A review on graph representation for object-oriented programming

Umar Farooq Khattak[1*], *Hussein* Ali Hussein Al Naffakh[2] and *Aıtızaz* Alı[3]

[1]UNITAR International University, Malaysia,
[2]College of Health and Medical Techniques, University of Alkafeel, AlNajaf, Iraq,
[3]Asia Pacific University, Malayisa

Abstract. Relationships and connections between entities are typically represented by graphs, a fundamental data structure in computer science. A graph illustrates the control flow among statements within a program, whereas a dependence graph delineates the interrelationships among various objects within it. The object-oriented paradigm introduces numerous innovative concepts, including encapsulation, inheritance, polymorphism, and dynamic binding. A complete object-oriented graph representation entails structuring and conceptualizing a graph based on the principles of object-oriented programming (OOP). Several models have been suggested previously designed to depict the diverse features of object-oriented programming. This study distinguishes between partial and fully object-oriented graph representations. It also analyses the fully object-oriented graph representation models.

1 Introduction

Graphs, a fundamental data structure in computer science, are widely used to model relationships and connections between entities [1]. Whether representing social networks, transportation systems, or hierarchical structures, effective graph representation is crucial for various applications. One powerful paradigm for achieving a robust and flexible graph representation is through the principles of Object-Oriented Programming (OOP) [9]. Object-Oriented Graph to design and model the components of a graph. This approach leverages the modularity, encapsulation, and extensibility of OOP to create a comprehensive and intuitive representation of complex relationships within a graph [9]. Graphs are mathematical structures that depict pairwise relationships between objects, and it is widely acknowledged that they offer an intriguing formalism for modeling the intricate structures encountered in software engineering. [1]. Graphical methods offer a visual depiction of the system's structure, facilitating the examination of the logical flow

* Corresponding author: umar.farooq@unitar.my

within the program Understanding the interactions and dependencies among program elements requires visually representing the program being tested through an intermediate graph representation is crucial. The graphical depiction of the program can be expressed as follows: G = (N, E).

N signifies the nodes representing program statements, whereas E signifies the edges representing the dependencies among these statements and G is the required Graph obtained from the Nodes and edges.
Graphs are highly valuable structures in programming, as many computer science problems can be effectively represented and solved using various graph techniques. Moreover, even if not directly employing graphs, adopting a graph-based approach to problem-solving and modeling can enhance task clarity and efficiency [2].

There exist multiple methods of graph representation, each carrying its own set of advantages and drawbacks. Certain situations or algorithms that utilize graphs as input may require one representation over another. Graphical approaches involve illustrating the system's structure, and facilitating the testing of the logical progression of the program's development [3].

2 Program Representation Using Graphs.

In computer science, a graph serves as an abstract data type designed to embody the concepts of undirected and directed graphs derived from graph theory in mathematics. [2]. Graphs come in many different flavors, many of which have found uses in computer programs.
The choice of internal program representation within a software development environment significantly influences the nature of that environment.

A graph illustrates the control flow among statements within a program, while a dependence graph represents program functionalities and relationships among various entities [4].
Numerous graphical representations, such as the System Dependence Graph (SDG), Program Dependence Graph (PDG), Data Flow Graph (DFG), Extended System Dependence Graph (ESDG), Call-based Object-Oriented System Dependence Graph (COSDG), and others, serve this purpose. These graphical presentations of the program have been categorized into three distinct parts based on fundamental characteristics. Figure 1 demonstrates the Control Flow Graph, Program Dependency Graph, and System Dependency Graph.

2.1 Control Flow Graph (CFG)

A Control Flow Graph (CFG) offers a graphical depiction of the control flow or computational pathways executed within program implementations. Widely employed in static analysis and compiler applications, CFGs facilitate precise representations of program flow. They enable the recognition of syntactic structures like loops by encapsulating information for each basic block [4].The control flow graph consists of nodes linked to basic blocks (groups of statements in a program) and edges connecting these nodes to depict control flow paths [4]. A control-flow graph (CFG) illustrates the flow of control between the basic blocks in a program. It's a directed graph denoted as G = (N, E), where each node n ∈ N represents a basic

block, and each edge e = (ni, nj) ∈ E indicates a potential transfer of control from block ni to block nj.

