Wuhan University
Journal of Natural Sciences

2025, Vol.30 No.3, 222-230

Article ID 1007-1202(2025)03-0222-09 DOI https://doi.org/10.1051/wujns/2025303222

Cite this article: ZOU Zhou, ZUO Zhengkang, HUANG Qing. AI Chain-Driven Control Flow Graph Generation for Multiple Programming Language[J]. *Wuhan Univ J of Nat Sci*, 2025, 30(3): 222-230.

AI Chain-Driven Control Flow Graph Generation for Multiple Programming Language

□ **ZOU Zhou[1,2,3], ZUO Zhengkang[1,2,3†], HUANG Qing[1,2,3]**

1. State International Science & Technology Cooperation Base of Networked Supporting Software, Jiangxi Normal University, Nanchang 330022, Jiangxi, China;

2. Jiangxi Provincial Key Laboratory for High Performance Computing, Jiangxi Normal University, Nanchang 330022, Jiangxi, China;

3. Language Intelligence Research Center, Jiangxi Normal University, Nanchang 330022, Jiangxi, China

Abstract: Control Flow Graphs (CFGs) are essential for understanding the execution and data flow within software, serving as foundational structures in program analysis. Traditional CFG construction methods, such as bytecode analysis and Abstract Syntax Trees (ASTs), often face challenges due to the complex syntax of programming languages like Java and Python. This paper introduces a novel approach that leverages Large Language Models (LLMs) to generate CFGs through a methodical Chain of Thought (CoT) process. By employing CoT, the proposed approach systematically interprets code semantics directly from natural language, enhancing the adaptability across various programming languages and simplifying the CFG construction process. By implementing a modular AI chain strategy that adheres to the single responsibility principle, our approach breaks down CFG generation into distinct, manageable steps handled by separate AI and non-AI units, which can significantly improve the precision and coverage of CFG nodes and edges. The experiments with 245 Java and 281 Python code snippets from Stack Overflow demonstrate that our method achieves efficient performance on different programming languages and exhibits strong robustness.

Key words: Control Flow Graph; Large Language Model; Chain of Thought; AI chain

CLC number: TP311.5

0 Introduction

The Control Flow Graph (CFG) serves as a cornerstone in software engineering, illustrating program behavior by showcasing statement sequence and the conditions governing their execution order[1]. As a graphical representation of program behavior, CFG plays a crucial role in numerous software engineering tasks, including code search[2-3], code clone detection[4-5] and code classification[6-7]. These applications help improve code quality and software performance.

Developers usually use static analysis methods to generate CFG for the given code. Static analysis methods can mainly be divided into two types: methods

Received date: 2024-09-25 © Wuhan University 2025

Foundation item: Supported by the National Natural Science Foundation of China (62462036, 62262031), Jiangxi Provincial Natural Science Foundation (20242BAB26017, 20232BAB202010), Distinguished Youth Fund Project of the Natural Science Foundation of Jiangxi Province (20242BAB23011), and the Jiangxi Province Graduate Innovation Found Project (YJS2023032)

Biography: ZOU Zhou, male, Master candidate, research direction: software engineering. E-mail: zzouzhou@jxnu.edu.cn

† Corresponding author. E-mail: zuo803@jxnu.edu.cn

This is an Open Access article distributed under the terms of the Creative Commons Attribution License (https://creativecommons.org/ licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

based on bytecode[8-9] and methods based on Abstract Syntax Tree (AST)[10]. The bytecode-based method first compiles the source code into bytecode (such as JVM instructions), and then analyzes it to identify the basic blocks that constitute the CFG. The AST-based method directly generates structured CFG from the abstract syntax tree of the source code by applying a set of predefined parsing rules.

The static analysis method has the advantage of generating CFG with high accuracy, but it also has the following obvious limitations: 1) The static analysis method cannot generate multi-language CFG across language boundaries. For example, bytecode-based methods can be well used in statically typed languages such as Java. In contrast, the dynamic features of Python (such as dynamic types and runtime modifications) make the static analysis of bytecode control flow very difficult. In addition, the AST approach requires manually customizing different rule sets according to the syntactic characteristics of each programming language (Java's "for loop" and Python's "for-in loop"), which limits scalability. 2) For developers with limited experience, traditional hard-coded static analysis methods bring high learning costs. Whether using bytecode or AST, developers require a thorough understanding of compiler principles and the underlying implementation details of specific programming languages.

