Computing and Informatics, Vol. 44, 2025, 1144–1177, doi: 10.31577/cai_2025_5_1144

ENHANCING LARGE-SCALE CODE UNDERSTANDING THROUGH GOAL STRUCTURING NOTATION AND LARGE LANGUAGE MODELS

Zezhong Chen

Shanghai Key Laboratory of Trustworthy Computing
East China Normal University, Shanghai, 200062, China
e-mail: 52215902018@stu.ecnu.edu.cn

Yuxin Deng*

Shanghai Key Laboratory of Trustworthy Computing
East China Normal University, Shanghai, 200062, China
&
MoE Key Laboratory of Interdisciplinary Research of Computation and Economics
Shanghai University of Finance and Economics
Shanghai, 200433, China
e-mail: yxdeng@msg.sufe.edu.cn

Wenjie Du

Shanghai Normal University
Shanghai, 200234, China
e-mail: wenjiedu@shnu.edu.cn

Abstract. Large language models (LLMs) aid programmers in understanding code but are limited by input length when handling large codebases. To address this, we propose using Goal Structuring Notation (GSN) – originally developed for articulating assurance cases in complex engineering projects – to represent and break down large codebases. We introduce a tool that leverages LLMs to automatically con-

* Corresponding author

vert large code into GSN. The generated GSN provides an overview that simplifies code comprehension and enhances communication among programmers. Experimental results demonstrate that our approach significantly increases programmers' confidence levels and reduces task completion times.

Keywords: Goal structuring notation, large language models, software maintenance, code comprehension

1 INTRODUCTION

Maintaining large codebases is a critical yet challenging task in contemporary software engineering. The inherent complexity of growing codebases, coupled with inadequate or outdated documentation, makes understanding and modifying code difficult and time-consuming, especially for new team members. Communication barriers in large or cross-functional teams further complicate maintenance efforts. Addressing these challenges necessitates technological innovations and improved methodologies to enhance software comprehension efficiency and effectiveness.

To tackle these issues, we explore the use of Goal Structuring Notation (GSN) [1, 2] and large language models (LLMs) [3] as promising solutions. GSN offers a structured and visual approach to representing complex arguments and justifications [4, 5]. Applying GSN to software maintenance can transform cryptic and convoluted codebases into clear, understandable diagrams that elucidate goals, strategies, and relationships within the code. This graphical representation bridges gaps caused by inadequate documentation by providing a high-level overview of the software's architecture and logic. Moreover, GSN's ability to encapsulate assumptions and justifications makes implicit knowledge explicit, enhancing understanding and communication among team members.

LLMs, with their advanced natural language processing capabilities, serve as critical tools for deciphering complex code structures [6]. They can analyze and interpret code, translating technical jargon and intricate programming concepts into accessible language. This translation is crucial for team members unfamiliar with specific aspects of the codebase or for newcomers acclimating to the project. Furthermore, LLMs can assist in generating up-to-date documentation, alleviating the burden on maintenance teams and ensuring that documentation evolves alongside the code. However, the limited input length of LLMs poses challenges for processing large codebases that exceed these restrictions.

To address this limitation, we propose breaking down large codebases into manageable parts using GSN. By combining GSN's structured visual framework with LLMs' nuanced language understanding, we present a synergistic solution.

Our approach systematically translates code into GSN elements – such as goals, sub-goals, strategies, and evidence – providing a structured visual representation of software logic. This process involves generating function call graphs, filtering nodes,

and using LLMs to create natural language descriptions of code functionality. This enables stakeholders to efficiently comprehend and manage large codebases. Experimental results demonstrate the efficacy of this methodology, showing significant improvements in programmers' confidence and reduced task completion times.

By integrating GSN and LLMs, we offer a comprehensive approach to tackling challenges associated with large codebases – including complexity, inadequate documentation, and communication barriers – paving the way for more efficient and effective software understanding and maintenance practices.

The key contributions of the current work are as follows.

1. **Innovative Method for Code Understanding**: We introduce the use of GSN in combination with LLMs to enhance the understanding and maintenance of large software codebases.
2. **Implementation of an Automated GSN Generation Tool**: We develop a tool that automatically transforms existing code into a GSN framework.
3. **Validation through Real-World Case Studies**: We validate the practicality and effectiveness of the proposed method through case studies involving real-world code repositories.

These contributions demonstrate the potential of combining GSN with LLMs in the realm of code maintenance and understanding, offering new tools and directions for research and practice in software engineering.

The structure of the paper is organized as follows. In Section 2, we review existing literature in the field of code understanding and maintenance. In Section 3, we detail the process of transforming code into the GSN framework. In Section 4 introduces the relevant technical implementations, including the generation of function call graphs and the integration of large language models. Section 5 provides a case study to help readers understand the process of transforming code into GSN. Section 6 presents the practical application of our methodology. In Section 7, we discuss the implications, advantages, and potential limitations of our method. In Section 8, we summarize our main findings and contributions.

Our tool Trusta is available at `https://github.com/AssuranceCase/Trusta`.

2 RELATED WORK

The field of code understanding and maintenance has witnessed considerable research [7, 8, 9], with various methodologies being proposed and evaluated over the years.

Visualization tools have played a significant role in aiding code comprehension. Research shows that visual representations of code structures and dependencies can significantly reduce the cognitive load on developers and improve their understanding of complex software systems. Tools like SoftVis3D [10], Callcluster [11], Code Park [12], 3D-Flythrough [13], Primitive [14] and AppMap [15] are notable

examples, offering visual metaphors to depict software architectures and relationships.

Another related domain is the application of natural language processing (NLP) techniques to interpret and document code [16]. Research in this area has focused on automatically generating documentation from source code using various NLP methods. Notably, recent advancements in large language models, such as ChatGPT-4 [17] and PaLM 2 [18], have been leveraged to translate complex code structures into more understandable narratives [6], aiding in both comprehension and documentation. Current state-of-the-art code comprehension tools integrating LLMs, such as GILT [6], can only support interactive translation of small code snippets. Our tool is specifically designed and developed for understanding large codebases.

In the specific context of using structured frameworks like GSN, there has been limited but noteworthy exploration. For instance, some studies have investigated the use of GSN for representing software design and architecture [19, 20], particularly in safety-critical systems. These studies highlight the effectiveness of GSN in clarifying design rationales and decision-making processes in software development.

However, the integration of GSN with large language models for the purpose of enhancing code comprehension and maintenance, as proposed in our study, represents a novel approach in this field. This combination harnesses the strengths of both visual structuring and advanced language understanding, potentially setting a new direction for research and practice in software maintenance. Unlike existing tools that visualize the entirety of code to aid programmers in understanding, our methodology introduces several innovative aspects:

- By dynamically running specific functionalities of the program and extracting the corresponding code, we present a more targeted display of the aspects users are concerned with.
- We utilize large language models to translate the corresponding code into natural language, facilitating comprehension among diverse stakeholders.

These enhancements not only differentiate our work from existing methods but also offer a comprehensive approach to addressing the complexities of code maintenance and comprehension.

In summary, while there have been several advancements in the realms of code visualization, NLP-based documentation, and dynamic software analysis, our approach uniquely combines GSN with large language models, offering a fresh perspective on tackling the enduring challenges of large software maintenance and comprehension.

3 METHODOLOGY

In this section, we introduce the key ingredients of our methodology, including the definition of GSN elements and the code to GSN transformation. Our approach

leverages the theoretical foundations of GSN and the advanced code comprehension capabilities of LLMs.

3.1 Definition of GSN Elements

GSN, a graphical argumentation notation, is widely adopted for its ability to represent complex arguments and assurance cases in a structured and comprehensible manner [21, 19]. In software engineering, GSN offers a high-level abstraction of software architecture and logic, enhancing understanding and communication among stakeholders. The core elements of GSN include:

Top-Level Goals: In our setting, these represent the primary objectives or the main purposes of the code. They encapsulate the high-level functionality or the key outcomes that the software is designed to achieve.

Sub-Goals: Derived from the top-level goals, sub-goals are more specific objectives that need to be met to accomplish the top-level goals. In software, these typically relate to specific functionalities or features of individual components.

Strategies: In GSN, strategies denote the approaches or methodologies adopted to achieve the goals. In software terms, this would translate to the specific algorithms or methodologies employed in the code to realize certain functions or features.

Solutions or Evidences: These elements are crucial for providing proof or validation that the goals or sub-goals have been met. In software, evidences might come in the form of test results, validation checks, or other forms of empirical verification.

Context: Context elements provide the necessary background information to understand the goals, such as environmental constraints, data definitions, or operational settings. In software, this often includes class constructors or initialization functions that set up the necessary environment for the code to function.

Assumptions: Assumptions in GSN are statements that are considered true within the argumentation framework. In software, assumptions might include premises about user inputs, system environments, or dependencies.

Justifications: Justifications offer the rationale behind choosing a particular strategy or making certain assumptions. In the context of software, this could involve explaining the design philosophy behind implementing a specific feature or the reasoning behind certain coding decisions.

Figure 1 gives an example GSN diagram. It visually demonstrates our methodology by decomposing software engineering goals into manageable components, complete with defined strategies, assumptions, and justifications, thus improving the clarity and maintainability of complex software systems. From the perspective of information representation and readability, compared to the one-dimensional linear reading of code, GSN allows for the two-dimensional display of information both horizontally and vertically.

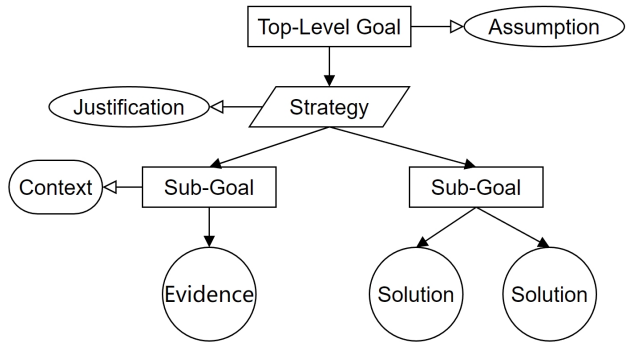

Figure 1. Goal Structuring Notation (GSN) framework diagram

3.2 Code to GSN Transformation Process

The translation of code into GSN is a systematic process that aids in the elucidation of software logic and architecture. The steps involved in this transformation are designed to distill complex code into a clear, structured format that aligns with GSN principles. Herein we detail the step-by-step methodology shown in Figure 2.

