Advancements in Call Graph Methodologies for Enhanced Program Comprehension: A Review

Rakan Alanazi [1†],

Department of Information Technology,
Faculty of Computing and Information Technology,
Rafha 91911, Saudi Arabia

Abstract
In the evolving landscape of software development, where maintaining and understanding complex systems is increasingly challenging, Call graph techniques play a critical role in enhancing software comprehension by providing a visual and structural representation of function calls within a system. This paper explores the role of call graphs in simplifying software maintenance and debugging. It highlights how call graphs significantly improve developers' understanding of system architectures and function interactions, reducing the time spent on manual code exploration. Furthermore, the paper explores recent advancements in call graph techniques, particularly the integration of machine learning and deep learning models with traditional call graph approaches. This hybrid methodology demonstrates enhanced accuracy and relevance in tasks such as program comprehension and code refactoring, making it a valuable tool for modern software engineering practices.

Keywords:
Call Graph; Program Comprehension; Software Comprehension; Software Analysis.

1. Introduction

As software systems become complex, so do the relationships between various components, making it increasingly difficult for engineers to gain a comprehensive knowledge of the system. Without a thorough understanding of the software's structure and dependencies, changes might have unexpected consequences, raising the possibility of errors and system instability. Effective program comprehension is essential for various types of software development tasks, including reuse, debugging, testing, maintenance, and evolution. Usually, developers use system documentation to gain a high-level understanding of the software, which they then map to the actual implementation. However, due to the dynamic nature of software development, documentation is frequently outdated and does not correctly reflect the current status of software [1]. This mismatch increases the effort of understanding the system, making manual mapping time-consuming and error prone [2]. To overcome these challenges, developers turn to various analysis techniques that enhance their understanding of the software's structure and implementation.

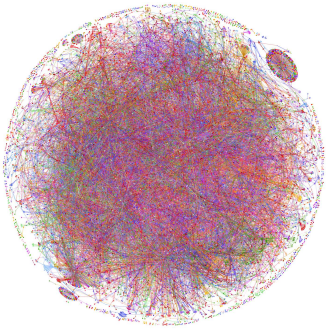

Fig. 1. Snippet of source code in Java and its corresponding call graph

One of the most widely used techniques for facilitating software comprehension and operational analysis is the call graph. Call graphs play an essential role in simplifying tasks related to software maintenance, debugging, and evolution by providing a visual representation of caller-callee relationships within a system. By doing so, they offer developers a clear view of function interactions, which helps in identifying dependencies and understanding the flow of execution. The traditional call graph, which represents function calls as directed graphs, has been a fundamental tool for visualizing software structure and dependencies. However, limitations such as single-level abstraction and static representations have hindered its effectiveness in facilitating comprehensive understanding. Figure 1 represents the function call graph of WEKA [3], an open-source data mining software tool written in Java.

Recent researches have focused on addressing these challenges through innovative methodologies

Manuscript received November 5, 2025
Manuscript revised November 20, 2025
https://doi.org/10.22937/IJCSNS.2025.25.11.6

and tools. For instance, various researchers [2,4,5,6] have developed advanced visualization tools that enhance the graphical representation of call graphs, allowing for a more improved understanding of software systems. Notably, techniques such as hierarchical clustering [7] and multi-level abstractions [5,8] have emerged, enabling developers to visualize call graphs at varying granularity levels, thus improving the comprehensibility of large codebases.

Moreover, the integration of semantic information with call graph structures provides a more detailed view of code dependencies. For example, the Semantic Code Graph (SCG) [9] model captures both the contextual meanings and structural relationships within the code. This enriched perspective supports software comprehension by revealing how different parts of the code interact, and it also helps identify potential issues that may arise from these dependencies

Additionally, the application of neural network models to call graph analysis has shown promising results. Recent studies [8,10] have demonstrated that incorporating call network information into neural models significantly improves the performance of source code summarization, aligning the generated outputs more closely with human expectations. This advancement underscores the importance of structural information in developing more accurate and relevant representations of software components.

Furthermore, call graphs have been effectively employed in analyzing software evolution [11,12] and defect prediction [13]. Tools such as GraphEvo [11] facilitate the examination of changes across different software versions, providing valuable insights into structural evolution, while novel defect prediction methods leverage deep learning techniques to enhance the identification of potential issues in evolving software systems.