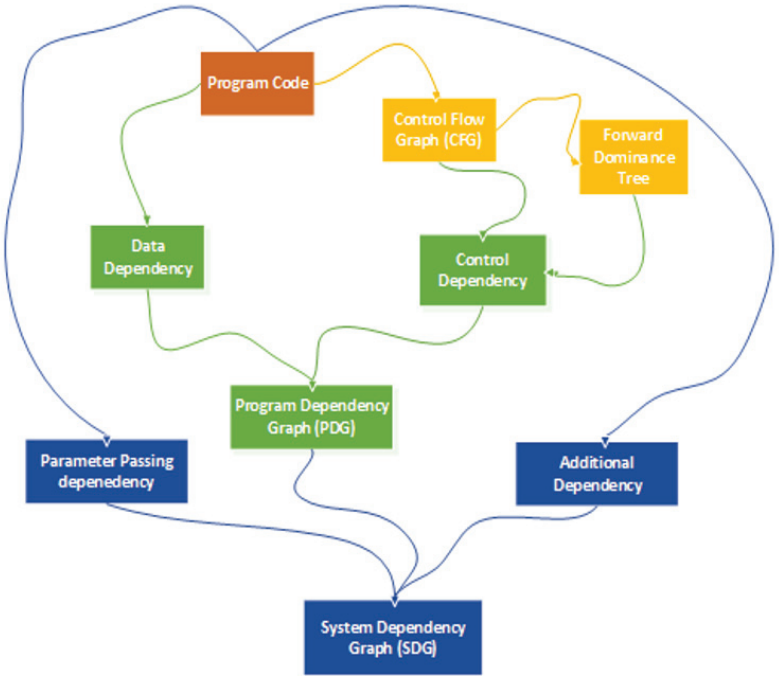

Fig .1. Program Presentation using Graph Models

Accordingly, a control flow graph contains all nodes and flows present in a flow diagram [19] as illustrated in Figure 2, such as the start and end nodes.

2.2 Program Dependency Graph

Program dependence analysis stands as a fundamental technique for comprehending programs and finds widespread applications in software testing and debugging endeavors [20]. The Program Dependency Graph portrays a program as a graph where nodes denote statements and logical expressions. The edges connected to a node signify data values pertinent to the node's operations, along with control variables that impact the node's ability to execute its functions. [5]. In a PDG, data, and control dependencies can be represented simultaneously.

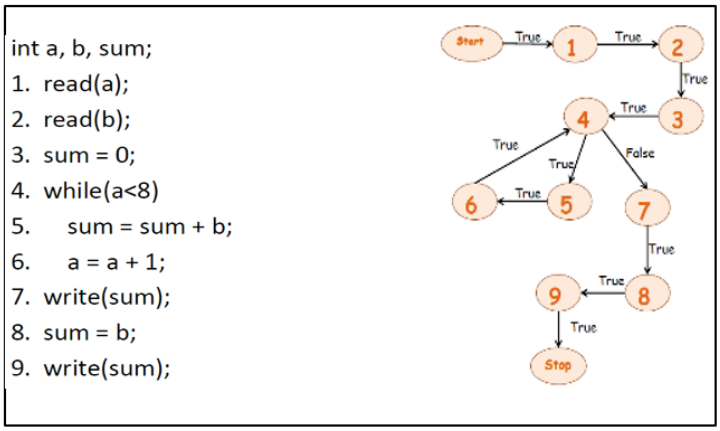

Fig .2. Call Flow Graph Presentation

The PDG distinctly delineates both data and control dependencies for each operation within a program. Data dependence graphs offer precise representations of the definition-use relationships that are implicitly embedded in a source program, aiding specific optimizing compilers. As an example, referring to Figure 2 makes use of a code fragment to compute the factorial of a number. In statement 9, there is a control predicate that determines how statements 11 and 12 will be performed. Data dependencies can be found between statement 11 and statements 7, 8, and 12. In this figure, the displayed Program Dependence Graph features the essential control dependence edges highlighted as emphasized lines, whereas data dependence edges are depicted by light-colored straight lines.

System Dependence Graph

System Dependence Graphs (SDGs) describe programs constructed using various methods and incorporating procedural calls, originating from Program Dependency Graphs (PDGs). It represents a language where parameters are passed by value, and a complete system comprises a main program and auxiliary procedures. [6].

