OPEN ACCESS JOURNALS
Indiana Journal of Multidisciplinary Research
Abbreviated Tittle- Ind J Multi Res
ISSN *(Online)*- 2583-3820
Journal Homepage Link- https://indianapublications.com/Journals/IJMR
DOI: https://doi.org/10.5281/zenodo.21559665

INDIANA PUBLICATIONS
PRODUCTIVE AND QUALITY RESEARCH

Research Article

Volume-06|Issue-04|2026

An Innovative Approach for Abstracting Program Slices from the Java Programming System

Aparna K. S[1] and Dr. R. N. Kulkarni[2]

[1]Assistant Professor, Department of Computer science & Engineering, Global Academy of Technology, Bangalore, Karnataka, India.
aparna.vastrad@gmail.com
[2]Professor and HOD, Department of Computer Science & Engineering, Ballari Institute of Technology and Management, Ballari, Karnataka, India.
rnkulkarni@bitm.edu.in

Article History
Received: 05.05.2026
Accepted: 22.06.2026
Published: 25.07.2026

Citation
Aparna, K. S. & Kulkarni, R. N. (2026). An Innovative Approach for Abstracting Program Slices from the Java Programming System. *Indiana Journal of Multidisciplinary Research, 6*(4), 75-83.

Abstract: In recent decades, the software and hardware industries have undergone significant transformation. Many organizations have adopted Java-based applications—either developed internally or sourced from third parties—to automate their day-to-day operations. Over time, these applications have been frequently updated to align with evolving business requirements. However, the original developers are often no longer available, and the continuous modifications have led to poorly structured systems. As a result, understanding and migrating these legacy applications to modern paradigms has become increasingly difficult for new developers. These systems also encapsulate critical business rules accumulated over years, which must be retained and extended during migration efforts. To tackle these issues, a novel approach is introduced for abstracting key components from existing Java programs, enabling a smoother transition to newer technologies or paradigms. The proposed methodology begins by restructuring the Java source code, followed by the abstraction of design elements in the form of control flow and data flow representations. Using this design data, functional dependencies are extracted and minimized to derive relevant program slices. These slices are then precisely represented using Object-Z notation, offering a clear and formal specification of the software behavior and user requirements

Keywords: Restructuring, Data flow graph, Control flow graph, Referred Variable, Defined variable, functional dependency, program slice.

Copyright © 2026 The Author(s): This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (**CC BY-NC 4.0**).

INTRODUCTION

In recent decades, a number of industries have relied heavily on automation to streamline operations and enhance service efficiency in order to meet rising demand and supply targets [1]. The rising need for software solutions, ideally employing object-oriented programming languages like Java, C++, etc., was further increased by this change in organizational operations [2]. These software programs have had multiple need-based modifications throughout time to satisfy changing business needs as the enterprises have changed. However, the program had undergone significant changes as a result of these small adjustments, becoming more sophisticated, unstructured, and challenging to maintain [3]. In particular, when the original design specifications and developers are no longer available, developers frequently find it difficult to comprehend and update these software systems. Consequently, one of the most important challenges in software engineering today is deciphering the code and moving older software programs to a new, updated paradigm while maintaining their internal business logics[5]. It's crucial to remember that although Java systems are still dependable, they are unable to keep up with the latest developments in computing technology. Their readability and comprehension have deteriorated over time due to frequent upgrades and poor maintenance, making them more challenging to handle. Furthermore, determining software design standards has emerged as a major obstacle in fulfilling contemporary software development requirement [6][7].

In order to successfully extract the essential functional characteristics and software design requirements, software engineers must develop a unique technique for re-engineering these systems. To facilitate the organized transition of Java systems to modern technologies, the main objective is to separate functionally independent modules and extract abstract functionalities [8]-[10]. Therefore, it becomes necessary to build methods that can systematically restructure and analyse Java programs to improve their maintainability and aid in the derivation of important design specifications from various program abstraction levels in order to handle the aforementioned issues.