Although different programming languages maintain the same basic logical structures such as sequences, selections and loops, they use unique syntactic forms to express similar semantics. Because the grammar of each language is different, parsing these different expressions brings challenges. Large language models (LLMs) can effectively understand and interpret these syntactic differences through extensive pre-training on multilingual programming data. This function enables LLMs to recognize equivalent semantic expressions across languages, making it particularly effective for tasks such as CFG generation[11]. Furthermore, LLM uses natural language as input, significantly reducing the technical barriers for users[12].

However, it is very difficult to generate CFG nodes and edges directly from the code snippets. LLMs often struggle to recognize and process the boundaries of different structures within the input code. For example, two code statements "if (i==1) return true;" may be treated as one node, instead of being treated as two separate nodes. This may lead to a critical loss of control flow information in the generated CFG. To alleviate this problem, we develop a fine-grained Chain of Thought (CoT) method[13-14]. It involves four steps: structure hierarchy extraction to identify nested structures, nested code block extraction to obtain code blocks at each structure, CFG generation of nested code blocks, and graph fusion to integrate all nested code blocks' CFGs.

Although the fine-grained CoT prompt method is effective, it encounters limitations when a single prompt contains all procedural responsibilities. This may lead to the accumulation of errors and generate overly complex "epic" prompts that are difficult to optimize and control[15-16]. To solve this problem, we use the single responsibility principle in software engineering to break down this "epic" prompt into an AI chain[17-18]. Each step is handled by an independent AI or non-AI unit, and tailor-made prompts are provided for each AI interaction. This modular strategy allows for gradual contact with LLM and promotes more precise and controlled CFG generation.

To evaluate the effectiveness of our CFG-Chain method, we collect 245 Java and 281 Python code snippets from the first 50 pages of recent Stack Overflow posts tagged with Java and Python. We first evaluate the performance of each AI unit in CFG-Chain using GPT-4. Each AI unit achieves over 80% accuracy on both Java and Python datasets, which confirms the efficacy of our designed prompting strategy and the overall reliability of our method. Next, to evaluate the generalization of CFG-Chain across different LLMs, we apply CFG-Chain to GPT-4 and the Code LLaMa series models. Experimental results show that CFG-Chain consistently achieves strong performance across different LLMs, and the performance improves as the model parameters increase.

The main contributions of this paper are as follows:

- We propose a universal CFG generation method tailored to accommodate the diverse syntactic expressions found across programming languages. This method begins by using an epic CoT prompt to help LLMs identify structural boundaries within nested code structures.
- To enhance boundary detection in LLM-based code analysis, we refine our method by decomposing complex code into atomic-level code blocks. For each block, we generate a CFG subgraph, which significantly improves the accuracy of delineating structural boundaries.

1 CFG-Chain Framework

Our approach for CFG generation uses the LLMs' deep understanding of the varied syntactic expressions to generate the node and edge representation of a CFG for different programming languages. The overall framework of our CFG-Chain is shown in Fig. 1. We follow a human-like thinking process to decompose the task into single-responsibility sub-problems and design functional units. These units are linked in serials, in parallel, or with split-merge structure to create a step-by-step interaction with the LLM. We use GPT-4[19] as underlying LLM. It is important to note that our method mainly focuses on the task itself including its characteristics, data properties, and information flow, rather than on the selection of LLMs. Thus, besides GPT-4, other LLMs are also applicable to our approach.

1.1 Hierarchy Task Decomposition

LLMs usually struggle with boundary awareness, which can lead to incorrect CFGs. This issue becomes more pronounced as code nesting complexity increases, making it difficult to generate accurate CFGs with a single instruction like "Generate the CFG of the given code". We improve the instruction by decomposing it into multiple detailed sub-instructions. Each sub-instruction is executed through a separate LLM call, which helps the LLM better detect boundaries in complex nested code.

To develop a reasonable task decomposition, we analyze the code in Fig. 2(a) and find that the code has hierarchical nesting relationships with three layers marked by green, orange, and blue. Each layer contains code blocks that can be processed in the same way to generate the CFG for that layer. Therefore, we design a "recursively nested code replacement" strategy to process the nested code blocks layer by layer. Beginning with the innermost code block (depicted as blue in Fig. 2(a)), we convert it into a CFG, and replace the code block with a specified string (referred to as "code masking") by a non-AI unit, then work outward through the higher layer (shown in orange in Fig. 2(a)) until all layers are replaced. This method allows us to decompose the CFG generation task into a chain of AI steps, each with a small and manageable unit, as shown in Fig. 1.