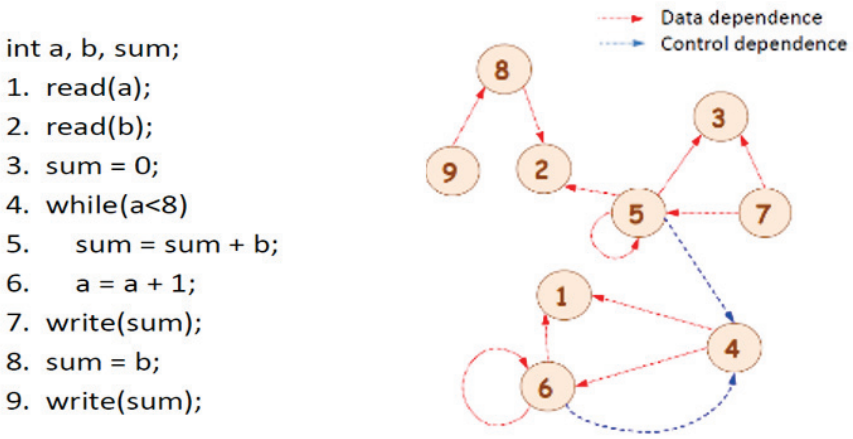

Fig .3. Program Dependence Graph Presentation

Figure 4. illustrates a program for finding the sum of numbers from 1 to 10. The main method () and the add method () are applied in this program. The System Dependence Graph of the required program can also be viewed in Figure 4, which illustrates the flow between two methods in the program. The Program Dependency Graph illustrates the flow of a single method, whereas the System Dependence Graph can display a number of procedures simultaneously.

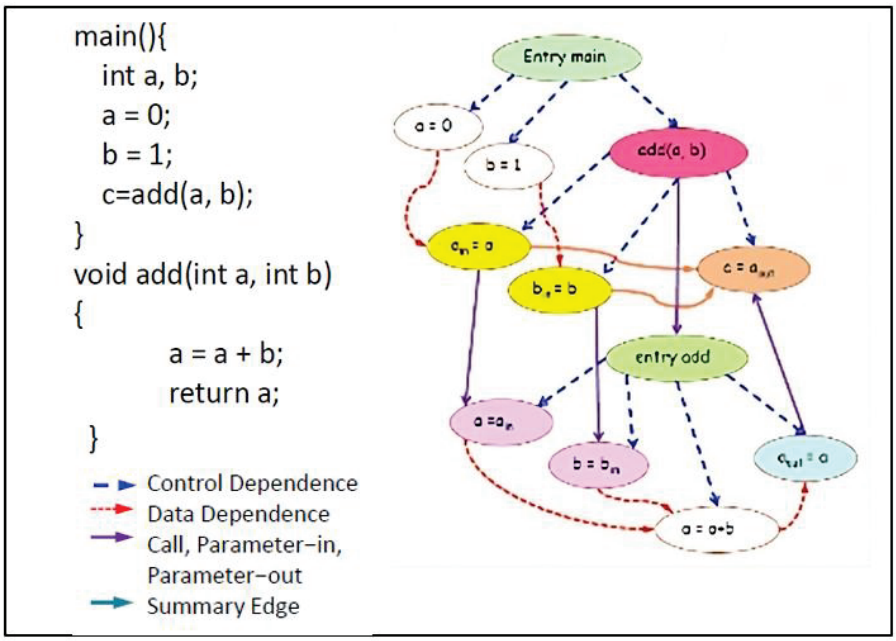

Fig .4. The System Dependence Graph for the example program

3 Comparison of Flow Graph & Dependence Graphs

Incorporating parameters such as sensitivity, handling, procedural calls, slices, exceptions, and generation of test cases, a comprehensive analysis of Program Dependence Graphs (PDGs), Control Flow Graphs (CFGs), and System Dependence Graphs (SDGs) has been carried out in table 1. [5].