MATERIALS

***Corresponding Author:** Aparna K. S

In the first stage, the suggested framework's working flow incorporates reorganizing the input Java program, while removing superfluous elements, extracting the design information analysing the control flow and data flow, in the form of control flow and data flow tables. In order to lower computing costs and facilitate the creation of program slices that operate as formal software specifications, the framework makes use of this design abstraction technique to further identify and refine the key functional dependencies, in order to develop specifications, the software slices are further identified and represented in formal notations. To calculate the functional relationships at a lower and more efficient cost, the suggested re-engineering framework additionally integrates computational procedures for re-structuring, design information extraction, and data flow table (DFT) computation. The functional dependences are then used to calculate the minimal attributes that affect the results of program slicing for formal software specification extraction using a minimal cover algorithm (MCA). Static-backward slicing techniques are essential to the suggested framework's successful derivation of formal software requirements and program abstraction. Here, the methods essentially examine the dependencies and extract pertinent Java program code. Refactoring plays a fundamental role in software engineering by enhancing maintainability, reducing technical debt, and supporting long-term software evolution. Nandini et al.[11] conducted a tertiary systematic literature review that examined two decades of research on code smells and refactoring practices. While the study provides valuable insights into patterns, tools, and trends, it lacks integration of program slicing, which is crucial for preserving data and control dependencies during refactoring. In parallel, Cordeiro et al.[12] performed one of the first empirical evaluations of large language models (LLMs) for automated refactoring, demonstrating that GPT-like systems can perform syntactically and semantically correct transformations. However, their success depends heavily on prompt formulation, code complexity, and contextual limits, restricting consistent applicability in large-scale software systems. Alakula et al.[13] proposed enhance software comprehension through interactive visualization and dependency exploration. Despite their strengths in dynamic analysis and comprehension support, these methods do not formalize functional dependencies or compute minimal covers, which are essential for optimizing slicing. Similarly, Riouak et al.[14] developed IntraJ, an AST-based Java analysis framework for intraprocedural dependency exploration, and Hejderup et al.[15] proposed to improve call-based dependency precision. Both, however, are limited by the lack of intraprocedural capability and minimal dependency abstraction. Program slicing's main goal is to remove superfluous statements from source codes while maintaining the Java program's semantics. It is also essential for analyzing Java programs' functional dependencies [16][17]. A pair of (p,V), where p is the program and V is a subset of program variables, can be referred to as a slicing criterion. Numerous software tasks, such as software testing, debugging, re-engineering, program analysis, and comprehension, have made extensive use of this program slicing technique [18], as it facilitates comprehension of a program's structural elements without needing execution, the concept of static program slicing is still used to analyse functional dependencies in Java applications. This makes it a highly useful tool for re-engineering. Its ability to recognize control and data linkages aids in the efficient derivation of software design and requirement specifications as well as the refactoring of unstructured Java code into modular and manageable components [19]. Static slicing is less dependent on real execution traces and more general test cases than dynamic slicing, and it is computationally cheap. Additionally, static slicing techniques facilitate security analysis, debugging, and performance optimization in addition to making program maintenance easier [20]. Due of the high computational cost of analysis larger programs, traditional slicing techniques typically do not guarantee an acceptable scalability factor. In addition to addressing precision vs. performance trade-offs, which are still important and unresolved for real-time applications, these techniques mostly suffer from memory and performance limitations. The dynamic slicing techniques depend on particular execution traces, it is challenging to cover every potential program behaviour, particularly in concurrent and multi-threaded systems A significant body of research has focused on program slicing as a means to isolate relevant program components while preserving behavioural semantics. Stoica et al.[21], Seghir et al.[22], and Vezina's et al.[23] highlighted various static, dynamic, and hybrid slicing approaches, noting trade-offs between computational efficiency and slicing precision. Although hybrid and symbolic methods improve recovery of relevant program statements, they often suffer from runtime overhead or scalability constraints in large Java applications. Recent machine learning–based slicing techniques. Yadavally et al.[24] Soremekun et al.[25], Shahandashti et al.[26] have demonstrated improved dependency handling but remain dependent on execution data, which limits their use in contexts where such data is incomplete or unavailable. The prime motive of this work is to comprehend the given Java input code and extract formal specifications from the executable program for effective re-engineering and enhancing software performance. The re-engineering framework integrates a set of components to enhance the requirements abstraction and system modularity that includes the following stages In the domain of formal specification and requirement abstraction, several studies emphasize the role of slicing in deriving verifiable software models. Vistein et al.[27]), Ferrarotti et al[28], and Miskell et al[29] proposed frameworks that transform program slices into formal specifications to enhance traceability, verification, and comprehension. Complementary studies by Shivram and Handigund[30] advanced methods for extracting object-oriented features and