1. **Function Call Graph Generation**: This process commences with the creation of a function call graph. The graph is a visual representation of the software's functional structure, where nodes correspond to function names and edges depict the call relationships between these functions.
2. **Function Node Filtration**: Following graph generation, we proceed to prune the function call graph. This involves the removal of nodes that represent functions not pertinent to the software's core functionality, thus simplifying the graph and focusing on the most relevant aspects of the code.
3. **Natural Language Transformation Using LLMs**: Large language models are then employed to read and analyze the function nodes along with their corresponding bodies of code. These models generate a natural language description of each function's purpose and deduce the strategies that connect each sub-goal with its immediate higher-level goal. This step leverages the language model's capability to understand and articulate code in human-readable terms.
4. **Contextualization of Constructors and Initializers**: Constructors and initialization functions within the code are translated into context nodes within the GSN framework. These nodes provide critical background information for the understanding of the software's operations and goals.
5. **Integration of Test Cases as Evidence Nodes**: Test cases associated with the code are integrated into the GSN as evidence nodes. These nodes serve to validate the achievement of the respective sub-goals they are connected to, offering empirical proof of functionality.

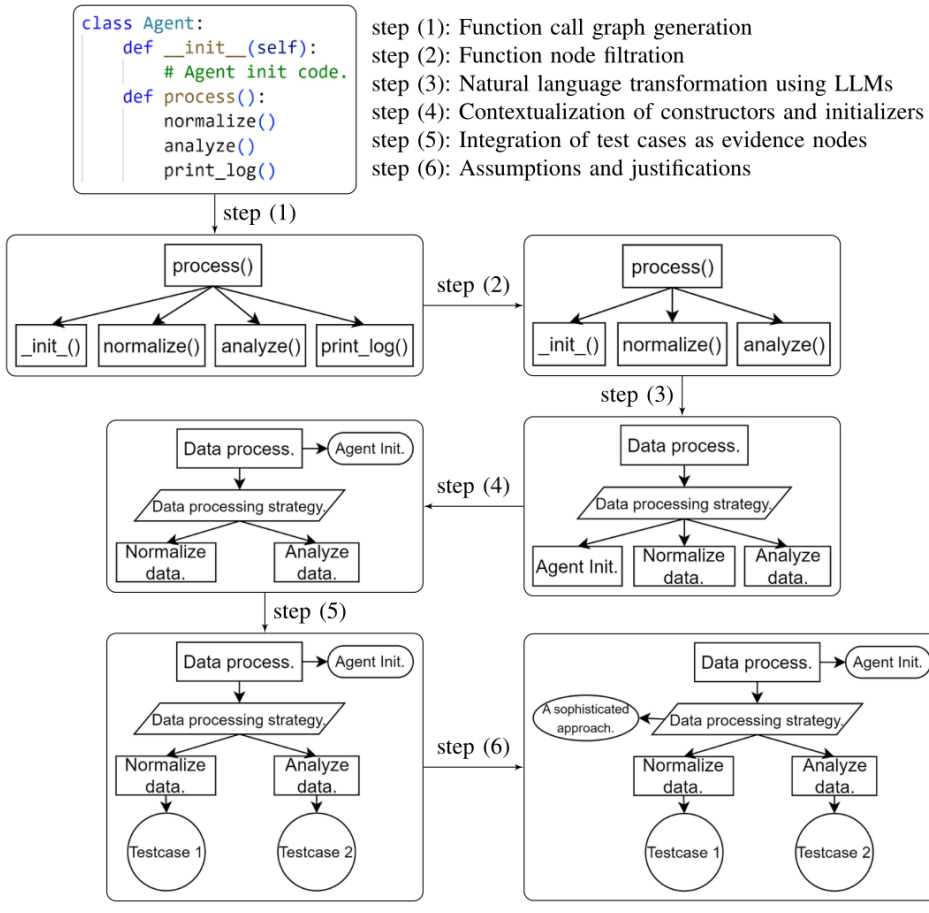

Figure 2. Flowchart of the code to GSN transformation process

6. **Assumptions and Justifications**: The final step involves the addition of assumptions and justifications where necessary for expressiveness, need, and logical completeness. Assumptions are added to the GSN to represent premises considered true within the software's operational context, while justifications provide the rationale for the strategies and design decisions made.

4 TECHNICAL IMPLEMENTATION

This section provides a detailed introduction to the techniques involved in the six steps of transforming code into GSN.

Steps (2), (4), and (5) are straightforward and can be assisted by program static analysis [22] with some manual additions. Step (2) can define non-logical

function naming rules, such as "print*", as a basis for simplifying the function call graph. Step (4) uses program features and project conventions to extract class constructors and initialization functions via static analysis, converting these nodes from sub-goal to context type in GSN. Step (5) extracts functions called within test cases using static analysis. If some test cases cover a particular sub-goal, these test cases are added as evidence nodes for that sub-goal. Step (6) involves subtler information often not explicitly expressed in code (e.g., might exist in comments and documentations), requiring manual completion considering project stakeholders.

Steps (1) and (3) are more challenging, requiring more advanced techniques such as dynamic code analysis [23] and large language models. Below we explain steps (1) and (3) in more detail.

4.1 Usage of Function Call Graph Tools

Step (1) involves creating function call graphs, which can be either dynamic or static [24]. Our study specifically refers to dynamic call graphs. A dynamic call graph can be highly accurate but only represents a single execution instance of the program. On the other hand, a static call graph represents all possible executions of the program.

For large programs, the codebase can be vast, and programmers often only focus on a specific functionality during maintenance. A dynamic call graph, generated during the program's execution, captures the content relevant to that functionality, ignoring the unrelated parts of the code. This significantly reduces the scope of code that needs to be read.

The principle of generating dynamic call graphs involves monitoring the function calls made by a program during its execution. Profilers like callgrind [25], which is part of the Valgrind tool suite, instrument the code to record each function entry and exit. When the program runs, callgrind collects data on the call relationships and the execution path taken, producing a detailed log of the function calls made. This log can then be visualized to create the dynamic call graph.

In this study, we used the pycallgraph library [26] to generate dynamic function call graphs for the Python project and the callgrind tool for the C project. These tools helped visualize the function calls made during the execution of the respective programs, providing a clear map of the code flow for the specific tasks being analyzed.

4.2 Usage of Large Language Models

Listing 1 encapsulates the structured prompt used in Step (3) of our methodology, showcasing how large language models are employed to bridge the gap between raw code and its representation within the GSN framework. The prompt is divided into three main sections to ensure comprehensive function analysis and strategy formulation:

Task Description: This initial section sets the context for the large language model, defining its role as both an assurance case expert and a proficient software engineer. It outlines the model's tasks, which include analyzing the provided code to determine the function's goal and elucidate the strategy behind the use of external functions to achieve this goal.

```
You are an Assurance Case expert as well as a good software engineer.
Your answers always need to follow the following output format and you always have to try to provide the specified information.
I will provide you with code that contains a function. Please analyze the content of the code and complete the following tasks: (1) Describe in one sentence the goal of this function. (2) This function in the code calls several external functions. Along with the names of these external functions, I will provide their corresponding sub-goals. Please explain in one sentence the strategy behind using these external functions, considering their sub-goals, to achieve the overall goal of the function. If the names of these external functions and their sub-goals are not provided, just complete the first task.

Example 1:
Function Code:
'''
def process_data(data):
    cleaned_data = clean_data(data)
    normalized_data = normalize_data(cleaned_data)
    result = analyze_data(normalized_data)
    return result
'''
External Functions:
['clean_data: Removes any invalid or corrupt data.', 'normalize_data: Brings all data to a standard format or range.', 'analyze_data: Performs statistical analysis on the data.']
Task Answers:
(1) Function Goal: Prepare and analyze given data, ultimately returning the analysis results.
(2) One-sentence Strategy: Cleans, Normalizes, and Analyzes the data to ensure accurate and consistent statistical results.

Example 2:
Function Code:
```

```
```
def clean_data(data):
    # Remove null or missing values
    data = data.dropna()
    # Remove duplicate entries
    data = data.drop_duplicates()
    return data
```
External Functions:
[]
Task Answers:
(1) Function Goal: Removes any invalid or corrupt data.
(2) One-sentence Strategy: ''

Tips: Please add the following in strict format, including
 "Function Goal:", "One-sentence Strategy:".

Function Code:
```
<NEW_FUNCTION_CODE>
```
External Functions:
<NEW_EXTERNAL_FUNCTIONS>
Task Answers:
(1) Function Goal:
(2) One-sentence Strategy:
```

Listing 1. Step (3) prompt structure: natural language transformation using LLMs

Input and Output Format with Examples: This section specifies the format for inputs (function code and external functions) and expected outputs (function goal and strategy). It provides clear examples in two different scenarios, demonstrating how the model should interpret the code, identify the goal of the function, and describe the strategic use of external functions based on their sub-goals.

Example Template for Additional Scenarios: The final section offers a template for further examples, with placeholders for new function codes and external functions. This template is designed to be filled with specific code analyses, facilitating the model's task of generating consistent and structured responses.

In the prompt structure used for the LLM, there are two specific placeholders that require substitution with relevant data as input for the model. The first placeholder, marked as ⟨NEW_FUNCTION_CODE⟩, should be replaced with the source code of the function currently under analysis. This represents the actual code of the function node being examined.

The second placeholder, ⟨NEW_EXTERNAL_FUNCTIONS⟩, is to be filled with the goals of all external functions called by the current function, providing a clear context for their inclusion and use within the main function's logic. The output from the LLM will then articulate the primary goal of the current function and the strategy behind employing these external functions to achieve that goal. If the current function does not call any external functions, ⟨NEW_EXTERNAL_FUNCTIONS⟩ placeholder should be left empty, and the model will only output the function's goal.

