Models and Downstream Tasks for Source Code Understanding. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE.

[52] Shangwen Wang, Kui Liu, Bo Lin, Li Li, Jacques Klein, Xiaoguang Mao, and Tegawendé F Bissyandé. 2021. Beep: Fine-grained Fix Localization by Learning to Predict Buggy Code Elements. *arXiv preprint arXiv:2111.07739* (2021).

[53] Shangwen Wang, Ming Wen, Bo Lin, and Xiaoguang Mao. 2021. Lightweight Global and Local Contexts Guided Method Name Recommendation with Prior Knowledge. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 741–753. https://doi.org/10.1145/3468264.3468567

[54] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated Patch Correctness Assessment: How Far are We?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 968–980. https://doi.org/10.1145/3324884.3416590

[55] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.

[56] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.. In *IJCAI*. 3034–3040.

[57] Jerry Wei, Arief Suriawinata, Louis Vaickus, Bing Ren, Xiaoying Liu, Jason Wei, and Saeed Hassanpour. 2019. Generative image translation for data augmentation in colorectal histopathology images. *Proceedings of machine learning research* 116 (2019), 10.

[58] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 53–64.

[59] Hongjun Wu, Zhuo Zhang, Shangwen Wang, Yan Lei, Bo Lin, Yihao Qin, Haoyu Zhang, and Xiaoguang Mao. 2021. Peculiar: Smart Contract Vulnerability Detection Based on Crucial Data Flow Graph and Pre-training Techniques. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 378–389. https://doi.org/10.1109/ISSRE52982.2021.00047

[60] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (2017), 951–976.

[61] Qizhe Xie, Zihang Dai, Eduard Hovy, Minh-Thang Luong, and Quoc V Le. 2019. Unsupervised data augmentation for consistency training. *arXiv preprint arXiv:1904.12848* (2019).

[62] Shuhan Yan, Hang Yu, Yuting Chen, Beijun Shen, and Lingxiao Jiang. 2020. Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 344–354.

[63] Lewei Yao, Runhui Huang, Lu Hou, Guansong Lu, Minzhe Niu, Hang Xu, Xiaodan Liang, Zhenguo Li, Xin Jiang, and Chunjing Xu. 2021. FILIP: Fine-grained Interactive Language-Image Pre-Training. *arXiv preprint arXiv:2111.07783* (2021).

[64] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2019. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962* (2019).

[65] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural detection of semantic code clones via tree-based convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 70–80.

[66] Shiwen Yu, Ting Wang, and Ji Wang. 2022. Data Augmentation by Program Transformation. *Journal of Systems and Software* (2022).