The first AI unit, called structure hierarchy extraction, identifies the code layers of the code structure. Its output guides the next AI unit, called nested code block extraction, to extract the code blocks from each code layer. Then, a non-AI unit called Code Mask with Nested Code Blocks replaces the extracted code blocks with the specified string. The process repeats the execution of the two AI units until all code layers' code blocks are extracted. Next, the nested code blocks are input into the CFG generation of nested code block AI unit, which generates individual CFGs for each code block. During this step, a non-AI unit called Atomic Block-CFG examples retrieval provides prompt examples specific to the nested code block. Finally, the graph fusion AI unit integrates the nodes and edges of all nested code block's CFG, assembling them into a final complete CFG.

1.2 Running Example

To illustrate how the AI units collaborate and transform data throughout the process, we demonstrate with a nested Java code example in Fig. 2.

The process begins by entering the Java code into the structure hierarchy extraction unit. This unit identifies the different layers of the Java code, assigns each structure a unique identifier, and represents the code's nested structure statements by indentation. As shown in Fig. 2(a), for the Java code "for {⋯while {if {⋯}}}", the unit outputs "for_block_1 while_block_1 if_block_1".

Next, the Java code and its nested layers are input into the nested code block extraction unit. This unit itera-

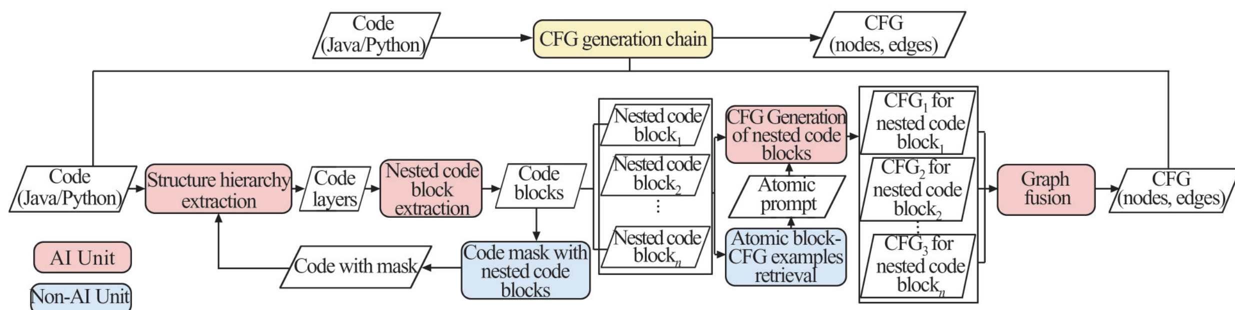

Fig. 1 Overall framework of CFG-Chain

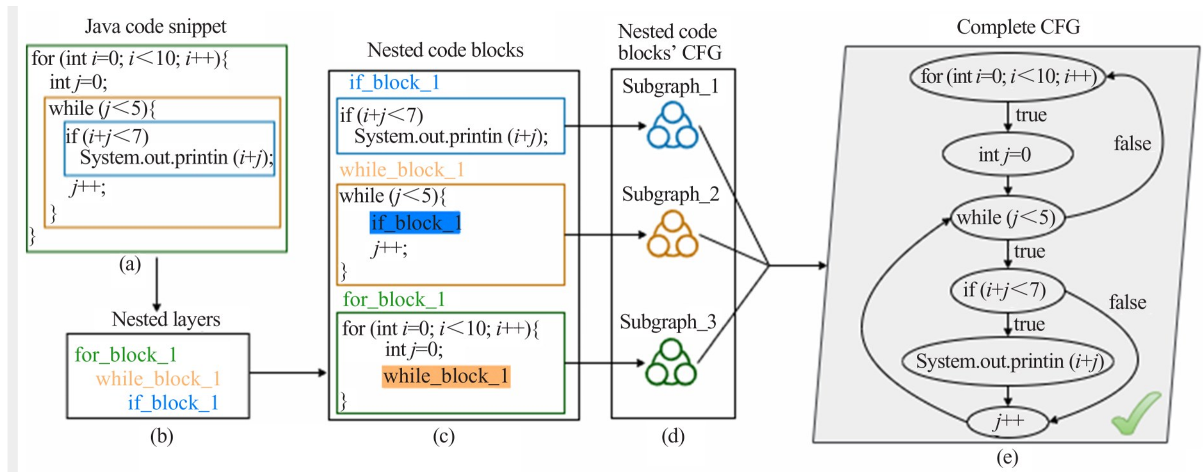

Fig. 2 Running example

tively extracts the innermost nested code blocks based on the nested layers. As shown in Fig. 2(b), the process begins by extracting the code block corresponding to the innermost code layer (" if_block_1") from the Java code. The extracted code block is highlighted with blue border in Fig. 2(c). After that, the block is replaced in the original Java code with a placeholder, creating a modified version where the extracted portion is replaced. The nested code block extraction unit subsequently processes this masked code to obtain the code block of the next nested layer ("while_block_1"). The output is indicated by an orange border in Fig. 2(c). This extraction-and-masking procedure repeats recursively until the outermost code block ("for {⋯}", highlighted with green in Fig. 2(c)) is extracted.

After all nested code blocks are extracted, they are inputted into the CFG generation of nested code block unit to generate their CFGs. In this example, the nested code block "if", "while", and "for" are converted to their respective CFGs as shown in Fig. 2(d)). This process is executed in parallel, using multiple CFG generation of nested code block units.

Finally, the graph fusion unit combines these individual CFGs into a complete CFG for the original Java code, as shown in Fig. 2(e).

1.3 Prompt Design for AI-Units

In this section, we discuss how to design natural language prompts that guide the LLM to perform specific AI functionalities. According to Ref. [20], task description and examples are crucial for prompt design. To make our prompts more consistent, we design a generic template that includes a task description and a set of input-output examples.

1.3.1 Structure hierarchy extraction unit