The input and output of a single invocation of the large language model using this prompt can be represented by the following formula:

$$\begin{cases} \text{Prompt}(\mathit{FunctionCode}, \mathit{ExternalFunctionsGoals}) \\ \qquad\qquad\qquad \rightarrow \{\mathit{FunctionGoal}, \mathit{Strategy}\} \\ \text{Prompt}(\mathit{FunctionCode}, \emptyset) \rightarrow \{\mathit{FunctionGoal}, \emptyset\} \end{cases}$$

where the variables are explained as follows:

- *FunctionCode*: The source code of the current function under analysis.
- *ExternalFunctionsGoals*: A list of goals for each external function called by the main function. This includes the names of the functions and their respective goals, providing the context for their strategic use.
- *FunctionGoal*: A concise statement of the primary objective of the function being analyzed.
- *Strategy*: The rationale for utilizing specific external functions to achieve the main function's goal, outlined only when external functions are present. If *ExternalFunctionsGoals* is empty, indicating no external functions are called, then the strategy component is also empty, highlighting the direct relationship between the function's external dependencies and the formulation of a strategy.

When processing a complex, multi-layered function call tree using the outlined steps, it is imperative to traverse the call hierarchy from the bottom up. This traversal begins with the leaf nodes – functions that do not invoke other functions – and progresses upwards. Such a bottom-up approach ensures that the analysis of higher-level functions can leverage the goals identified for the functions they call, thereby maintaining consistency and continuity in the development of the GSN framework. This methodology allows for a structured decomposition of software logic, making the intricate relationships within the code comprehensible and systematically documented. This also reflects one of GSN's characteristics: "Being able to simply summarize viewpoints": It shows that GSN users were able to summarize the viewpoint [4]. In our method, summarizing the function's role from the bottom up in the function call tree results in fewer high-level goal nodes, with each goal covering a broader scope until the entire functionality is summarized in a single node.

Then, from top to bottom, the number of nodes increases, with each sub-goal covering a narrower scope and revealing more details. Thus, once a complete GSN is generated, stakeholders can examine the code at different levels, rather than searching for information in incomplete documentation or extensive but overly detailed code.

The large language model prompt in Listing 1 offers a structured approach for the language model to analyze functions and their interrelationships, ensuring a consistent and goal-oriented analysis. By following the detailed format for output, the model can succinctly describe the function's goal and the rationale behind the use of each external function in achieving that goal. This forms a crucial component of the GSN, linking code functionality to a structured argumentation framework that enhances clarity and facilitates better maintenance practices.

The structured prompt ensures the generation of precise and coherent function goals and strategies, which are essential for creating accurate and informative GSN diagrams.

5 CASE STUDY

This section demonstrates how to transform the code of a Python-based AI inference system into a GSN. The AI inference system contains approximately 8000 lines of code. Using our Trusta tool, we generated a GSN with 34 nodes representing the inference steps, which took 8 minutes to complete. Through this process, we clearly express the code's goals, logic, and related test cases.

Step 1: Generation of Function Call Graphs. First, we generate the function call graph of the AI inference system, where each node represents a function and edges indicate the call relationships between functions. This helps to reveal the dependencies between the system's modules. The function calls starting from the main function are shown in Figure 3.

Step 2: Filtering of Function Nodes. After generating the function call graph, we filter out the function nodes that are not related to the core functionality of the AI inference system. This simplifies the graph and focuses on the key parts of the inference process. The filtering rules ($FUNCTION\_IGNORE\_KEYWORDS$) are obtained by fuzzy matching function names using configuration files, as shown in Listing 2. In actual projects, this configuration should be organized according to the project's specifics and the purpose of generating the GSN. For example, if only the upper-level business code is of interest, one can filter out low-level library functions unrelated to the business or set a whitelist ($FUNCTION\_NEED\_KEYWORDS$) to include business-related functions. The result of filtering the function call graph in Figure 3 is shown in Figure 4, where the structure is simpler and contains only the desired content.

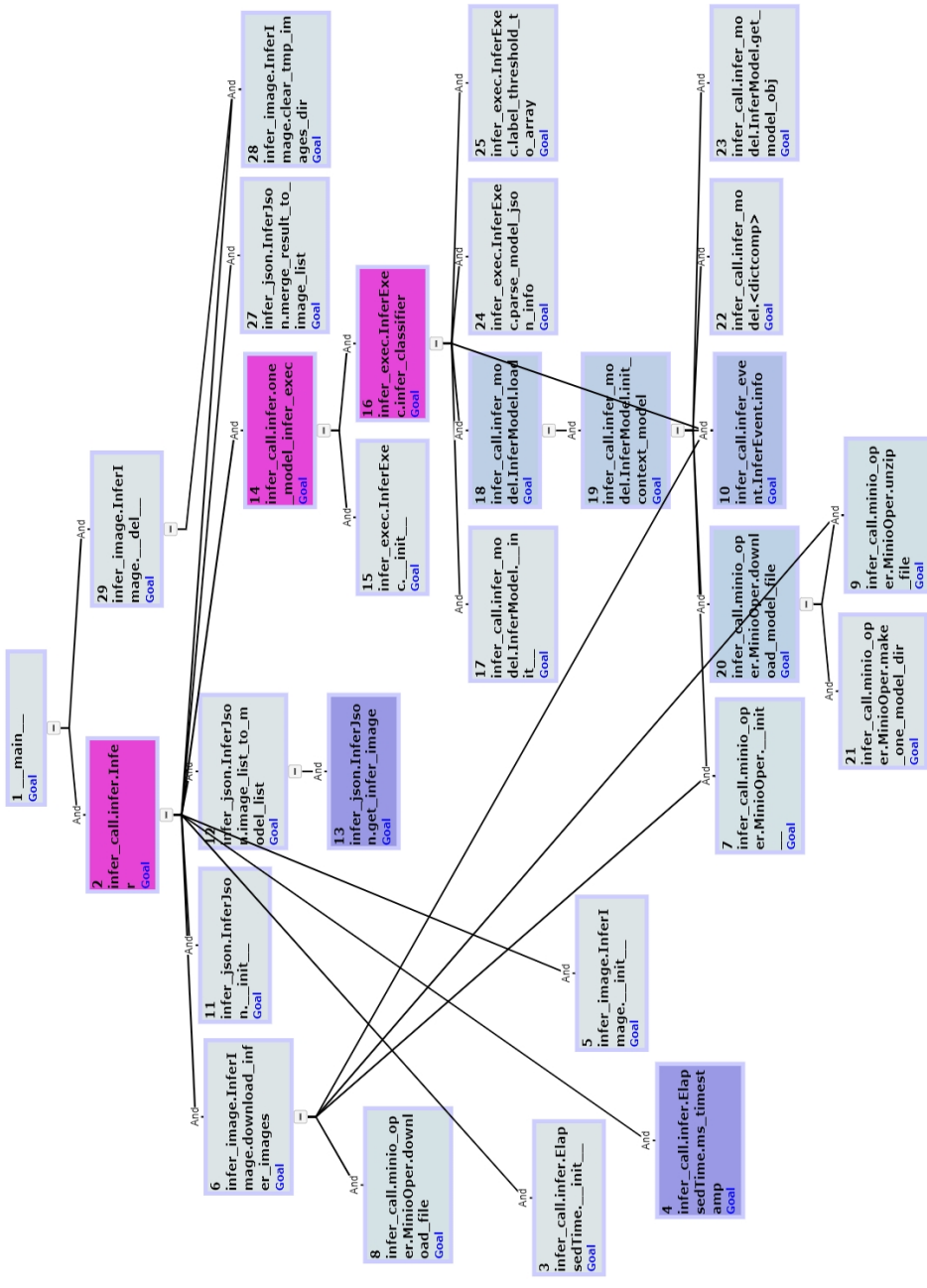

Figure 3. Output of Step 1: Generation of Function Call Graphs

```
{
    "FUNCTION_NEED_KEYWORDS": [
    ],
    "FUNCTION_IGNORE_KEYWORDS": [
        "InferEvent.info", "MinioOper.unzip_file",
        "MinioOper.make_one_model_dir",
        "InferImage.__del__", "ETATimeIter.print_info",
        "ElapsedTime", "__main__",
        "infer_model.<dictcomp>"
    ]
}
```

Listing 2. Configuration file for function filtering

Step 3: Natural Language Transformation using LLM. We use LLM to analyze the function nodes and their code content, generating natural language descriptions for each function, as shown by the green nodes in Figure 5. Additionally, we generate relationship descriptions for the parent and child nodes, i.e., strategies in the GSN, as shown by the blue nodes in Figure 5. This helps us identify sub-goals and link them to higher-level goals.

Step 4: Contextualization of Constructors and Initializers. In this step, we transform constructors and initializer functions in the code into context nodes in the GSN framework. These nodes provide background information for the goals. Some nodes in Figure 5 are transformed into descriptions, becoming context information for their parent nodes, as shown by the white parts in nodes 2, 6, 14, 16, and 19 in Figure 6. This step further simplifies the overall graph structure without losing its original expressiveness.

Step 5: Integration of Test Cases as Evidence Nodes. We integrate test cases into the GSN graph as evidence nodes. Test cases provide verification of whether subgoals have been achieved, ensuring that the inference system functions as expected. By scanning the content of the test cases, if the test case covers nodes (functions) in the GSN, it is attached as evidence to the corresponding node as a child node. The yellow nodes in Figure 7 are the newly added test case nodes. For simplicity, we only retained the main nodes (green goal nodes) in the figure, hiding auxiliary information (blue strategy nodes and white context information).

Step 6: Addition of Assumptions and Justifications. Finally, we can manually add assumption and justification nodes based on the needs of the GSN graph's readers. Assumptions represent the premises for goal nodes, while justifications explain the reasons for choosing certain strategies or design decisions, enhancing the logicality and completeness of the GSN graph.