***Corresponding Author:** Aparna K. S

reusable components from requirement specifications, strengthening the linkage between natural language requirements and formalized software models. Collectively, these studies underscore the growing convergence between refactoring, slicing, and formal specification research. However, a clear research gap persists in integrating slicing-based dependency abstraction with formal requirement extraction to achieve behaviour-preserving and computationally efficient software re-engineering. Addressing this gap requires a unified framework that balances precision and performance while ensuring semantic traceability from code-level structures to formal requirement specifications.

METHODS

The proposed methodology focuses on deriving the software specifications and user requirements from the program slices, which are represented using the precise mathematical notation. These slices are derived from data flow tables, using the minimal cover variables. These minimal attributes are abstracted from the functional dependencies, extracted from the control flow and data flow tables. Existing methods typically represent input programs using graphical structures to extract the design information. However, these graphical representations introduce significant overhead in terms of memory usage and maintenance. As program complexity increases, the number of nodes and edges in the graph also grows proportionally. Representing such large-scale graphs in memory, especially in matrix form, becomes increasingly challenging. The matrix size escalates with the number of program statements, making it difficult to manage, store, and analyse. In existing approaches, these graphical approaches are used to model control flow analysis, but they require considerable space and processing time. The nodes in these graphs represent computation statements, function calls, or control structures (if, while, for, switch etc) of the given input Java program. The number of edges is influenced by the cyclomatic complexities and by the basic Java features like inheritance, message passing, and object interaction. Hence for a directed graph with V vertices, the maximum number of edges E can be given by the formula Emax = V(V - 1). The time complexity for analysing such graphs is O(V + E). Moreover, during additional operations like depth-first search (DFS) and breadth-first search (BFS) used for traversing the graph require $O(V^2)$ time in the worst case. These complexities highlight the inefficiencies in graph-based approaches, especially for large or deeply nested programs. To address these limitations in space and time, the proposed methodology utilizes tabular representations instead of graphical models. In our approach, the input programs are first restructured (as discussed in detail in Section 2). The restructured program is then statically traced, and control flow is represented in a Control Flow Table (CFT) and data flow tables. These steps are elaborated in detail in the following sections.

Restructuring the Input Program: Restructuring refers to the process of modifying the structure of a software program without altering its functionality. It is a critical preliminary step in the proposed methodology to prepare the code for static analysis and slicing. The restructuring phase involves the following key operations:

Algorithm for restructuring

1. Elimination of Comment and Blank Lines: All non-executable comment lines and empty lines are removed to streamline the code.
2. conversion of Multiline Statements to Single-Line Statements: Statements spanning multiple lines are merged into a single line to simplify parsing and analysis.
3. Separation of Multiple Statements in a Line: If a single line contains multiple executable statements, each is separated into individual lines for clarity.
4. Line Numbering: Each executable statement is assigned a unique line number to aid in tracking control and data flow.
5. In lining of User-Defined Functions: If the program references user-defined functions from external classes or imported packages, the relevant function definitions are substituted directly into the main program.
6. The resulting restructured program is then used for control flow analysis, as described in Section 2.2.