This AI unit extracts the nested layers from the input code. The prompt is shown in Fig. 3(a). It starts with a task description of "You are a professional ⋯", followed by five examples and a space to input the code to be analyzed. To help the LLM recognize different code layers, the task description also includes five common types of code blocks: method declarations, for loop declarations, if statements, while loop declarations, and switch statements. In addition, Ref. [20] shows that diverse examples can improve the performance of LLMs. Therefore, we ensure that each type of code block appears at least once in the selected five examples. For example, we ensure that among the five examples provided, at least one of them possesses the characteristic of a switch structure. By this way, if the input code includes a switch, the LLM can use the example to correctly extract the matching "switch_block".

1.3.2 Nested code block extraction unit

This AI unit is responsible for extracting the code blocks according to the code layers extracted by the first AI unit. Figure 3(b) shows the prompt, which includes the task description "Please extract the inner block of the given Java code ⋯" along with five corresponding examples. Each demonstration example consists of two inputs: the original code and its corresponding code layer, and one respective outputs: the nested code block.

1.3.3 CFG generation of nested code blocks unit

This AI unit is designed to generate the nodes and edges of all nested code blocks' CFG. Figure 3(c) shows the prompt content of this unit, which includes a task description, "Please convert the following code snippet to ⋯," and five examples. Each example in the prompt in-

cludes a code block input and the corresponding CFG output. When a nested code block is input into this unit, it outputs the corresponding CFG. To provide more effective prompts, we design a simple example retrieval

Structure Hierarchy Extraction

You are a professional programmer, Please analyze and format the following Java code snippet to identify the nested code blocks and indent them to reflect the code block structure. You can only use the following blocks to identify different code blcoks. 1.method_block; 2. for_block; 3. if_block; 4. while_block; 5. switch_block *Task description*

```
## Java code snippet:
public static void main(String[] args) {
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) {
            System.out.println(i);
        }
    }
}
## Code layer:
method_block_1
    for_block_1
        if_block_1
```

Demonstration example-1
Demonstration example-2
Demonstration example-5

```
## Java code snippet:
    {{Java code snippet}}
## Code layer:
```

Query

(a)

Nested Code Block Extraction

Please extract the inner blocks of the given Java code snippet according to the code structure. Ensure that the extracted code blocks are derived from the given code snippet and do not include any code that the original snippet does not contain. *Task description*

```
## Java code snippet:
public static void main(String[] args) {
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) {
            System.out.println(i);
        }
    }
}
## Code layer:
method_block_1
    for_block_1
        if_block_1
## Nested code blocks
## Block_1 -> if_block_1
    if ( i%2 == 0) {
        System.out.println(i);
    }
```

Demonstration example-1
Demonstration example-2
Demonstration example-5

```
## Java code snippet:
    {{Java code snippet}}
## Code layer:
    {{code layer}}
## Nested code blocks:
```