For another case study of our tool applied to cryptography code, see Appendix A.

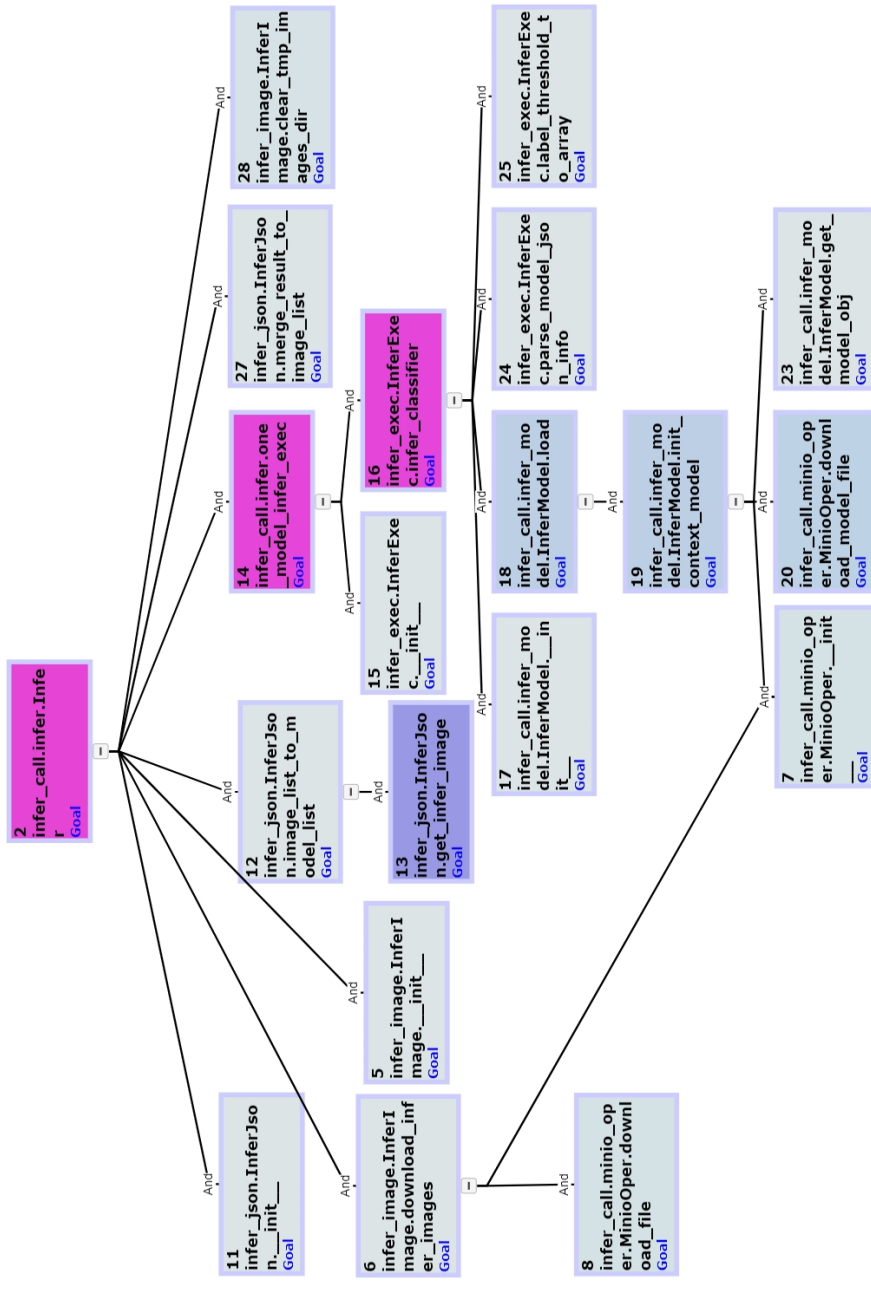

Figure 4. Output of Step 2: Filtering of Function Nodes

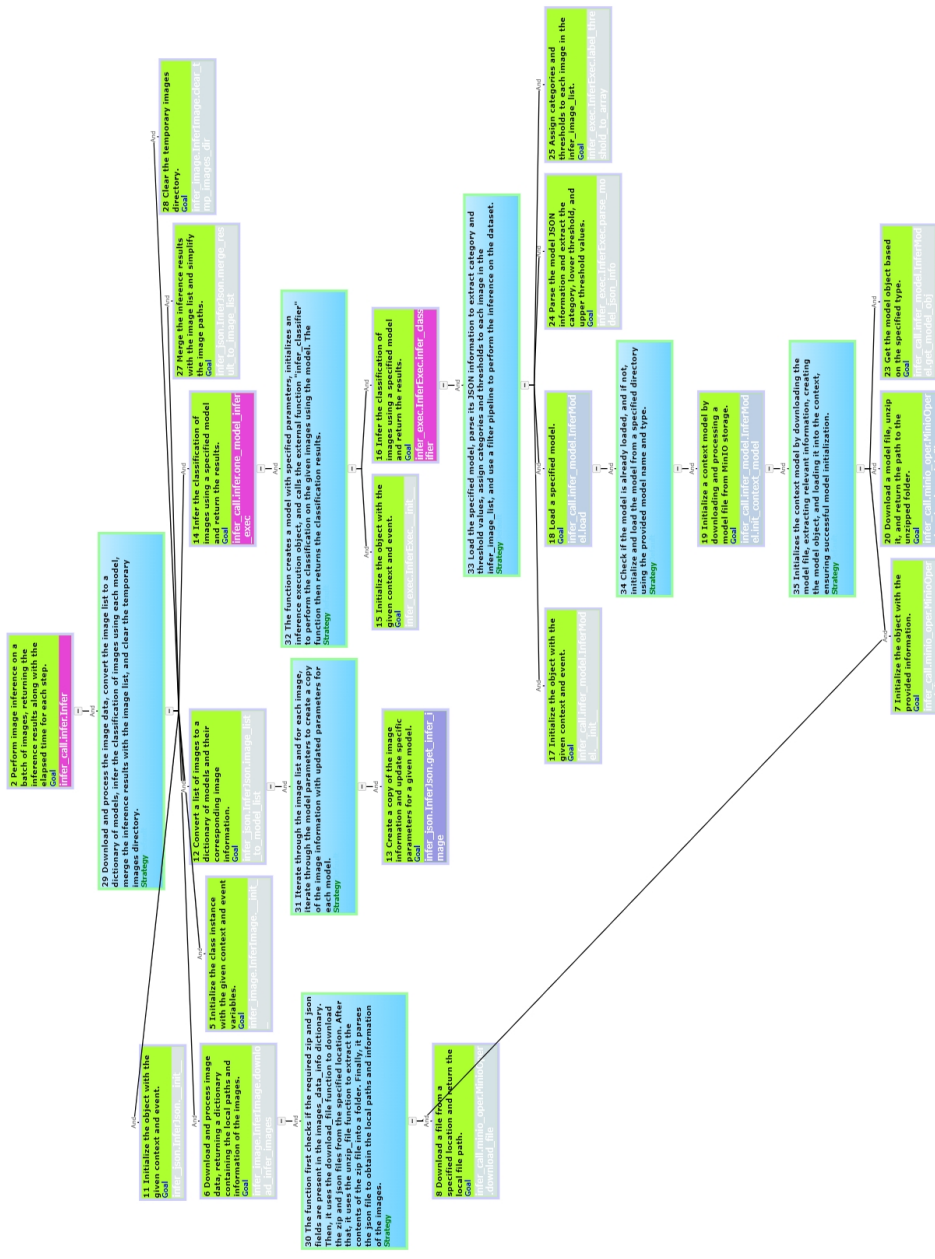

Figure 5. Output of Step 3: Natural Language Transformation

Figure 6. Output of Step 4: Contextualization of Constructors and Initializers

6 EXPERIMENTAL SETUP AND RESULTS

6.1 Experimental Setup

We designed an experiment to evaluate the impact of using GSN generated by LLMs on programmers' understanding and maintenance of code. We selected two projects for the experiments: a Python-based AI inference system running in a production environment, and the QEMU [27] project, a large open-source C language software. QEMU is a generic machine and userspace emulator and virtualizer, with a codebase that spans millions of lines of code. In our experiments, we used QEMU to simulate a SabreLite development board for loading Linux kernels, demonstrating one of its many capabilities. We used our tool (Trusta) and spent 76 minutes generating a dynamic function call graph-based GSN with 212 nodes.