We restricted the scope of our literature review to research conducted within the past six years, specifically from 2018 onward. The reason for this was to ensure that the review focuses on the most recent advancements and updates in the field, reflecting the current state of knowledge and technological progress. By narrowing the timeframe, we aim to capture the latest trends, methodologies, and innovations that have emerged in the study of call graphs and their role in program comprehension. Our search strategy involved the use of specific keywords to target research directly related to the topic. The list of search terms included "Call Graph" and "Program Comprehension" with variations and combinations of these terms applied to capture a broad range of studies addressing different aspects of call graph analysis, software comprehension, and their applications in software maintenance and debugging. By limiting the search to recent studies and employing targeted search terms, we ensured that the review presents an up-to-date and relevant synthesis of research, emphasizing the importance of call graph techniques in modern software development.

The rest of this paper is organized as follows: Section 2 presents a brief background on the topic of call graphs. Section 3 discusses the related work. Finally, Section 4 concludes the paper.

2. Background

In this section, we present a brief background on the topic of call graphs and define some of the terms repeatedly used in the rest of this paper. In a software system, a call graph [13] is represented as a directed graph $\vec{G} = (V, E)$, where $\{V\}$ is the set of entities, and $\{E\}$ is the set of edges. Each edge, $\{e\}$ in E, represents a call between two entities (u_{caller} v_{callee}). The in-degree of a node v, denoted by $deg^-(v)$, refers to the number of incoming edges to node v, while the out-degree, $deg^+(v)$, indicates the number of outgoing edges from node v. Figure 2 illustrates snippet of source code in Java and its corresponding call graph.

In object-oriented programming (OOP), call graphs can be constructed at different levels of abstraction, including Function Call Graph (FCG), Class Call Graph (CCG), and Package Call Graph (PCG). Each type of graph provides a distinct view of the system, with FCG focusing on the interactions between individual functions, CCG visualizing the relationships between classes, and PCG capturing the dependencies between packages. These different perspectives enable developers to analyze the system from multiple vantage points, facilitating a deeper understanding of the software and allowing for more effective maintenance and evolution of the codebase.

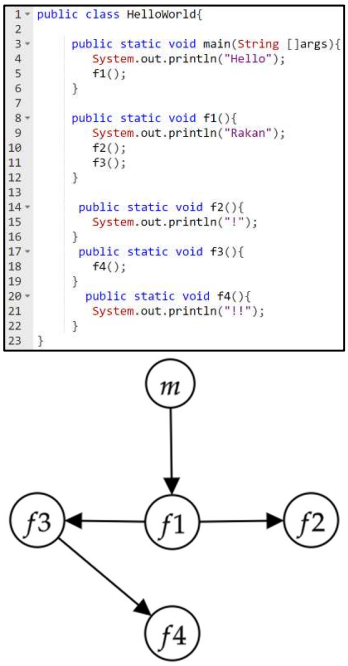

Fig. 2. Snippet of source code in Java and its corresponding call graph

3. Literature Review

Several researchers have developed diverse tools for creating and visualizing call graphs of systems [7, 11, 15, 16, 17, 18, 19]. These tools are primarily used to render graphical representations. For instance, Alnabhan et al. [18] proposed a two-dimensional software visualization approach where different geometric shapes represent various source code entities. Specifically, classes and methods are depicted as rectangles and circles, respectively, with arrows illustrating their relationships.

However, these tools have limitations in visualizing call graphs [5], including the fact that they are limited to a single level of abstraction, typically at the function level, which may be insufficient for various maintenance activities that require diverse levels of comprehension. In addition, these tools frequently only show a section of the call graph, failing to reveal the entire system structure, which may affect effective software comprehension. Also, including all system components, such as classes, methods, and attributes, might cause information overload, making it harder to understand the software system. Furthermore, generated call graphs are often provided in static formats such as PNG and DOT, which take additional work to comprehend and understand, potentially lowering program comprehension efficiency [6].

Several researchers have contributed to addressing the challenges associated with program comprehension and visualization tools. One such effort is seen in Graph Buddy, an interactive code browsing tool proposed in [6]. The tool can be integrated as a plugin for popular IDEs, Graph Buddy enhances code comprehension by visually representing code dependencies through a semantic code graph. This approach allows developers to explore complex code relationships more effectively while ensuring that only relevant details are displayed, thereby preventing information overload and maintaining clarity in large codebases. The tool supports both Scala and Java and has been evaluated in a user study involving 10 programmers. The study revealed that tasks involving code comprehension were considerably more challenging without the visual assistance provided by Graph Buddy, demonstrating the tool's potential to improve the process of navigating and understanding complex codebases. This work aligns with ongoing efforts to improve program comprehension through visualization and semantic analysis, offering a practical solution for managing code dependencies in large software systems.