Query

(b)

CFG Generation of Nested Code Blocks

Please Convert the following code snippet to a control flow graph and respresented by python code. *Task description*

```
## Nested code block:
if ( i%2 == 0) {
    System.out.println(i);
}
## Control flow graph:
dot.node('block1_node1', 'if(i%2 == 0)')
dot.node('block1_node2', 'System.out.println(i)')
dot.edge('block1_node1','block1_node2', label='true')
```

Demonstration example-1
Demonstration example-2
Demonstration example-5

```
## Nested code block:
    {{Nested code block}}
## Control flow graph:
```

Query

(c)

Graph Fusion

Please create complete control flow graph of the given Java code snippet by fusion all the subgraphs. Ensure that, the final graph does not contain any placeholder symbols such as 'BLOCK_x' *Task description*

```
## Java code snippet:
public static void main(String[] args) {
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) {
            System.out.println(i);
        }
    }
}
## subgraph_1:
dot.node('block1_node1', 'if(i%2 == 0)')
dot.node('block1_node2', 'System.out.println(i)')
dot.edge('block1_node1','block1_node2', label='true')
## subgraph_2:
dot.node('block2_node1', 'for(int i = 1; i <= 10; i++)')
dot.node('block2_node2', 'block_1')
dot.edge('block2_node1', 'block1_node2', label='true')
dot.edge('block2_node2', 'block2_node1')
## complete control flow graph:
with dot.subgraph(name='cluster_main') as main:
    main.node('block2_node1','for(int i = 1;i <= 10;i++)')
    main.node('block1_node1','if(i%2 == 0)')
    main.edge('block2_node1','block1_node1',label='true')
    main.node('block1_node2','System.out.println(i)')
    main.edge('block1_node1','block1_node2')
    main.edge('block1_node2','block2_node1')
    main.attr(label = 'main')
```

Demonstration example-1
Demonstration example-2
Demonstration example-3

```
## Java code snippet:
    {{Java code snippet}}
## subgraph_1:
    {{subgraph_1}}
...
## subgraph_n:
    {{subgraph_n}}
## complete control flow graph:
```

Query

(d)

Fig.3 The prompts used in each AI units

strategy. Specifically, we prepare five examples for each of the six types of nested code blocks (i. e., method_block, while_block, if_block, switch_block, and for_block). These examples constitute our knowledge base. For each nested code block type, we select five matched examples form the knowledge base. In addition, we use the Python-like code of Graphviz to represent the nodes and edges of the generated CFG.

1.3.4 Graph fusion unit

This unit combines the nodes and edges from each nested code block's CFG to create a complete CFG for the original input code. The prompt is shown in Fig. 3 (d). It includes a task description "Please create complete control flow graph of the given ⋯", and three examples. The input for this unit is CFGs for nested code blocks. The output is the complete CFG.

2 Experiment Setup

2.1 Research Question

We evaluate the CFG-Chain to address the following two research questions (RQ).

- RQ1: What is the quality of each AI unit in CFG-Chain?
- RQ2: How does the performance of CFG-chain generalize across different LLMs?

2.2 Data Preparation

According to the official website of OpenAI, the training data for GPT-4 is current up to September 2021. To ensure that the dataset we use has not been seen during training process of LLMs, we use BeautifulSoup to scrape real-world code examples from Stack Overflow. We collect the latest posts under tags like Java and Python and extract the code inside "<code>" HTML tags. Next, we filter and clean the code samples by the following steps: 1) We remove the code samples that do not contain control flow statements (e.g., "if statement", "for statement", etc.). 2) We compress each code sample by removing the unnecessary comments and empty lines. As a result, we obtain 245 samples of Java code and 281 samples of Python code.

2.3 Evaluation Metrics

In RQ1, we use accuracy as the evaluation metric for the first two AI units: structure hierarchy extraction and nested code block extraction. Accuracy is a straightforward binary metric, where value 1 indicates correct output and value 0 denotes incorrect output. For the CFG generation of nested code blocks and graph fusion AI units, we use node coverage and edge coverage as the evaluation metrics because the output of the nested code block generation unit is a CFG with nested code blocks, and the output of graph fusion unit is a complete CFG. In RQ2, we also use node coverage and edge coverage to evaluate the CFG generated from the code. These metrics provide a measure of how accurately the CFG captures the behavior of the code. Node coverage refers to the number of correct nodes in the generated CFG as a proportion of the CFG summary points. Edge coverage refers to the proportion of the correct number of edges in the generated CFG to the total number of edges.