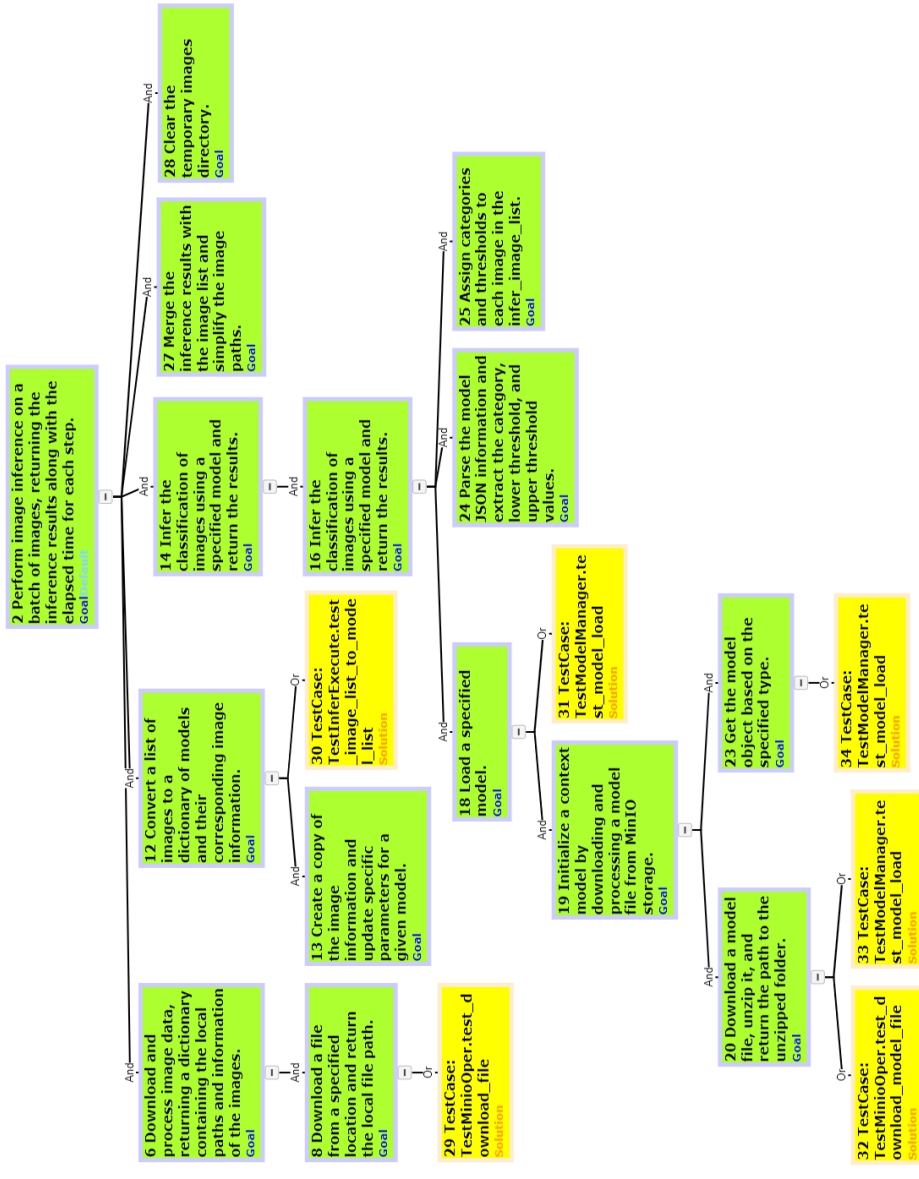

Figure 7. Output of Step 5: Integration of Test Cases as Evidence Nodes

The experiments were divided into six groups, with each group consisting of 10 participants, as shown in Table 1.

Category	**Traditional Method**	**Enhanced Method**
SPF	10 participants	10 participants
SP	10 participants	10 participants
JP	10 participants	10 participants

Note:
SFP = Senior Programmers Familiar with the Project;
SP = Senior Programmers Not Familiar with the Project;
JP = Junior Programmers Not Familiar with the Project.

Table 1. Classification of experiment participants and methods used

Traditional Method: Programmers received only the code documentation, source code, search engines, and LLMs.

Enhanced Method: In addition to the materials provided in the Traditional Method, programmers in this group were also given GSN diagrams generated by large language models.

The participants included programmers with varying experience levels, some of whom were familiar with the projects. Their task was to understand and maintain the code within a limited time frame, which involved fixing a known bug and adding a new feature. Task 1 required debugging QEMU's initialization flow by identifying and removing an erroneous return statement in the *qemu_init_main_loop* function, ensuring that the VM launch process completed successfully. Task 2 involved adding a post-initialization output to the QEMU startup sequence, where participants had to print the string "Trusta" after the initialization, ensuring the message appeared via the debug console or log without interfering with the core emulation logic.

The large language model used in this experiment is Llama3-70b [28], the state-of-the-art open-source model, which was utilized to generate the GSN diagrams. We further developed the Trusta tool (`https://github.com/AssuranceCase/Trusta`) for visualization, as shown in Figure 8. One of the steps in generating the function call graphs involved using the pycallgraph library [26] for the Python project and the callgrind [25] tool for the C project.

To quantify the impact of the method on code understanding and maintenance, we used the following two metrics:

Confidence Level: The confidence level of the programmers in understanding and modifying the code, rated on a Likert scale (1 to 5) [29].

Task Completion Time: The time taken to complete the specified tasks (in minutes).

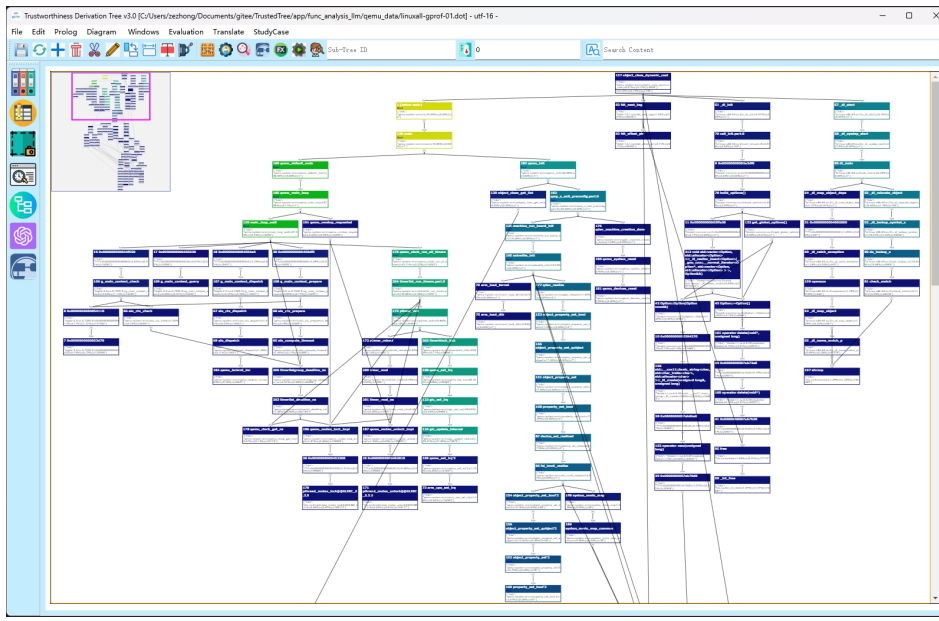

Figure 8. Full overview of the GSN diagram for the QEMU project

6.2 Experimental Results and Analysis

The results showed that programmers in the Enhanced Method group had significantly higher confidence levels and shorter task completion times compared to the Traditional Method group.

Figure 9 presents the successful task completion times for the six experimental groups, excluding the two participants in the JP-Traditional group who did not complete the task within 120 minutes. The tasks involved modifying a single line of code, simplifying the assessment of completion times.

The *Enhanced Method*, which included GSN diagrams generated by LLMs in addition to traditional resources, reduced task completion times across all groups. The effect was more pronounced among programmers unfamiliar with the project: in the **SP** group, the mean completion time decreased by approximately 45 %; in the **JP** group, the reduction was about 58 %; and the **SPF** group saw a modest decrease of around 23 %. These results indicate that GSN diagrams significantly aid programmers – especially those without prior project familiarity – in understanding and maintaining code more efficiently. The Enhanced Method also reduced variability in completion times, as evidenced by narrower interquartile ranges in the boxplots. This suggests more consistent performance when additional structured documentation is provided. Despite the simplicity of the tasks, participant experience and project familiarity influenced performance. SPF participants com-

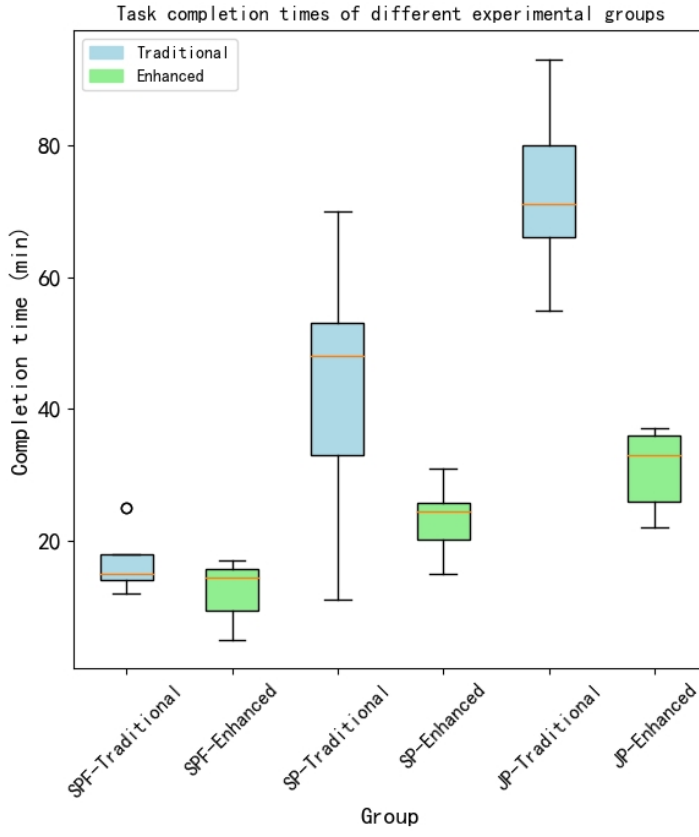

Figure 9. Task completion times across different experimental groups

pleted tasks fastest under both methods, highlighting the advantage of prior knowledge.

After the participants read the tasks, we asked each of them to rate their confidence level in completing the tasks. The average results for each group are shown in Figure 10. Participants' confidence levels varied among the six experimental groups. Compared to the Traditional Method, the Enhanced Method, which added GSN diagrams generated by LLMs to the traditional resources, consistently improved participants' confidence levels. This effect was particularly evident among programmers unfamiliar with the project, indicating that additional structured guidance can significantly enhance confidence.

The data indicate that the programmers in the Enhanced Method group experienced a significant increase in confidence level and a notable decrease in task completion time.