Abstraction of Control Flow Information from the Restructured Java Program: Once the Javan program has been restructured, the next step involves extracting its control flow information. This is achieved by statically analyzing the program and capturing control constructs and function calls in a structured Control Flow Table (CFT). The CFT provides a clear and concise representation of the execution flow within the program, as explained in the results section.

Algorithm: Control Flow Table Construction:

Input: Restructured Java program file Output: Control Flow Table (CFT)

- Step 1: Begin analysis from the first executable line of the program. Initialize the Start column (Start = 1), representing the starting line number.
- Step 2: Scan the program from the main () function. Identify: Control Structures (e.g., if, while, for, switch, etc.) and Function/Method Invocations. Based on the discovery, If any control statement is encountered, then Update the End column with its line number. Update the Trans1 column with the line number immediately following the opening brace {(i.e., the start of the true block). Update the Trans2 column with the line number where the corresponding closing brace} appears (i.e., the start of the false block or the continuation after

***Corresponding Author:** Aparna K. S

the control structure). If a function call is encountered, Set Trans1 to the line number where the called function begins. Set Trans2 to the line number where the execution resumes in the calling function after the invocation.

- Step 3: Recursively analyse the blocks referenced in Trans1 (true block) and Trans2 (false block) , following the same procedure outlined in Steps 1 and 2 until each block concludes.
- Step 4: Continue repeating Steps 1–3 recursively for all control statements and function calls within the main () function until the entire program has been analysed. The constructed Control Flow Table serves as the foundation for the subsequent step—data flow analysis, which is described in Section 2.3.

Abstraction of the Data Flow Table in Control Flow Order: The data flow within a program is closely tied to its control flow. Variables may be defined (assigned a value) and referred (used) at various points throughout the program. To capture this information accurately, the Data Flow Table (DFT) is constructed by following the control flow order defined in the Control Flow Table (CFT). This ensures that the program execution path is respected during data flow analysis. The DFT records which variables are defined and referred on each line of the program, thereby enabling a detailed examination of variable usage and dependencies.

Algorithm: Data Flow Table Construction

Input: Restructured Java Program, Control Flow Table (CFT) Output: Data Flow Table (DFT):

- Step 1: Traverse the program sequentially in the order specified by the Control Flow Table.
- Step 2: For each line of the program, identify variables that are defined (i.e., assigned a value). Identify variables that are referred to (i.e., used in expressions, conditions, or function arguments).
- Step 3: Update the corresponding row in the Data Flow Table with the identified defined and referred variables for each line, The constructed DFT provides a foundation for abstracting functional dependencies, which are described in the following section.

Abstraction of Functional Dependencies from the Data Flow Table: In traditional approaches, functional dependencies (FDs) are often derived using tree-based analysis of the program structure. In our proposed method, we extract functional dependencies directly from the Data Flow Table by identifying relationships between defined and referred variables. A functional dependency exists when the value of one or more variables (referred variables) depends on the value of other variables (defined variables). By capturing such relationships from the DFT, we can infer the functional behavior of the program without relying on complex tree traversals.

Algorithm: Functional Dependency Extraction

Input: Data Flow Table (DFT), Restructured Java Program Output: Functional Dependencies (FDs)

- Step 1: Scan the Data Flow Table to identify rows that contain both defined and referred variables.
- Step 2: For each such row, extract the line number from the DFT. Retrieve the corresponding statement from the restructured Java program using the line number. Record the dependency in the form:
- Defined Variable(s) → Referred Variable(s).
- Step 3: Note that, due to contributions from multiple developers and multiple revisions of the software, these dependencies may include redundant or duplicate variables. These redundancies are addressed in the next step through minimization, described in Section 2.5.

Finding the Minimal Cover of Functional Dependencies: After extracting the functional dependencies (FDs) from the Data Flow Table (DFT), the next step is to minimize them by deriving their minimal cover. A minimal cover is a simplified set of FDs that preserves the original dependency relationships but eliminates redundancy. This process ensures optimal slicing by focusing only on essential variable relationships.