To calculate node coverage and edge coverage, we recruit 4 annotators, and each with over three years of development experience, to act as annotators for drawing CFGs. All the drawn CFGs are used as standard answers. After generating CFGs using our approach, the output CFGs are compared with the standard answers by annotators to determine their correctness. The following two factors are considered when evaluating the correctness of the output CFGs:

- Number of nodes and edges: We compare the number of matching nodes and edges between the generated CFG and the standard CFG. If some nodes or edges are missing or extra, node coverage and edge coverage will decrease, but the CFG is not marked entirely incorrect.
- Node and edge labels: A node must represent the correct program element (e. g., a statement or control structure), and an edge must show the correct control flow. Incorrect labels lead to lower node or edge coverage but do not fully invalidate the CFG.

3 Experimental Result

3.1 Quality of Each AI Unit (RQ1)

3.1.1 Performance of different prompt examples

Before evaluating CFG-Chain, we first investigate how examples in the prompt influence the performance of LLMs. We only run this experiment on the first AI unit, that is, Structural Hierarchy Extraction. Specifically, we divide the prompt examples for this unit into three types: single-structure, multi-structure, and complex nested structure. Single-structure examples contain only one common type of code block, such as a simple "if statement". Multi-structure examples include two or more different types of code blocks, such as a combination of an "if statement" and a "for loop". Complex

nested structure examples show multi-layer nested code blocks, such as a "while loop" nested within an "if statement", which is itself nested inside a "for loop". We conduct this experiment by using the Java dataset collected in Section 2.2.

The experimental results are shown in Table 1. With single-structure examples, the LLM achieves 62% accuracy in identifying code layers. With multi-structure examples, the accuracy rises to 82%. For complex nested structure examples, the LLM achieves 71% accuracy. These results show that while single-structure examples are simple and easy to understand, they are insufficient for handling complex scenarios. Multi-structure examples are better at helping the LLM understand the relationships between different code blocks. However, higher structural complexity reduces the LLM's ability to recognize the code layers accurately. These experiments provide data-driven insights for optimizing prompt examples.

Table 1 LLM accuracy by example type

%

Example type	Accuracy
Single-structure	62
Multi-structure	82
Complex nested structure	71

3.1.2 Performance of each AI unit

Table 2 presents the experimental results of running CFG-Chain on the two programming language datasets. The first AI unit, structure hierarchy extraction, shows consistent performance on two datasets, achieving an accuracy of 82% on Java and 80% on Python. This indicates that structure hierarchy extraction AI unit can effectively extract the code layers and recognize the boundaries of different structures in the input code.

The second AI unit, nested code block extraction, achieves an accuracy of 84% on the Java dataset and 80% on the Python datasets, suggesting that this unit is capable of accurately extracting nested code blocks.

For the third AI unit, nested code block generation, we observe strong node coverage and edge coverage on both datasets. On the Java dataset, we achieve a node coverage of 89% and an edge coverage of 87%. On the Python datasets, the node coverage is 88% and the edge coverage is 84%. The high node and edge coverages are due to our approach's ability to decompose complex multi-layer nested code into several simpler nested code blocks. By simplifying the code hierarchy, the LLM generates more accurate CFGs for these simpler nested code blocks.

Table 2 Performance of each AI unit

%

AI units	Datasets	Accuracy	Node coverage	Edge coverage
Structure hierarchy extraction	Java	82		
	Python	80		
Nested code block extraction	Java	84		
	Python	80		
CFG generation	Java		89	87
	Python		88	84
Graph fusion	Java		86	82
	Python		87	80

As for the last AI unit, graph fusion, we observe strong node coverage on both datasets, with a value of 86% on Java and 87% on Python. However, there is a slight decrease in edge coverage, with a value of 82% on Java and 80% on Python. This is because in a CFG, an error in a node affects all edges connected to it, while an error in an edge does not affect the related nodes.

3.2 Performance of CFG-Chain (RQ2)

As shown in Table 3, our approach exhibits strong performance across four kinds of LLMs. Among these LLMs, GPT-4 achieves the highest performance on both Java and Python datasets. On the Java dataset, its node coverage is 86% and edge coverage is 82%, respectively. On the Python dataset, its node coverage is 87% and edge coverage is 80%. The notable performance of GPT-4 can be largely attributed to its extensive parameter count of 1.8 trillion. Within the Code LLaMa series, Code LLaMa-7B shows the weakest performance, but the performance improves as the number of parameters increases.