Table 1 Comparison of CFG, PDG and SDG

S No.	Parameters	CFG	PDG	SDG
1	Control Dependency	Yes	Yes	Yes
2	Data Dependency	No	Yes	Yes
3	Transitive Dependency	No	No	Yes
4	Single Procedure	Yes	Yes	Yes
5	Multiple Procedures	No	No	Yes
6	Intra-procedural Calls	No	Yes	Yes
7	Inter-procedural Calls	No	No	Yes
8	Multiple types of Edges	No	Yes	Yes
9	Slicing	No	Yes	Yes
10	Flow-Sensitive	Yes	Yes	No
11	Context-Sensitive	No	No	Yes
12	Inheritance & Polymorphism	No	No	Yes
13	Dynamic Binding	No	No	No
14	Test Case Generation	Yes	Yes	Yes
15	Exception Handling	No	Yes	No
16	Parameter Passing	No	No	Yes

4 Object Oriented graph representation

The object-oriented approach integrates unique characteristics such as polymorphism, encapsulation, inheritance, and dynamic binding. The fusion of these attributes gives rise to complex relationships among program entities, resulting in additional types of dependencies between classes beyond traditional data, call, and function dependencies [15]. The importance of considering and modeling each of these dependencies cannot be overstated [11]. Coverage analysis for operational programs has been conducted utilizing models such as CG, CFG, DUG, DFG, and their improved versions, which solely depict the control flow, data flow, and calling relationships. Similarly, the SDG and its variants only portray data, control, and call dependencies. Nevertheless, there is a lack of representation of the intricate connections between classes, and the resulting dependencies that ensue from them. [13].

In object-oriented programming, a system is envisioned as a network of interconnected objects. Objects are often exchanged in large numbers of messages, and methods are generally small and well-described. This results in more complexity

moving from intra-method interfaces to inter-method interfaces and inter-class interfaces. Object-oriented programs are associated with new fault hazards, according to an analysis of their fault models [1]. The execution of the wrong target method, sending messages to the wrong class, and the interactions between superclasses and subclasses are some of the reasons why faults can occur [13]. The graph representation has been characterized based on partial and fully Object Oriented and shown in Figure 5.

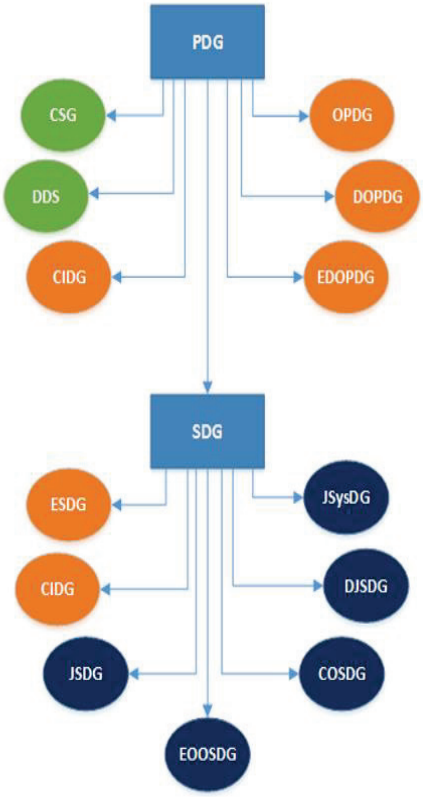

Fig .5. PDG, SDG and their sub-graphs

4.1 Partial Object-Oriented Graph Representation Related studies

The Ottestein and Ottenstein definition of slicing as a graph accessibility issue was the first to identify it. In their method, they calculate static slices of programs by using a program dependence graph. The drawback of this strategy is that it is only useful for single-method programs. A PDG is incapable of managing function calls and inter-method communication in a program with multiple methods [6]. A CDS elucidates the static control dependence relationships present within and among the various methods of a class. The DDS outlines the data dependence relationship among the statements and predicates of the program [4]. The object-oriented program

dependence graph (OPDG) was introduced by Krishnaswamy as an alternative dependence-based representation. It represents a program's control dependencies, control flow, and data dependencies. In an object-oriented program, the OPDG is composed of three tiers, namely: Data Dependence Subgraph (DDS), Control Dependence Subgraph (CDS), and Class Hierarchy Subgraph (CHS). OPDGs are generated as classes are assembled in object-oriented programs, containing all class representations [5].

Xu, B al et 2002 introduced a Dynamic Object Program Graph by adding dynamic slicing. In this approach, forward analysis is integrated with backward analysis. During the forward activity, it records details on the object program dependency graph (OPDG) and calculates dynamic slices (utilized to maintain dynamic execution information) as required. For obtaining the final version of dynamic slices, it traverses the OPDG highlighted in the backward process [7].