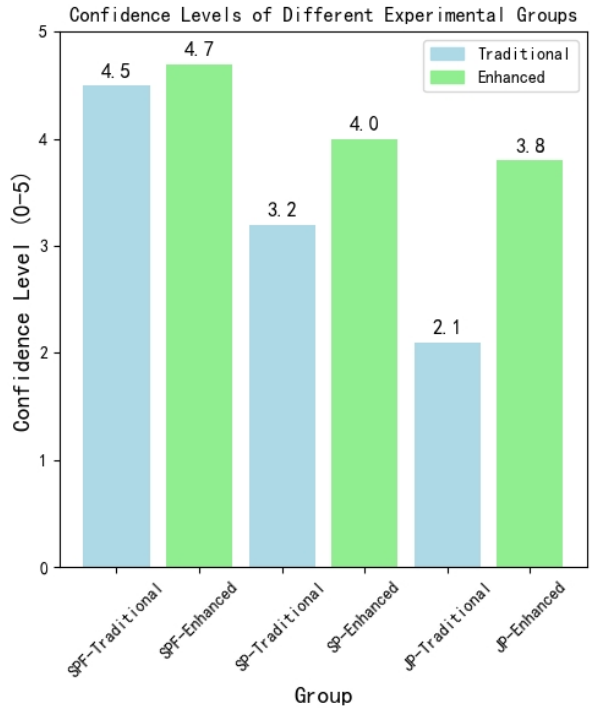

Figure 10. Average confidence levels of participants across experimental groups

The superior performance of the Enhanced Method group in terms of confidence level and task completion time is mainly attributed to the clear presentation of code structure and logic provided by the GSN diagrams and the varying levels of detail in code summaries generated by the LLMs. Specifically:

Confidence Level: The GSN diagrams and the natural language descriptions generated by the LLMs helped programmers better understand the overall architecture and specific implementations of the code, making them more confident in making modifications.

Task Completion Time: The summaries generated by the LLMs reduced the amount of code programmers needed to read, allowing them to understand the code more quickly. Additionally, the GSN diagrams enabled programmers to quickly locate the parts of the code relevant to the task, further reducing the time needed to search the code, thus allowing them to complete tasks more quickly.

We conducted a survey on the participants' satisfaction with the Trusta tool, and 88 % of users reported being satisfied. However, feedback highlighted areas for improvement, including adaptive scaling, GSN diagram aesthetics, and the need

for progress indicators. These insights are crucial for Trusta's future development, emphasizing the importance of user-centric design for the tool.

These results demonstrate that GSN diagrams generated by the Llama3-70b model can significantly enhance programmers' understanding and maintenance efficiency, proving to be highly practical.

6.3 Visualization Results

To illustrate the effectiveness of the GSN diagrams more clearly, we provide screenshots of the GSN diagrams for the QEMU experiment. Figure 8 is an overview of the entire GSN diagram in the Trusta tool.

Figure 11 illustrates the top-level structure of the GSN diagram for the QEMU experiment, along with the corresponding code and function call graph for comparison.

Figure 12 shows a portion of the GSN diagram for the QEMU experiment related to memory management and the corresponding function call graph.

7 DISCUSSION

This section discusses the inherent difficulties we encountered and the implications of these challenges for future research and practice in software engineering.

7.1 Function Call Graph Generation

One of the primary complexities lies in the initial step of generating a function call graph that accurately represents the software's functional structure. This requires a nuanced understanding of the codebase, as well as sophisticated analysis tools capable of parsing and visualizing complex interdependencies among functions. The task is further complicated by dynamically loaded modules and polymorphic behaviors, where the determination of call relationships can be non-trivial and requires advanced static and dynamic analysis techniques, which depend on the capabilities of the compiler or code parser of the target programming language. Recursive functions generate a connection line pointing to the node itself, reflecting the recursive state but failing to represent the details of recursive execution.

Static graphs may include unused paths, while dynamic graphs are context-specific. Both static and dynamic graphs can generate GSN. Each has its pros and cons: static graphs are more comprehensive but can be overly large and hard to read, with lower accuracy and efficiency. Conversely, in large systems where only certain functionalities need maintenance, more targeted dynamic graphs are more suitable. The downside of dynamic graphs is that they require the system to be up and running, which increases the difficulty of generation.

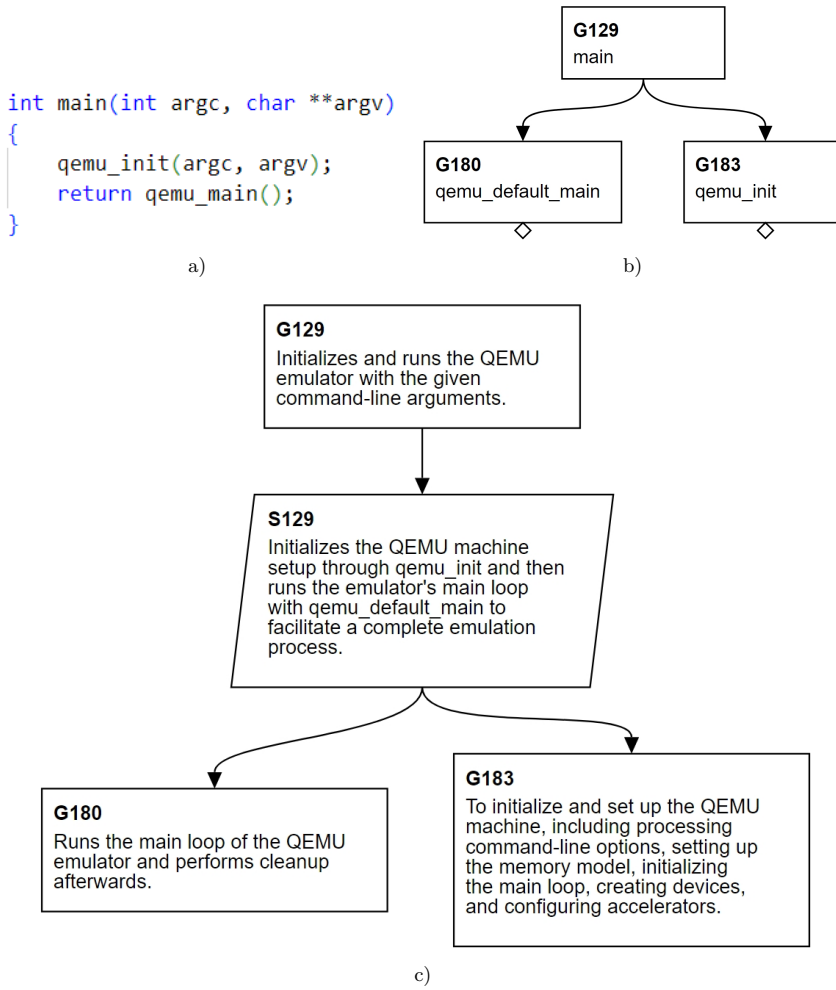

Figure 11. Comparison of QEMU project's top-level code, function call graph, and GSN diagram

7.2 Function Node Filtering

The filtration of function nodes to isolate those critical to the core functionality of the software introduces another layer of difficulty. This step demands a balance between comprehensiveness and conciseness, necessitating a deep semantic understanding of the code to discern which functions are essential. The subjective nature of "relevance" in this context poses a significant challenge, highlighting the need

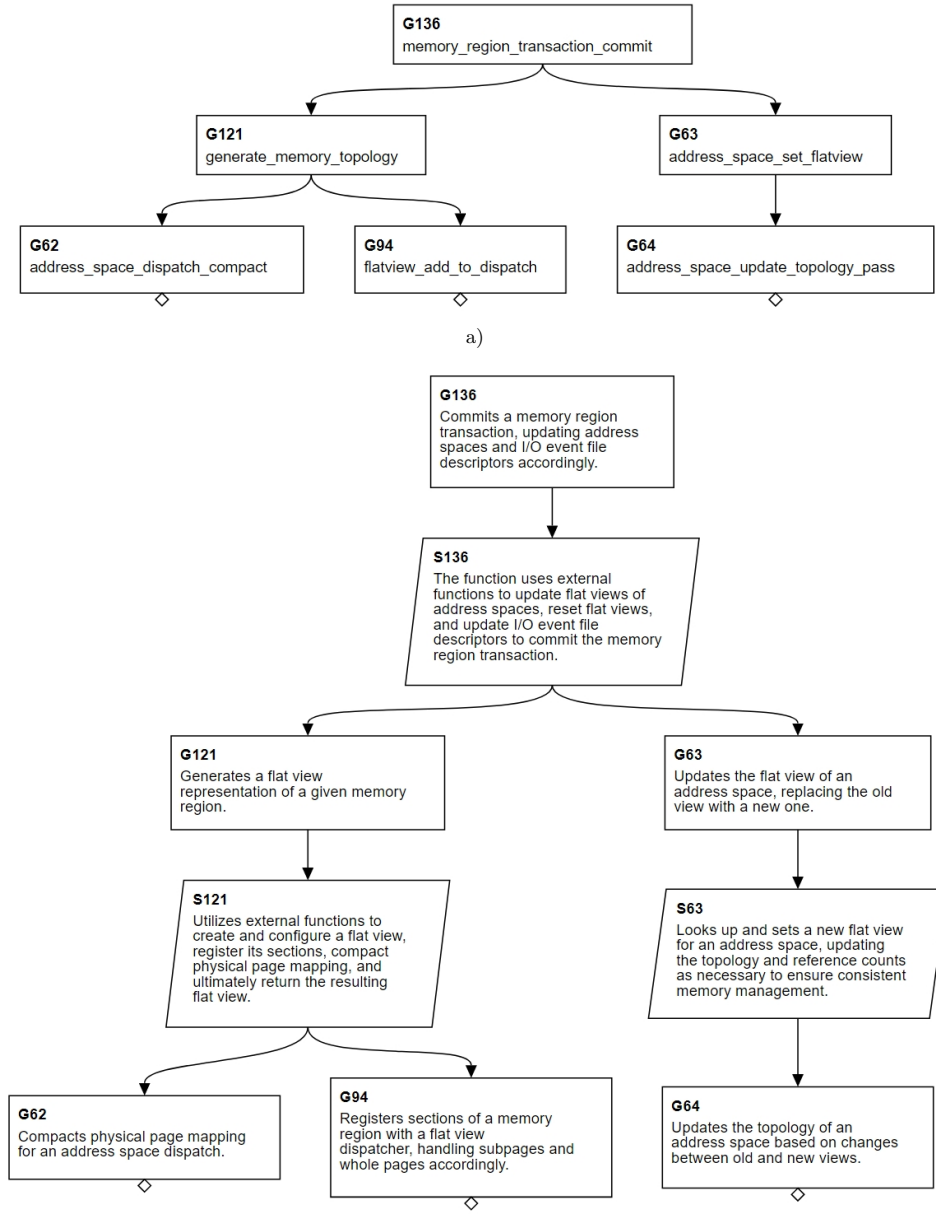

Figure 12. Function call graph and GSN diagram for memory management

for intelligent algorithms that can adaptively determine importance based on the software's context and the user's goals.