The performance difference among the LLMs can be attributed to their model architectures and training data characteristics. GPT-4, with its 1.8 trillion parameters and advanced attention mechanisms, excels in capturing long-range dependencies and complex control flows, which is crucial for CFG generation. GPT-4 is trained on a large and diverse dataset that includes both natural language and programming code, which helps it achieve good generalization across different program-

Table 3 Performance of CFG-Chain on different LLMs %

LLMs	Datasets	Node coverage	Edge coverage
GPT-4	Java	86	82
	Python	87	80
Code LLaMa-7B	Java	63	57
	Python	65	57
Code LLaMa-13B	Java	67	61
	Python	65	55
Code LLaMa-34B	Java	73	66
	Python	76	69

ming languages and syntax styles. In contrast, Code LLaMa models are trained specifically on code-related datasets and are designed for code understanding tasks. However, their smaller parameter sizes (7 B to 34 B) limit their capacity to process deeply nested code structures. Additionally, the narrower focus of their training data may lead to lower robustness when dealing with uncommon or language-specific syntax. This results in lower node and edge coverage compared to GPT-4.

4 Discussion

4.1 Internal Threats

One internal threat is the accumulation of errors within the AI chain. If an AI unit generates inaccurate results, these errors can propagate through subsequent units, and lead to an incorrect CFG. We incorporate a scoring and optimization mechanism into our CFG-Chain in the future. This mechanism aims to score and optimize the output of each AI unit, minimizing the potential for error propagation and enhancing the effectiveness of our approach.

4.2 External Threats

One external threat is the instability of the outputs generated by the LLM. We set the temperature parameter in the LLM configuration to zero, which improves the stability of our approach. In addition, Ref. [20] suggested that structured prompts contribute to stable LLM outputs. Based on this insight, we transform the natural language prompts in CFG-Chain into structured prompts in the future to further ensure the output stability. At last, our CFG-Chain is a general approach. When superior new models emerge, only the base model version needs to be updated. Another external threat lies in the limited number of programming languages used in our experiments. In this paper, we only evaluate CFG-Chain on Java and Python. These languages were chosen because they have distinct grammatical systems (static and dynamic types) and wide-ranging applications. In addition, our CFG-Chain is not restricted by the syatex or rules of any specific programming language. It uses natural language and large language models to understand and interpret the code in a general way.

5 Conclusion

In this study, we propose a novel approach to generate CFG for code in different programming languages using LLM-based natural language processing. Our approach involves a CoT with four steps, namely structure hierarchy extraction, nested code block extraction, CFG generation of nested code blocks, and graph fusion. We design this CoT as an AI chain based on the single responsibility principle and support it with well-crafted prompt instructions. This design leads to strong performance in terms of node and edge coverage. Our approach provides a new LLM-based alternative solution for the development of software engineering tools that require generally significant engineering and maintenance effort. In the future, we will apply the method to less common programming languages, such as Kotlin, Ruby, etc.

References

[1] Shivers O. Control flow analysis in scheme[C]//*Proceedings of the ACM SIGPLAN* 1988 *Conference on Programming Language Design and Implementation*. New York: ACM, 1988: 164-174.

[2] Chen L, Ye W, Zhang S K. Capturing source code semantics via tree-based convolution over API-enhanced AST[C]//*Proceedings of the* 16*th ACM International Conference on Computing Frontiers*. New York: ACM, 2019: 174-182.

[3] Guo D, Ren S, Lu S, *et al*. GraphCodeBERT: Pre-training code representations with data flow[EB/OL]. [2021-09-13]. https://arxiv.org/pdf/2009.08366.

[4] Hu X, Li G, Xia X, *et al*. Deep code comment generation [C]//*Proceedings of the* 26*th Conference on Program Comprehension*. New York: ACM, 2018: 200-210.

[5] Wang W H, Li G, Ma B, *et al*. Detecting code clones with graph neural network and flow-augmented abstract syntax

tree[C]//2020 *IEEE* 27*th International Conference on Software Analysis*, *Evolution and Reengineering* (*SANER*). New York: IEEE, 2020: 261-271.

[6] Wang W H, Li G, Shen S J, *et al*. Modular tree network for source code representation learning[J]. *ACM Transactions on Software Engineering and Methodology* (*TOSEM*), 2020, **29**(4): 1-23.