According to Horwitz et al. [8], system dependence graph (SDG) can be used as an intermediate representation of a program and a two-phase graph reachability algorithm can be developed based on the SDG to determine an intermediate slice between two procedures.Interprocedurally slicing traverses backwards along every edge of the SDG other than parameter-out edges, and highlights the vertices encountered. In the second pass, all edges excluding call and parameter-in edges are traversed backward from all vertices highlighted previously [8]. Slices are obtained by unioning the vertices identified by passes one and two. C++ programs with a partial object-oriented design, have been incorporated in current slicing approaches based on system dependence graphs.

4.2 Fully Object-Oriented Graph Representation Related studies

Polymorphism, Data hiding, inheritance, etc. are all aspects of object-oriented programming that are included in ESDG. Each method is depicted by the PDG, with a method entry vertex signifying the entry point into the method. Additionally, a class entry vertex is included, which is connected to the method entry vertex of each method in the class by a class edge [13].

As part of their extension of the SDG, Larson and Harrold [12] used object-oriented programs to illustrate the SDG. For every class in an object-oriented program, they created Class Dependence Graphs (ClDGs). With a ClDG, it is possible to determine how a class is controlled and how data is dependent on it, without having to know the environment where the class is called. Procedure dependence graphs are used to represent each method in a ClDG.

As an extension to earlier dependency-based representations, the Java Software Dependency Graph (JSDG) describes numerous variations of program dependencies [16]. The JSDG is composed of a group of dependence graphs, describing Java methods, classes, interfaces, and packages accordingly. It is feasible to construct Java software engineering tools based on the JSDG [16].

The Java System Dependence Graph (J Sys DG) was enhanced by Walkinshaw et al. [22]. The J Sys DG comprises a collection of graphs illustrating the control and dependency relationships among Java statements. It is a multigraph delineating both control and data dependencies within the statements of a Java program. Statements are classified based on whether they contribute to the program's structure (e.g., headers representing methods, classes, interfaces, and packages) or its behavior (e.g.,

those within a method body). [18]
In a Java program, we observe that methods align with procedures. We exclusively focus on those methods and statements that correlate with the method execution trace. As we collect method execution traces containing all possible execution paths at the statement level. Therefore, the DJSDG must encompass all details regarding the relationships between each statement vertex that is contained within an executed method [19]. The DJSDG does not account for class inheritances, overrides, or overloads. As a result, the DJSDG is simpler compared to the JSDG. An intermediate representation is built using a Dynamic Java System Dependence Graph (DJSDG), upon which slice computations are performed [19].

The COSDG has been derived from the ESDG. As in ESDG, the classes in a COSDG are displayed by a class dependence graph, excluding those features that are not relevant to assess test coverage. As a result, inherited methods are incorporated into the representation and are represented differently when calling polymorphic and inherited methods. In order to maximize the efficiency of polymorphic and inheritance coverage measurements for object-oriented programs, these changes have been made to COSDG [14].

An Extended Object-Oriented System Dependence Graph has been designed to simplify the process of applying object-oriented regression testing by employing hierarchical decomposition slicing. It was suggested by Tao et al. [15], to retain separate graphs for statements, methods classes and packages, regardless of whether they were affected by the change, even though this does not result in an additional space requirement for EOOSDG. A number of additional dependencies were considered, such as type dependence and read/write dependence and package membership dependence in order to take into account more aspects of Java programs and determine correct slices [15].

Table 2 Description of Graphs

S.No	Acronym	Abbreviation	Description	Object Oriented
1	PDG	Program Dependence Graph	Control + Data dependencies for single procedure	No
2	CDS	Control Dependence Sub-graph	Control dependencies for single procedure	No
3	DDS	Control Dependence Sub-graph	Data dependencies for single procedure	No
4	OPDG	Object Program Dependence Graph	PDG + Object-Oriented Features	Partial
5	DOPDG	Dynamic Object Program Dependence Graph	OPDG + Dynamic Slicing	Partial
6	EDOPDG	Efficient Dynamic Object Program Dependence Graph	DOPDG + few modifications	Partial
7	SDG	System Dependence Graph	Set of PDGs + Interprocedural calls for whole sys.	Partial