Recently, authors in [20] proposed *FcTree*, a tool designed to visualize call graphs during the execution of a program. The design of *FCTree* optimized the advantages of existing visualization tools to address the limitations of existing call graph visualizations. *FcTree* creates a visual representation that highlights the sequence and relationships between function calls as they occur in real-time. This visualization aids developers in understanding the dynamic behavior of software by providing clear insights into how functions interact during execution. The tool helps in debugging, performance analysis, and program comprehension by making it easier to trace the flow of execution and identify potential issues or inefficiencies within the code. The authors conducted a pilot study to evaluate the effectiveness of *FCTree*. This study revealed that existing single-view function call visualizations often fail to present the necessary information comprehensively. The evaluation included both subjective and objective

experiments, as well as a field study in a real-world scenario, demonstrating the practical applicability of *FCTree*.

Gharibi et al. [17] proposed a hierarchical clustering technique for static call graphs. The approach facilitates program comprehension by grouping call graphs into relatable clusters. These clusters are driven by similar execution paths. The hierarchical clustering approach allows developers to understand big software systems by breaking them down into more manageable subcomponents. However, a potential drawback of this technique is the creation of a large number of cluster nodes. This can lead to an overly complex structure, making it difficult for developers to navigate through the numerous clusters efficiently. While the technique helps to reduce the complexity of large codebases, managing and interpreting a large number of clusters may still be challenging in cases of very large or complex software systems.

To address the limitations of hierarchical clustering in managing large call graph structures, the author [8] presents an innovative approach that enhances program comprehension by labeling and describing clusters of execution paths using topic modeling techniques. As clustering algorithms generally do not provide insight into the meaning of the resulting clusters, the author employs Latent Dirichlet Allocation (LDA) and Bidirectional Encoder Representations from Transformers (BERT) models to automatically generate meaningful labels for these clusters. Cluster labeling is crucial as it serves as an indicator of the quality and success of the clustering process, helping developers interpret and understand the grouped execution paths. By applying these topic models, the approach aims to analyze and interpret the semantic meaning behind the clusters, ultimately providing developers with clearer insights into software functionality. In his case study on two systems—SweetHome3D and jMonkeyEngine—a combination of LDA and BERT (BERT+LDA) yielded the best results, significantly improving the coherence of cluster labels. This combined model offers a richer, more contextual understanding of the clusters by describing the functionality of a group of execution paths in a more meaningful way. This technique not only aids in program comprehension but also enhances the usability of the hierarchical clustering approach by offering richer, more meaningful descriptions of cluster contents.

A similar direction is taken by Bhattacharjee et al. [21], presenting a novel approach to improving program comprehension through the creation of an abstract code summary tree. This technique involves transforming a hierarchical clustering tree (AHC tree) of a program's call graph into a more understandable format by flattening the clusters and reducing the number of nodes for simplicity. The abstract nodes are then summarized into natural language text derived from method comments, helping developers grasp high-level overviews of the codebase without needing to dive into complex code details. To validate the effectiveness of this method, the authors collected feedback from developers who used the tool for code maintenance tasks. The results indicated that the generated abstract summary tree provided a clearer, more accessible way for developers to understand large and complex codebases. This, in turn, made it easier to perform software maintenance and debugging activities. The study concludes that this approach significantly enhances program comprehension by offering a clear, hierarchical overview of the code structure combined with meaningful textual summaries.

Building on this work, Alanazi R. et al. [5,8] recently introduced a coarsening technique to create multi-level, hierarchical representations of the call graph. This advancement addresses the limitations faced in understanding large call graphs and the issue of single-level granularity. The proposed hierarchical clustering approach of execution paths allows for visualizing the call graph at different granularity levels for various software units like packages, classes, and functions, aiding in a more comprehensive understanding of the software system. The main contributions of the research include the development of a mechanism to link hierarchical abstraction clusters to their corresponding call graphs, enhancing the visualization and comprehension of the software system's structure. By focusing on clustering execution paths over multiple levels of abstractions, the researchers aimed to reduce the size of the graph at each hierarchical level, enabling developers to better understand the software system with varying levels of granularity, unlike approaches that provide a single level of granularity only.