Algorithm: Minimal Cover of Functional Dependencies:

Input: Functional Dependencies (FDs) from the DFT, Output: Minimal Cover Set of Functional Dependencies.

1. Step 1: Arrange the FDs in descending order based on the number of attributes on the left-hand side (LHS), to bring them into canonical form.
   For eg:

- B D H → G
- G I → N
- B → H

2. Step 2: Remove extraneous attributes from the LHS of each FD: Compute the closure of each attribute set on the LHS. If the closure includes more attributes than necessary to determine the RHS, remove the extraneous attributes. For eg:

- B → G
- G → N
- B → H

3. Step 3: Eliminate redundant FDs that can be inferred through transitivity.

4. Step 4: Apply Armstrong Axioms (reflexivity, augmentation, and transitivity) to simplify and merge FDs wherever possible. Perform union operations to combine multiple FDs with the same LHS. Ensure that the resulting set of FDs is minimal, non-redundant, and logically equivalent to the original set.

***Corresponding Author:** Aparna K. S

- B → G H D
- G → N

Step 5: The resulting minimal cover attributes are retained and used in the next step for program slicing, as they represent the critical variables that determine program behaviour.

Extracting the Backward Slice from Minimal Cover Attributes: Once the minimal cover of functional dependencies is obtained, backward slicing is performed using these attributes. The goal of slicing is to extract only those parts of the program that directly or indirectly influence the value of a selected slicing criterion variable. This process is based on Weiser slicing algorithm. The minimal attributes from the previous step are used as slicing criteria. Using the DFT, trace all lines where these attributes are either defined or referred. Starting from the line where the slicing variable is used, recursively track backward all statements that contribute to its computation. Only those statements that logically influence the slicing variable are included in the final slice. The result is a set of line numbers, which correspond to the relevant program statements in the restructured Java program. These statements form the backward slice, and non-contributing statements (e.g., unrelated variable declarations or control structures) are omitted. This process is repeated for each minimal cover attribute, generating multiple precise slices for a given program. These slices provide a formal and concise representation of program behavior and serve as a foundation for extracting software specifications. By focusing only on logically essential statements, the proposed slicing method enhances program comprehension, reduces complexity, and supports software re-engineering efforts.

RESULTS

Let us consider the example of employee, where Net-salary is computed using the basic salary components for n employees and this program is simple java program with lot of comment lines, white spaces, blank lines import statements, which are pre-processed and the restructured java program is in Table 2. The control flow and data flow analysis is derived in the form of control flow table and data flow table as shown in table 3 and 4respectively.The functional dependencies are extracted and minimized and the output is shown in Figure 2 and the backward slice is extracted as discussed in section 2.6 and the software requirements and specification is represented in Figure 5. The next step is to extract the functional dependencies from the DFT table as,

- BS → HRA,
- BS → DA,
- BS DA HRA → GS,
- GS → ITX,
- GS ITX → NS

And from these functional dependencies, finding the minimal cover using closure concept and slicing using these minimal attributes using the minimal cover algorithm as discussed above and finally removal of extraneous attributes leads to these FDs

- BS → GS,
- GS → NS,
- BS → HRA,
- BS → DA,
- GS → ITX

And after using union principle for above FDs, we have
BS → GS HRA DA,
GS → NS, ITX.

These are the final FDs and after minimization, we have the start symbol to derive minimal cover attributes BS, GS, HRA, DA, NS, ITX.

The backward slices are extracted for all these variables. The backward slices are derived by parsing from bottom up for the variables
HRA = {19, 11, 6, 3} and
ITX = {19, 14, 13, 12, 11, 6, 3}and the snapshot for the DA = {3, 12, 13, 19}is derived from the Figure 3.