8	ClDG	Class Dependence Graph	Set of PDGs + Inter-procedural calls within class	Partial
9	ESDG	Extended System Dependence Graph	Extends SDG by representing a class with ClDG	Partial
10	JSDG	Java System Dependence Graph	SDG + Interfaces& Packages in Java	Fully (10 dependency)
11	JSysDG	Java System Dependence Graph	JSDG +few modifications	Fully (9 dependency)
10	DJSDG	Dynamic Java System Dependence Graph	JSDG + Dynamic Slicing	Fully (9 dependency)
11	COSDG	Call-based Object-oriented System Dependence Graph	Dependencies + Flow + Call Graph + Inherited Call + Polymorphic calls	Fully (9 dependency)
12	*EOOSDG*	Extended Object-Oriented System Dependence Graph	ESDG +additional Dependency	Fully (14 dependency)

5 Analysis and Discussion

A fully object-oriented graph representation involves organizing and modeling a graph using principles of object-oriented programming (OOP). In a graph, nodes represent entities, and edges represent relationships between these entities. Object-oriented graph representation leverages OOP concepts like classes and objects to create a flexible and modular structure for graph modeling.

The first graph representation which fully supports object-oriented programming was labelled Java System Dependence Graph (JSDG). Feature representations specific to Java, including inheritance, interfaces, single, and packages could be supported by it. The J System extended its capabilities by incorporating multiple graph representations, surpassing previous methodologies. It introduces a refined depiction of call sites, particularly those involving polymorphic objects either as callers or parameters. This enhanced representation enables precise slicing of data members belonging to an object variable utilized within a method or defined by it [21]. A JSysDG is a multiple graphs that illustrates the control and data dependencies among the statements of a Java program. One of the most sophisticated program representations utilized for slicing object-oriented programs is the Java System Dependence Graph (JSysDG). This graph is capable of representing packages, interfaces, classes, and methods, providing robust support for polymorphism, dynamic binding, and inheritance. [21]. A coarse-grained dynamic slice approach has been presented by Xi et al. for Java programs [3][23]. For tracing method executions, the technique uses aspects of AspectJ code, which contains information regarding method calls [3]. The intermediate representation was constructed along with the slicing procedure using the Dynamic Java System Dependence Graph (DJSDG).

As a way to analyze test coverage using internal representations, COSDG represents object-oriented programs according to dependencies. Incorporating fundamental aspects like control flow, data flow, and method calls, COSDG (Coverage-Oriented Software Dependency Graph) goes a step further by illustrating

object-oriented characteristics like polymorphism inheritance and classes. Notably, in representing polymorphic calls and returns, COSDG avoids the addition of specific vertices for polymorphic calls and edges for returns [22]. COSDG (Coverage-Oriented Software Dependency Graph) does not rely on region vertices or control flow edges to explicitly depict control flow in program constructs. This omission is intentional, as these elements are deemed irrelevant to the specific goal of coverage analysis. Instead, our representation focuses on capturing the coverage of different program elements, such as loops, blocks, and function calls, within a test suite. The emphasis lies on assessing the extent to which these elements are covered rather than quantifying the frequency of their execution [14].

Panda (2016) introduced the Extended Object-Oriented System Dependency Graph (EOOSDG) in the previous architectural model [15]. A transitive approach eliminates the superfluous edges because EOOSDG takes longer to identify the necessary nodes for program slicing [15]. The elimination of further redundant edges may result in the loss of vital information from duplicated edges. Cost redundancy is another problem raised by EOOSDG graph representation because there are too many redundant edges. There is a need to filter it using some technique, such as transitivity. Transitivity is a technique that filters the redundant edges by ignoring edges that are less important than others. This technique can be used to reduce graph complexity and make it more efficient. It also helps to identify the most important nodes in a graph. The transitive technique reduces the cost of redundant edges but raises another important factor, the effect of information lost about these edges, which may have a semantic effect on other nodes.

6. Conclusion

In a fully object-oriented graph representation, the graph is arranged and modeled according to the principles of object-oriented programming (OOP). Based on the literature, it has been established that the Java system dependence graph (JSDG) is an extremely rich features graph representation model compared to other fully object-oriented graph representation models. Implementation of JSDG considers all possible relationships in Java whose dependencies are defined in the OO programs. As a result, the Java system dependence graph (JSDG) has been proposed as a potential model for presenting object-oriented programming.