The authors measured the effectiveness of their approach through a comprehensive user study involving 18 software engineers from various industries. Participants performed tasks using the proposed system and provided feedback via a structured survey, which included questions assessing the tool's usefulness and usability using the System Usability Scale (SUS). They evaluated the meaningfulness of the hierarchical clusters generated by the tool and its ability to identify software functionalities, which is essential for program comprehension. Additionally, the effectiveness was gauged by the tool's capacity for visual exploration of execution paths in both call graphs and hierarchical views, with participants rating its usefulness in these tasks.

In addition to hierarchical clustering, other researchers have explored the use of vector representations for software clustering. The authors [22] propose using vector representations of code elements alongside traditional call graph structures to improve the accuracy and relevance of software clustering. This hybrid approach leverages the strengths of both techniques: vector semantics capture the contextual meaning of code, while call graphs represent the structural relationships between components. The study demonstrates that integrating these methods results in more meaningful and useful clusters, aiding in tasks such as program comprehension, maintenance, and refactoring. In particular, the proposed method enhances the analysis of software systems by integrating document embeddings, generated via the *Doc2Vec* algorithm, with call graphs obtained from Static Graph Analyzers to form an augmented graph. Utilizing the Louvain Algorithm on this augmented graph, the method successfully uncovers the community structure and proposes module-level clusterings. The integration of vector semantics and call graph information effectively generates meaningful clusterings of software systems, outperforming state-of-the-art clustering algorithms and traditional agglomerative methods. The findings show that the proposed method efficiently recovers the ground truth clustering of the Linux Kernel, emphasizing the importance of incorporating semantic information and call graph data in clustering large software systems.

In another area, the author [23] explores the structural properties of function-call graphs in open-source software by applying techniques from complex network analysis. The authors investigate various metrics such as degree distribution, clustering coefficient, and shortest path length to understand the topological characteristics of these graphs. The study reveals that function-call graphs exhibit small-world and scale-free properties, which are typical of complex networks. These insights help in comprehending the architectural design and evolution of software systems, and they highlight potential areas for optimization and maintenance. The findings underscore the importance of applying complex network theory to software engineering for enhanced software analysis and comprehension.in [9] a new model for representing code dependencies is presented named Semantic Code Graph (SGC). The traditional call graph primarily focuses on the structural relationships between functions or methods in a program. It represents how functions call each other, depicting the flow of execution within the software. The nodes typically represent functions or methods, and the edges represent the calls between these functions. However, the proposed representing Semantic Code Graph goes beyond the structural relationships captured in a call graph by incorporating semantic information. It not only represents the connections between code elements (like classes, methods, and variables) but also embeds the meaning and context of these elements. This model integrates both the structural aspects of the code and the semantic relationships, providing a richer, more comprehensive view of the software. The paper conducts a study that aims to evaluate the SCG's effectiveness in enhancing software comprehension compared to existing models like the Class Collaboration Network (CCN) and Call Graph (CG) by analyzing eleven open-source projects, employing various data analysis techniques. The results indicate that the SCG significantly enhances software comprehension capabilities compared to the other models.

Another use of call graphs is in the software evolution, The paper [11] presents a novel framework named *GraphEvo*, which utilizes call graphs to analyze and comprehend the evolution of software systems. By examining the changes in call graphs across different software versions, *GraphEvo* identifies and characterizes the evolution of software

structures. This framework provides valuable insights into significant modifications, recurring patterns, and the overall impact of these changes on software architecture. The findings indicate that *GraphEvo* is a powerful tool for understanding the dynamic nature of software systems, assisting developers in tasks such as maintenance, refactoring, and managing software evolution by offering a detailed view of structural changes over time. in [12], The paper proposes a novel approach to analyze the evolution of software systems through Call Graph Evolution Analytics. This method helps in extracting valuable information from a series of evolving call graphs, which represent different versions of a software system. The author develops two key techniques: Call Graph Evolution Rules (CGERs) and Call Graph Evolution Subgraphs (CGESs). CGERs are designed to capture co-occurrences of dependencies, similar to association rule mining, which helps in identifying how different parts of the software interact. On the other hand, CGESs focus on the evolution of dependency patterns, allowing for a more detailed analysis of how these dependencies shift across different versions of the software. Together, these techniques provide a comprehensive framework for analyzing software evolution and dependency management, which is crucial for ensuring software stability and reducing potential errors during the evolution process.