[7] Zhang J, Wang X, Zhang H Y, *et al*. A novel neural source code representation based on abstract syntax tree[C]//2019 *IEEE/ACM* 41*st International Conference on Software Engineering* (*ICSE*). New York: IEEE/ACM, 2019: 783-794.

[8] Vallée-Rai R, Co P, Gagnon E, *et al*. Soot: A Java bytecode optimization framework[C]//*CASCON First Decade High Impact Papers*. New York: ACM, 2010: 214-224.

[9] IBM. WALA —Static analysis framework for Java [EB/OL]. [2018-04-19]. http://wala.sourceforge.net/.

[10] Pawlak R, Monperrus M, Petitprez N, *et al*. SPOON: A library for implementing analyses and transformations of Java source code[J]. *Software*: *Practice and Experience*, 2016, **46** (9): 1155-1179.

[11] Jiang X, Zheng Z, Lyu C, *et al*. Treebert: A tree-based pre-trained model for programming language[C]//*Uncertainty in Artificial Intelligence*. New York: PMLR, 2021: 54-63.

[12] Ellis K. Human-like few-shot learning via Bayesian reasoning over natural language[EB/OL]. [2023-09-29]. https://arxiv.org/pdf/2306.02797.

[13] Wang X, Wei J, Schuurmans D, *et al*. Self-consistency improves chain of thought reasoning in language models[EB/OL]. [2022-04-27]. https://arxivorg/pdf/2203.11171.

[14] Webson A, Pavlick E. Do prompt-based models really understand the meaning of their prompts?[EB/OL]. [2021-09-21]. https://arxivorg/pdf/2109.01247.

[15] Huang Q, Sun Y B, Xing Z C, *et al*. Let's discover more API relations: A large language model-based AI chain for unsupervised API relation inference[J]. *ACM Transactions on Software Engineering and Methodology.* New York: ACM, 2024, **33**(8): 1-34.

[16] Ji Z, Yu T, Xu Y, *et al*. Towards mitigating LLM hallucination via self reflection[C]//*Findings of the Association for Computational Linguistics*: *EMNLP* 2023. New York: ACM, 2023: 1827-1843.

[17] Wu T S, Terry M, Cai C J. AI chains: Transparent and controllable human-AI interaction by chaining large language model prompts[C]//*Proceedings of the* 2022 *CHI Conference on Human Factors in Computing Systems*. New York: ACM, 2022: 1-22.

[18] Wu T, Jiang E, Donsbach A, *et al*. Promptchainer: Chaining large language model prompts through visual programming [C]//*CHI Conference on Human Factors in Computing Systems Extended Abstracts*. New York: ACM, 2022: 1-10.

[19] Achiam J, Adler S, Agarwal S, *et al*. GPT-4 technical report [EB/OL]. [2023-04-26]. https://arxivorg/pdf/2303.08774.

[20] Liu J, Shen D, Zhang Y, *et al*. What makes good in-context examples for GPT-3? [EB/OL]. [2021-06-12]. https://arxivorg/pdf/2101.06804.

面向多种编程语言的人工智能链驱动的控制流图生成

邹舟[1,2,3], 左正康[1,2,3†], 黄箐[1,2,3]

1.江西师范大学 国家网络化支撑软件国际科技合作基地, 江西 南昌 330022

2. 江西师范大学 高性能计算江西省重点实验室, 江西 南昌 330022

3. 江西师范大学 语言智能中心, 江西 南昌 330022

摘要：控制流图是程序分析的基本结构，是理解软件内部执行和数据流的基础。由于Java和Python等编程语言的复杂语法，传统的CFG构建方法，如基于字节码分析和基于抽象语法树（AST）的方法，往往面临着泛化能力低和学习成本高等局限性。为了解决这些问题，本文提出了一种基于大型语言模型（LLM）和系统化思维链（CoT）的CFG生成方法。该方法直接从自然语言中解释代码语义，通过遵循单一职责原则的模块化人工智能链策略，将CFG的生成分解为多个独立的、可管理的步骤，不同的步骤对应由单独的人工智能或非人工智能单元进行处理。该方法简化了CFG的构造过程，并增强了方法的泛化性。在Stack Overflow上爬取的245段Java代码和281段Python代码上进行的实验结果表明，该方法在不同的编程语言上均取得了高效的性能，且具有良好的鲁棒性。

关键词：控制流图；大语言模型；思维链；人工智能链

□