Reference

1. Joyner, W. D., & Melles, C. G. (2017). Introduction: Graphs—Basic Definitions. In Adventures in Graph Theory (pp. 1-39). Birkhäuser, Cham.
2. Wallis, W. D. (2010). A beginner's guide to graph theory. Springer Science & Business Media.
3. Arora, V., Bhatia, R. K., & Singh, M. (2012). Evaluation of flow graph and dependence graphs for program representation. International Journal of Computer Applications, 56(14).
4. Allen, F. E. (1970). Control flow analysis. ACM Sigplan Notices, 5(7), 1-19.

5. J. Ferrante, J. Worren and K. Ottenstein, “The Program Dependence Graph and its use in optimization”, In ACM Transactions on Programming Languages and Systems, vol. 9, no. 3, pp. 319-349, July 1987.
6. S. Horwitz, T. Reps and D. Binkley, “Interprocedural slicing using dependence graphs,” In ACM Transactions on Programming Languages and Systems, vol. 12, pp. 26-60, January 1990.
7. Arora, V., Bhatia, R. K., & Singh, M. (2012). Evaluation of flow graph and dependence graphs for program representation. International Journal of Computer Applications, 56(14).
8. Ottenstein, J. K., and Ottenstein, M. L. The Program Depenedence Graph in a Software Development Environment. ACM SIGPLAN Notices 19, 5 (1984), 177–184.
9. Mohapatra, D. P., Mall, R., & Kumar, R. (2006). An overview of slicing techniques for object-oriented programs. Informatica, 30(2).
10. Panda, S. (2016). Regression Testing of Object-Oriented Software based on Program Slicing (Doctoral dissertation).
11. Horowitz, S., Reps, T., and Binkley, D. Interprocedural Slicing using DependenceGraphs. ACM Transactions on Programming Languages and Systems 12, 1 (1990), 26–60.
12. Hon. F. Li, Juergen Rilling, and Dhrubajyoti Goswami. Granularity-driven dynamic predicate slicing algorithms for message passing systems. Automated Software Engineering 1, 11:63 – 89, 2004.
13. Najumudheen, E. S. F., Mall, R., & Samanta, D. (2011). Test coverage analysis based on an object-oriented program model. Journal of Software Maintenance and Evolution: Research and Practice, 23(7), 465-493.
14. Najumudheen, E. S. F., Mall, R., & Samanta, D. (2010). A Dependence Representation for Coverage Testing of Object-Oriented Programs. J. Object Technol., 9(4), 1-23.
15. Panda, S. (2016). Regression Testing of Object-Oriented Software based on Program Slicing (Doctoral dissertation).
16. Zhao, J. (1998, December). Applying program dependence analysis to Java software. In Proceedings of Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium (pp. 162-169).
17. Xi, L., Li, M., Dan, Z., & Wei, L. (2011, May). An approach of coarse-grained dynamic slice for Java program. In 2011 IEEE 3rd International Conference on Communication Software and Networks (pp. 670-674). IEEE.
18. Walkinshaw, N., Roper, M., & Wood, M. (2003, September). The Java system dependence graph. In Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation (pp. 55-64). IEEE.
19. https://www.sciencedirect.com/topics/computer-science/control-flow-graph (Accessed on 28 February 2024).
20. Shu, G., Sun, B., Henderson, T. A., & Podgurski, A. (2013, March). Javapdg: A new platform for program dependence analysis. In 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (pp. 408-415). IEEE.

21. Galindo, C., Pérez, S., & Silva, J. (2023). Program slicing of Java programs. Journal of Logical and Algebraic Methods in Programming, 130, 100826.
22. Najumudheen, E. S. F., Mall, R., & Samanta, D. (2019, February). Modeling and coverage analysis of programs with exception handling. In Proceedings of the 12th Innovations on Software Engineering Conference (formerly known as India Software Engineering Conference) (pp. 1-11).
23. Mishra, A., Kabat, M. R., & Mohapatra, D. P. (2023, March). Dynamic Program Slicing of Agent-oriented Software. In 2023 2nd International Conference for Innovation in Technology (INOCON) (pp. 1-7). IEEE.