The paper also includes empirical analysis conducted on ten large evolving software systems, with a detailed case study on Maven-Core. This empirical work demonstrates the practical applicability of the proposed techniques in real-world scenarios, showcasing their effectiveness in supporting dependency evolution management. Furthermore, the author connects his findings to established theories in software evolution, such as Lehman's laws, thereby situating their contributions within a broader theoretical context. Lastly, the author expresses his intention to publish a more comprehensive study on Stable CGERs and CGESs mining, indicating a commitment to advancing research in this area and providing deeper insights into software evolution analytics.

Call graphs have also been used in software defect prediction. The paper [13], presents a novel approach to predicting software defects by leveraging deep learning and network portrait divergence (NPD). The authors propose using network portrait divergence, a measure of structural differences between network snapshots over time, to capture the evolution of software systems. Deep learning techniques, including Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks, are employed to analyze software networks and predict defects. By integrating this metric with deep learning models, the study aims to enhance the accuracy of defect prediction. The authors conduct a study that compares the performance of Network Portrait Divergence (NPD) with traditional software metrics like Lines of Code (LOC) and Cyclomatic Complexity (CC), demonstrating NPD's outperforming in capturing the evolutionary aspects of software networks for defect prediction. Also, Experiments conducted on real-world software projects further validate the effectiveness of NPD, showcasing its practical applicability and reliability in real-world scenarios. The results demonstrate that this approach effectively identifies potential defects by considering the dynamic changes in software structure, thus improving maintenance and quality assurance processes. This method provides a robust tool for anticipating defects in evolving software systems, contributing to more reliable and maintainable software development.

Furthermore, other advancements in neural network models have also been applied to call graph analysis, The authors [10] propose a method for encoding the context of function calls in a neural network model in order to produce more accurate and relevant source code summaries. By incorporating call network information, the model can gain a better understanding of function relationships and dependencies, resulting in increased summarization performance. The results show that this approach outperforms existing methods that do not take call graph context into account, emphasizing the value of using structural information in neural source code summarization. A study was conducted involving 20 programmers, the generated summaries were perceived to be as accurate, readable, and concise as those written by humans. This finding indicates that the proposed method not only improves the technical performance of summarization models but also aligns well with human expectations and standards for quality.

4. Conclusion

The advancements in call graph research have significantly improved the ability of developers to understand and manage complex software systems. Although traditional static call graphs are useful, however, it has been limited in their capacity to provide comprehensive insights into the intricate relationships and dependencies within large-scale applications. Recent methodologies, such as hierarchical clustering, multi-level abstractions, and semantic code graphs, have addressed these limitations by offering more detailed, context-rich, and dynamic representations of software structures. These innovations have proven effective in enhancing program comprehension and aiding tasks like debugging, maintenance, refactoring, and software evolution analysis. Moreover, the integration of neural networks and deep learning techniques into call graph analysis has led to breakthroughs in areas like source code summarization and defect prediction. These models have demonstrated the value of structural and semantic information, resulting in more accurate predictions and summaries that align closely with human judgment. Tools and approaches such as Call Graph Evolution Analytics further underscore the importance of call graphs in understanding software evolution and ensuring system stability over time. Future research should focus on combining these methodologies to create even more powerful tools, further enhancing the capabilities of call graph-based program comprehension. Through these innovations, developers will be better equipped to maintain, analyze, and evolve their codebases effectively.

Acknowledgment

The authors extend their appreciation to the Deanship of Scientific Research at Northern Border University, Arar, KSA, for funding this research work through project number NBU-FFR-2024-1661-06.

References

[1] M. Z. Khan, R. Naseem, A. Anwar, I. U. Haq, A. Alturki, S. S. Ullah, and S. A. Al-Hadhrami, "[retracted] a novel approach to automate complex software modularization using a fact extraction system," Journal of Mathematics, vol. 2022, no. 1, p. 8640596, 2022.

[2] J. Mortara, P. Collet, and A.-M. Dery-Pinna, "Visualization of object-oriented software in a city metaphor: Comprehending the implemented variability and its technical debt," Journal of Systems and Software, vol.208, p. 111876, 2024.