7.3 Integration of LLMs

Integrating LLMs for the natural language transformation of code descriptions and strategies represents a cutting-edge aspect of our methodology, blending the fields of software engineering and natural language processing. However, ensuring that these models generate accurate, relevant, and coherent descriptions and strategies is a non-trivial endeavor. It involves training or fine-tuning models on domain-specific datasets and continuously validating their outputs against expert knowledge, underscoring the interdisciplinary challenge of this approach. For some relatively rare business domain codes, the training data of general LLMs may not include this domain, potentially resulting in suboptimal performance. In such cases, fine-tuning or using RAG (Retrieval-Augmented Generation) techniques may be necessary.

7.4 Ethical Considerations

The choice of Llama3-70b over other advanced models for integration into Trusta is due to several reasons: while other LLMs such as GPT-4 and PaLM 2 can also perform the corresponding tasks, our approach requires reading a large amount of code, which is often confidential material within organizations. Therefore, we selected Llama3-70b, a relatively advanced model that can be deployed locally, for our experiments.

8 CONCLUSION

We have introduced a novel methodology that leverages GSN and large language models to enhance the understanding and maintenance of large software codebases. Our approach addresses the perennial challenges of code complexity, inadequate documentation, and communication barriers in software maintenance.

The core contributions of our work include the development of an automated tool that transforms complex code into a GSN framework, providing a structured and comprehensible representation of the software's logic. We integrated large language models to generate natural language descriptions of code, facilitating better understanding and communication among programmers. Our unique combination of function call graphs with GSN aids in visualizing and navigating the structure and functionality of code. We validated our methodology through experiments on real-world projects, demonstrating significant improvements in programmers' confidence and efficiency.

Beyond these contributions, our methodology has broader implications for both research and industry practices. By providing a systematic way to bridge the gap between code and its conceptual understanding, our approach can influence future

research in software engineering, particularly in automated documentation and intelligent code analysis. It paves the way for developing more advanced AI-assisted tools that can further alleviate the challenges of maintaining large codebases.

In the industry, adopting our method could transform software maintenance workflows. Enhancing code comprehension and communication among team members, it can lead to more efficient collaboration and project management. The integration of large language models with code analysis tools signifies a shift towards more intuitive and intelligent development environments, which could become standard practice in the industry.

Future work could explore refining large language models for technical domains, developing more advanced analysis tools, and integrating our approach with CI/CD pipelines and automated testing scenarios. Additionally, expanding the scope of our experiments to include more diverse codebases and programming environments will help validate the wide applicability of our method.

Acknowledgements

This work was supported by the National Natural Science Foundation of China under Grant No. 62472175 and the "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software under Grant No. 22510750100.

REFERENCES

[1] Kelly, T.—Weaver, R.: The Goal Structuring Notation – A Safety Argument Notation. Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases, 2004, https://www.academia.edu/47943884/The_Goal_Structuring_Notation_A_Safety_Argument_Notation.

[2] SCSC Assurance Case Working Group: Goal Structuring Notation Community Standard Version 3. Technical Report No. SCSC-141C, Safety-Critical Systems Club, CA, USA, 2021.

[3] Rinard, M.: Software Engineering Research in a World with Generative Artificial Intelligence. Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), 2024, doi: 10.1145/3597503.3649399.

[4] Kobayashi, N.—Nakamoto, A.—Shirasaka, S.: What Is It to Structuralize with Multiple Viewpoints by Using Goal Structuring Notation (GSN)? International Journal of Japan Association for Management Systems, Vol. 10, 2018, No. 1, pp. 125–130, doi: 10.14790/ijams.10.125.

[5] Kobayashi, N.—Shirasaka, S.: Proposal on How to Use Assurance Cases for Learning the Mindset to Respect Diversity. International Journal of Japan Association for Management Systems, Vol. 12, 2020, No. 1, pp. 7–16, doi: 10.14790/ijams.12.7.

[6] Nam, D.—Macvean, A.—Hellendoorn, V.—Vasilescu, B.—Myers, B.: Using an LLM to Help with Code Understanding. Proceedings of the IEEE/ACM

46th International Conference on Software Engineering (ICSE '24), 2024, doi: 10.1145/3597503.3639187.

[7] Bragdon, A.—Zeleznik, R.—Reiss, S. P.—Karumuri, S.—Cheung, W.—Kaplan, J.—Coleman, C.—Adeputra, F.—LaViola Jr, J. J.: Code Bubbles: A Working Set-Based Interface for Code Understanding and Maintenance. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10), ACM, 2010, pp. 2503–2512, doi: 10.1145/1753326.1753706.

[8] Szabo, C.: Novice Code Understanding Strategies During a Software Maintenance Assignment. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, 2015, pp. 276–284, doi: 10.1109/ICSE.2015.341.

[9] Al-Saiyd, N. A.: Source Code Comprehension Analysis in Software Maintenance. 2017 2nd International Conference on Computer and Communication Systems (ICCCS), IEEE, 2017, pp. 1–5, doi: 10.1109/CCOMS.2017.8075175.

[10] Kühlmann, E.—Hamer, S.—Quesada-López, C.: Software Visualization Using the City Metaphor: Students' Perceptions and Experiences. 2023 XLIX Latin American Computer Conference (CLEI), IEEE, 2023, pp. 1–10, doi: 10.1109/CLEI60451.2023.10346099.

[11] Sbruzzi, J. I.: Callcluster: Extracción, Análisis y Visualización de Callgraphs. Jornadas Argentinas de Informática e Investigación Operativa, 2021, pp. 54–68, https://sedici.unlp.edu.ar/handle/10915/141020 (in Spanish).

[12] Khaloo, P.—Maghoumi, M.—Taranta, E.—Bettner, D.—Laviola, J.: Code Park: A New 3D Code Visualization Tool. 2017 IEEE Working Conference on Software Visualization (VISSOFT), 2017, pp. 43–53, doi: 10.1109/VISSOFT.2017.10.

[13] Oberhauser, R.—Silfang, C.—Lecon, C.: Code Structure Visualization Using 3D-Flythrough. 2016 11th International Conference on Computer Science & Education (ICCSE), IEEE, 2016, pp. 365–370, doi: 10.1109/ICCSE.2016.7581608.

[14] Voorhees, J.: Primitive: Immersive Development Environment. 2023, https://primitive.io/.

[15] Lawler, E.—Gilpin, K.—Byrne, D.: AppMap: Visualize Your Runtime Code, Identify Problems, Find Solutions, Before Production. 2023, https://appmap.io/.

[16] Fuchs, S.—Amor, R.: Natural Language Processing for Building Code Interpretation: A Systematic Review. Proceedings of the 38th International Conference of CIB W78, 2021, pp. 294–303, http://itc.scix.net/paper/w78-2021-paper-030.

[17] OpenAI: GPT-4 Documentation. 2023, https://platform.openai.com/docs/models/gpt-4.

[18] Google: Introducing PaLM 2. 2023, https://ai.google/discover/palm2/.

[19] Despotou, G.—Kelly, T.: Design and Development of Dependability Case Architecture During System Development. Proceedings of the 25th International System Safety Conference (ISSC), 2007, https://www.researchgate.net/publication/266472547_Design_and_Development_of_Dependability_Case_Architecture_during_System_Development.

[20] Matsuno, Y.: Design and Implementation of GSN Patterns: A Step Toward Assurance Case Language. Information Processing Society of Japan Online Transactions, Vol. 7, 2014, pp. 59–68, doi: 10.2197/ipsjtrans.7.59.

[21] Maksimov, M.—Fung, N.—Kokaly, S.—Chechik, M.: Two Decades of Assurance Case Tools: A Survey. In: Gallina, B., Skavhaug, A., Schoitsch, E., Bitsch, F. (Eds.): Computer Safety, Reliability, and Security (SAFECOMP 2018). Springer, Cham, Lecture Notes in Computer Science, Vol. 11094, 2018, pp. 49–59, doi: 10.1007/978-3-319-99229-7_6.

[22] Møller, A.—Schwartzbach, M. I.: Static Program Analysis. Technical Report. Aarhus University, Denmark, 2024, https://users-cs.au.dk/amoeller/spa/spa.pdf.

[23] Ball, T.: The Concept of Dynamic Analysis. Vol. 1687, 1999, pp. 216–234, doi: 10.1007/3-540-48166-4_14.

[24] Eisenbarth, T.—Koschke, R.—Simon, D.: Aiding Program Comprehension by Static and Dynamic Feature Analysis. Proceedings IEEE International Conference on Software Maintenance (ICSM 2001), 2001, pp. 602–611, doi: 10.1109/ICSM.2001.972777.

[25] Weidendorfer, J.: Sequential Performance Analysis with Callgrind and KCachegrind. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (Eds.): Tools for High Performance Computing. Springer, Berlin, Heidelberg, 2008, pp. 93–113, doi: 10.1007/978-3-540-68564-7_7.

[26] Kaszuba, G.: Python Call Graph. 2016, https://pycallgraph.readthedocs.io/en/master.

[27] Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05), USENIX, 2005, pp. 41–46, https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf.

[28] Meta: Introducing Meta Llama 3: The Most Capable Openly Available LLM to Date. 2024, https://ai.meta.com/blog/meta-llama-3/.

[29] Joshi, A.—Kale, S.—Chandel, S.—Pal, D. K.: Likert Scale: Explored and Explained. Current Journal of Applied Science and Technology, Vol. 7, 2015, No. 4, pp. 396–403, doi: 10.9734/BJAST/2015/14975.