Table 1: Input Java program

```
class ct_employee
{
   // Reading all the basic salary components
   private int basicSalary, HRA, DA, GS, incomeTax,
netSalary;
   public void read()
   {
      basicSalary = scan.nextInt();
      calculate();
   }
   public void calculate()
   {
      // Calculating all the parameters

      HRA = (10 * basicSalary) / 100;
      DA  = (20 * basicSalary) / 100;
      GS = basicSalary + DA + HRA;
      incomeTax = (30 * GS) / 100;
      netSalary = GS - incomeTax;
   }
   public void display()
   {
      // Displaying the calculated parameters

      System.out.println("Employee Salary Details");
      System.out.println("Basic Salary = " + basicSalary);
      System.out.println("HRA = " + HRA);
      System.out.println("DA = " + DA);
      System.out.println("GS = " + GS);
      System.out.println("Income Tax = " + incomeTax);
      System.out.println("Net Salary = " + netSalary);
```

***Corresponding Author:** Aparna K. S

```
    }
}

class Main     // Main Function
{
    public static void main(String args[])
    {
        // Reading the values of employee

        ct_employee employeeObj = new ct_employee();
        employeeObj.read();      // Calling read function
        employeeObj.display();   // Calling display function
    }
}
```

Table 2: Restructured Java Program

S. No.	Java Program
1	class employee
2	{
3	
4	private int basicSalary, HRA, DA, GS, incomeTax, netSalary;
5	public void read()
6	{
7	basicSalary = scan.nextInt();
8	calculate();
9	}
10	public void calculate() // calculating all the parameters
11	{
12	HRA = (10/100) * basicSalary;
13	DA = (73/100) * basicSalary;
14	GS = basicSalary + DA + HRA;
15	incomeTax = (30/100) * GS;
16	netSalary = GS - incomeTax;
17	}
18	public void display() // displaying the calculating parameters
19	{
20	System.out.println("Employee ID: " + employeeId + "\nEmployee Name: " + empName + "\nEmployee Basic Salary: " + basicSalary + "\nHRA: " + HRA + "\nDA: " + DA + "\nGS: " + GS + "\nIncome Tax: " + incomeTax + "\nNet Salary: " + netSalary);
21	}
22	}
23	class Main // main function
24	{
25	public static void main(String args[])
26	{
27	employee employeeObj = new employee();
28	employeeObj.read(); // calling read function
29	employeeObj.display(); // calling display function
30	}
31	}

Table 3: Control Flow Table

START	End	TRANS1	TRANS2
1	27	5	28
5	8	10	9
10	17	9	
8	29	18	30
30	31E		

Table 4: Data Flow Table

Line No	Referred Variable	Defined Variable
1		employee
2		
3		basicSalary, HRA, DA, GS, incomeTax, netsalary;
4		public void read()
5		
6		basicSalary
7		
8	calculate();	
9		
10	public void calculate()	
11		
12	basicSalary	HRA
13	basicSalary	DA
14	HRA, DA, basicSalary	GS
15	GS	IncomeTax
16	GS, incomeTax	netSalary
17		
18		public void display()
20	basicSalary, DA, HRA, GS, incomeTax, netSalary	
27		employee()
28	read()	
29	display()	