[3] I. H Witten, E. Frank, M. A Hall, and C. J Pal, "Data mining practical machine learning tools and techniques," 2017.

[4] C. Li, Y. Pei, Y. Shen, J. Lu, Y. Fan, X. Linghu, Y. Tian, and K. Wang, "Pyvisvue3d3: Python visualization from hierarchy tree to call graph," SoftwareX, vol. 26, p. 101689, 2024.

[5] R. Alanazi, G. Gharibi, and Y. Lee, "Facilitating program comprehension with call graph multilevel hierarchical abstractions," Journal of Systems and Software, vol. 176, p. 110945, 2021.

[6] K. Borowski, B. Balis, and T. Orzechowski, "Graph buddy—an interactive code dependency browsing and visualization tool," in 2022 Working Conference on Software Visualization (VISSOFT). IEEE, 2022, pp.152–156.

[7] G. Gharibi, R. Alanazi, and Y. Lee, "Automatic hierarchical clustering of static call graphs for program comprehension," in 2018 IEEE International conference on big data (Big Data). IEEE, 2018, pp.4016–4025.

[8] R. Alanazi, Software Analytics for Improving Program Comprehension. University of Missouri-Kansas City, 2021.

[9] K. Borowski, B. Balis, and T. Orzechowski, "Semantic code graph–an information model to facilitate software comprehension," IEEE Access, 2024.

[10] A. Bansal, Z. Eberhart, Z. Karas, Y. Huang, and C. McMillan, "Function call graph context encoding for neural source code summarization," IEEE Transactions on Software Engineering, vol. 49, no. 9, pp. 4268–4281, 2023.

[11] V. Walunj, G. Gharibi, D. H. Ho, and Y. Lee, "Graphevo: Characterizing and understanding software evolution using call graphs," in 2019 IEEE International Conference on Big Data (Big Data), 2019, pp. 4799–4807.

[12] A. Chaturvedi, "Call graph evolution analytics over a version series of an evolving software system," in Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–5.

[13] V. Walunj, G. Gharibi, R. Alanazi, and Y. Lee, "Defect prediction using deep learning with network portrait divergence for software evolution," Empirical Software Engineering, vol. 27, no. 5, p. 118, 2022.

[14] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 7, no. 2, pp. 158–191,1998.

[15] W. Jin, S. Xu, D. Chen, J. He, D. Zhong, M. Fan, H. Chen, H. Zhang, and T. Liu, "Pyanalyzer: An effective and practical approach for dependency extraction from python code," in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–12.

[16] K. Borowski and B. Bali's, "scg-cli - a tool supporting software comprehension via extraction and analysis of semantic code graph," arXiv.org, vol. abs/2310.03044, 2023.

[17] G. Gharibi, R. Tripathi, and Y. Lee, “Code2graph: automatic generation of static call graphs for python source code,” in Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, 2018, pp. 880–883.
[18] M. Alnabhan, A. Hammouri, M. Hammad, M. Atoum, and O. Al-Thnebat, “2d visualization for object-oriented software systems,” in 2018 International Conference on Intelligent Systems and Computer Vision (ISCV), 2018, pp. 1–6.
[19] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, “Pycg: Practical call graph generation in python,” in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021, pp. 1646–1657.
[20] F. Zhou, Y. Fan, S. Lv, L. Jiang, Z. Chen, J. Yuan, F. Han, H. Jiang, G. Bai, and Y. Zhao, “Fctree: Visualization of function calls in execution,” Information and Software Technology, p. 107545, 2024.
[21] A. Bhattacharjee, B. Roy, and K. A. Schneider, “Supporting program comprehension by generating abstract code summary tree,” in Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, 2022, pp. 81–85.
[22] M. Papachristou, “Software clusterings with vector semantics and the call graph,” in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 1184–1186.
[23] V. Tunalı and M. A. A. T¨uys¨uz, “Analysis of function-call graphs of open-source software systems using complex network analysis,” Pamukkale ¨Universitesi M¨uhendislik Bilimleri Dergisi, vol. 26, no. 2, pp. 352–358, 2020.

Rakan Alanazi received a Ph.D. in Computer Science from the University of Missouri - Kansas City in the United States. His research interests include Software Analytics, Software Engineering,
Machine Learning, Software Visualization, and Program Comprehension.