[30] Gaynor, A. et al.: Cryptography Is a Package Which Provides Cryptographic Recipes and Primitives to Python Developers. 2025, https://github.com/pyca/cryptography.

A CASE STUDY: TOOL USE IN CRYPTOGRAPHY CODE

In this appendix, we present a use case of our tool by applying it to a Python code that utilizes the `cryptography` library [30] for encrypting a message. The `cryptography` library contains approximately 70 000 lines of code. Using our Trusta tool, we generated a GSN with 53 nodes representing the data encryption steps, which took 12 minutes to complete. The purpose is to transform the code's execution process into GSN, thereby aiding in code comprehension and helping programmers explore the encryption process of the `cryptography` library.

```
from cryptography.fernet import Fernet

def main():
    key = Fernet.generate_key()
    f = Fernet(key)
    token = f.encrypt(b"A␣really␣secret␣message.")

if __name__ == '__main__':
    main()
```

Listing 3. Example code using the cryptography library

Using our tool, the above code is converted into a high-level GSN diagram, as shown in Figure 13. This diagram provides an overview of the encryption process implemented in the code.

In Figure 13, the top-level goal is "Generates a secure encryption key and uses it to encrypt a secret message". This is supported by sub-goals such as "Generates a random, URL-safe base64-encoded key". and "Encrypts given data". The Strategy (in node 3) provides information about the cryptography library and the methods used. Figure 13 elaborates on the sub-goals by mapping them to specific code statements and functions:

Node 3: The main goal to encrypt a message is achieved by the `main()` function.

C1 in Node 3: Initializing the cipher is done through `Fernet(key)`.

Node 4: Generating a key is performed by `Fernet.generate_key()`.

Node 6: Encryption is executed by `f.encrypt()`.

For a more in-depth understanding, the tool generates a detailed GSN diagram depicted in Figure 14. This diagram breaks down each step and shows the interactions between code components and functions.

In Figure 14, the goal **Node 8**: "Encrypts data from parts, including the current time and initialization vector, and returns the encrypted result" is achieved through several sub-goals:

Node 8 ⊢ Node 10: Checks if a given value is of type bytes and raises an error if it is not.

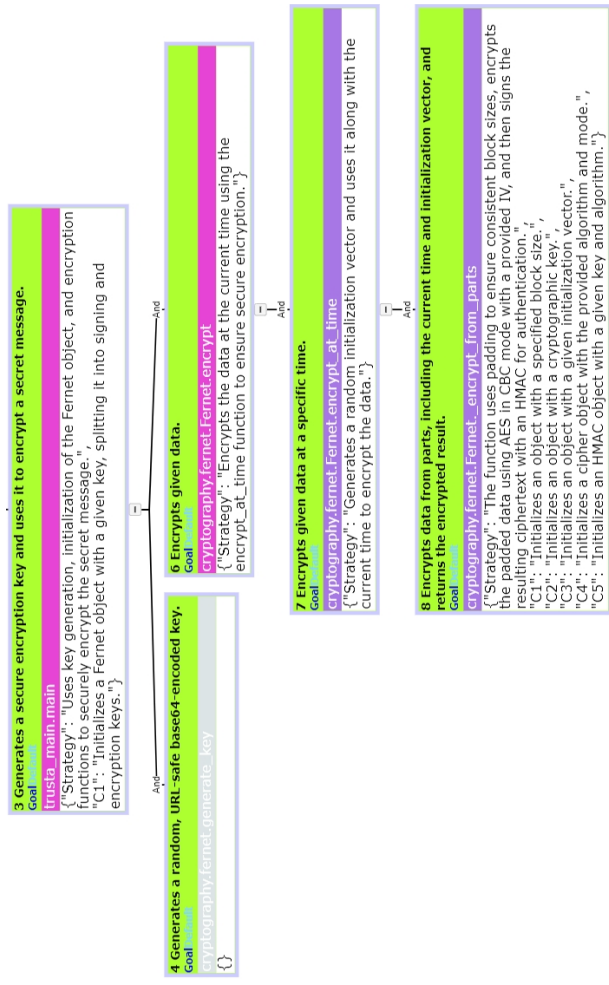

Figure 13. High-level GSN diagram of the encryption process

Node 8 ⊢ Node 13: Creates a padding context for cryptographic operations.

Node 8 ⊢ Node 15: Updates the internal buffer with new data and returns the padded result.

Node 8 ⊢ Node 18: Finalizes the encryption/decryption process by padding and returning the result.

Node 8 ⊢ Node 30: Initializes and sets up an encryption context.

Node 8 ⊢ Node 40: Updates a cryptographic context with given data and returns the resulting encrypted data.

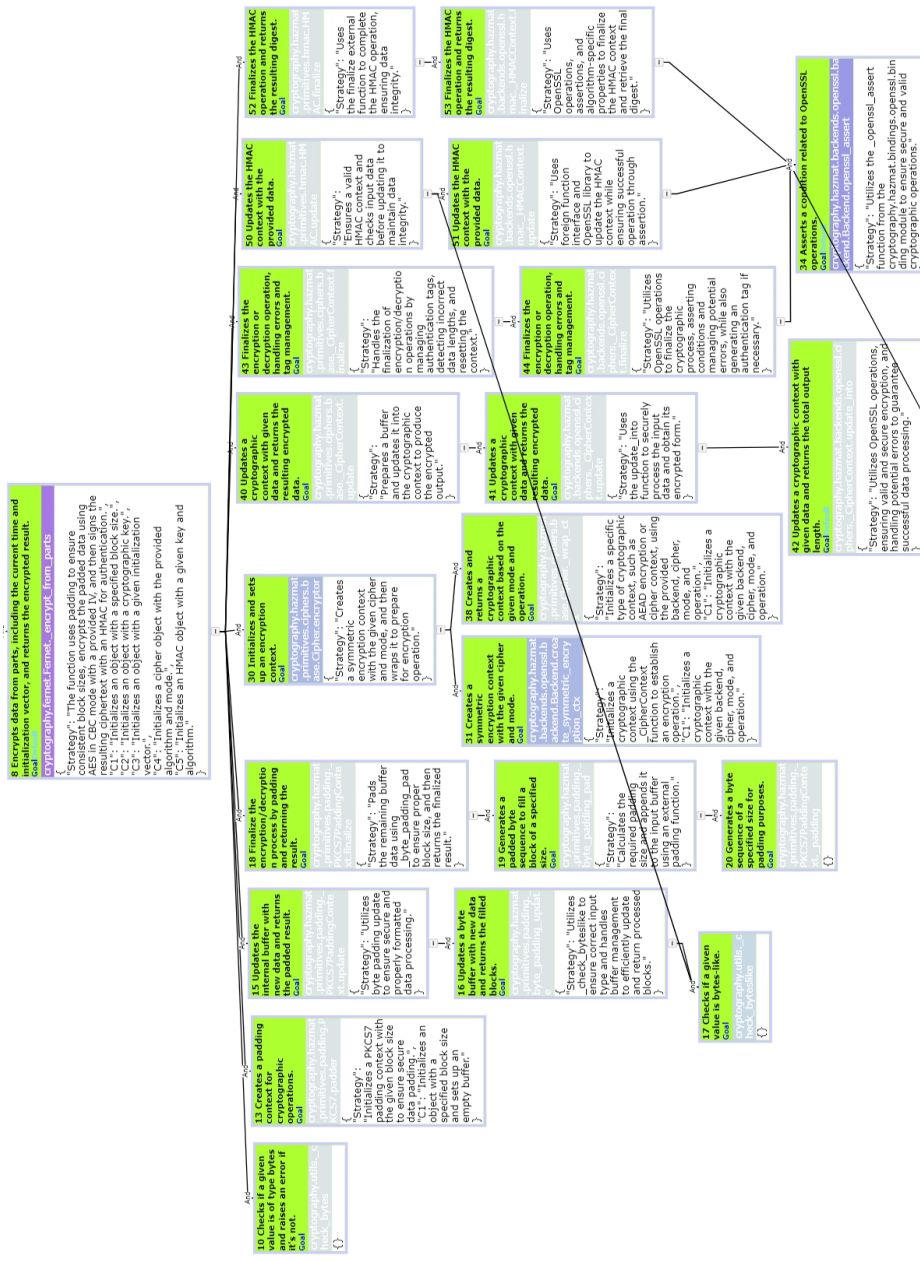

Figure 14. Detailed GSN diagram of code execution flow

Node 8 ⊢ Node 43: Finalizes the encryption or decryption operation, handling errors and tag management.

Node 8 ⊢ Node 50: Updates the HMAC context with the provided data.

Node 8 ⊢ Node 52: Finalizes the HMAC operation and returns the resulting digest.

The strategy (in node 8) employed is:

Strategy: "The function uses padding to ensure consistent block sizes, encrypts the padded data using AES in CBC mode with a provided Initialization Vector (IV), and then signs the resulting ciphertext with an HMAC for authentication."

The contexts (in node 8) provide additional information about the initialization of various cryptographic objects:

C1: Initializes an object with a specified block size.

C2: Initializes an object with a cryptographic key.

C3: Initializes an object with a given initialization vector.

C4: Initializes a cipher object with the provided algorithm and mode.

C5: Initializes an HMAC object with a given key and algorithm.

These contexts (C1–C5) correspond to the setup steps required before encryption and authentication processes can proceed.

The GSN diagrams generated by our tool offer an overview and a detailed explanation of the code's encryption process. This visual representation facilitates better understanding and communication among programmers, ultimately enhancing code comprehension and assisting them in exploring the encryption mechanisms within the `cryptography` library.

Zezhong Chen is currently pursuing his Ph.D. degree at the East China Normal University in Shanghai, China. His research interests include software engineering and formal methods.

Yuxin Deng is Professor at the Shanghai University of Finance and Economics. His research interests include concurrency theory, program semantics, formal verification, and quantum computing.

Wenjie Du is Lecturer at the Shanghai Normal University. Her research interests include concurrency theory and formal verification.