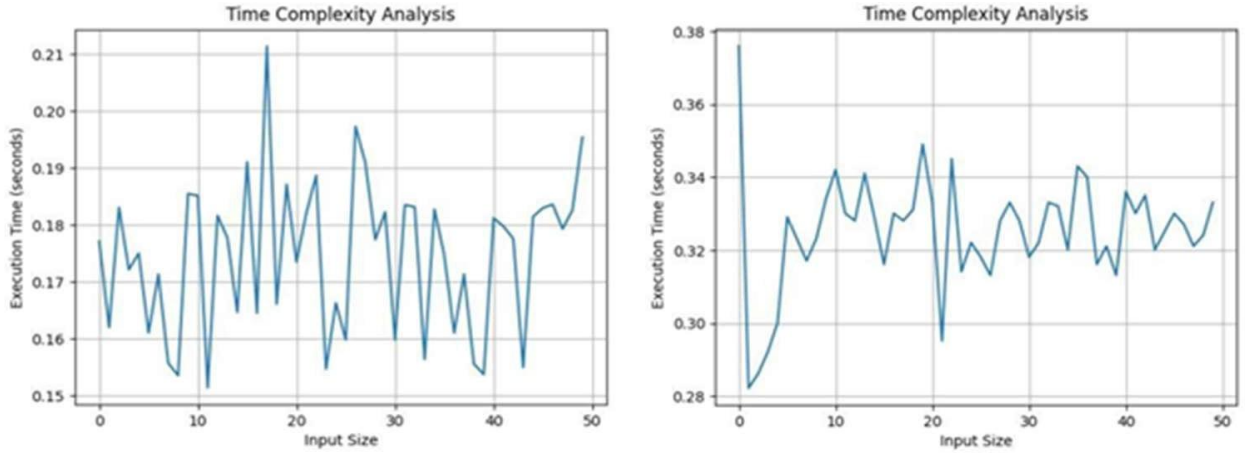

Figure 1: Comparative analysis graph before and after restructuring

***Corresponding Author:** Aparna K. S

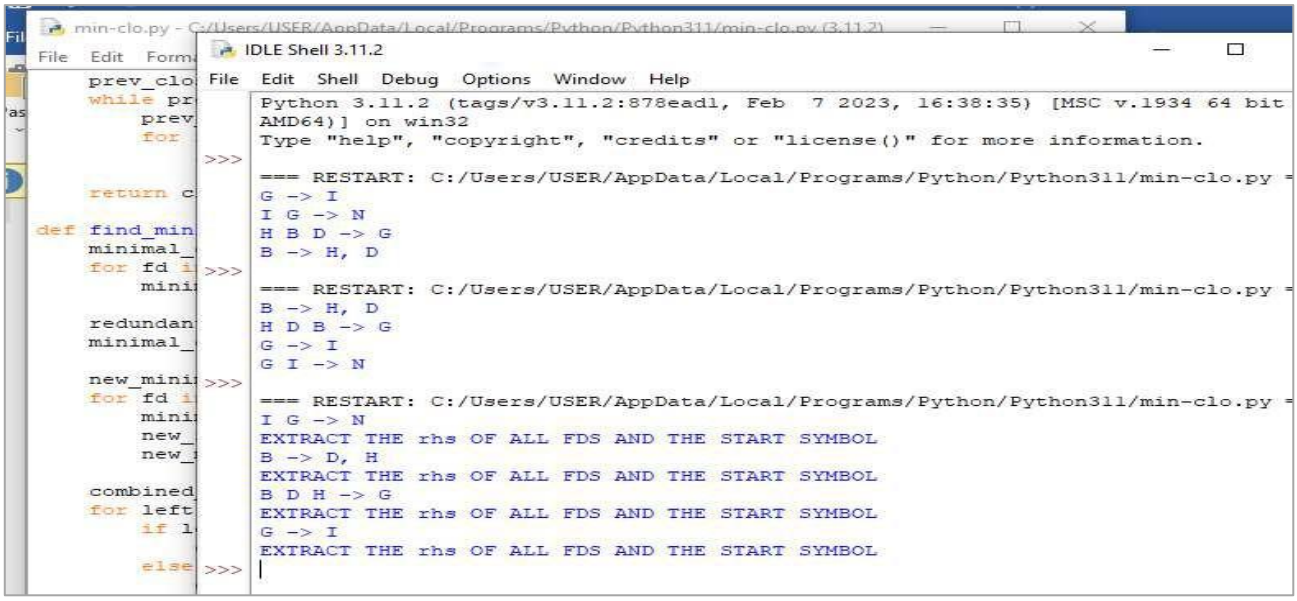

Figure 2: Minimal cover attributes

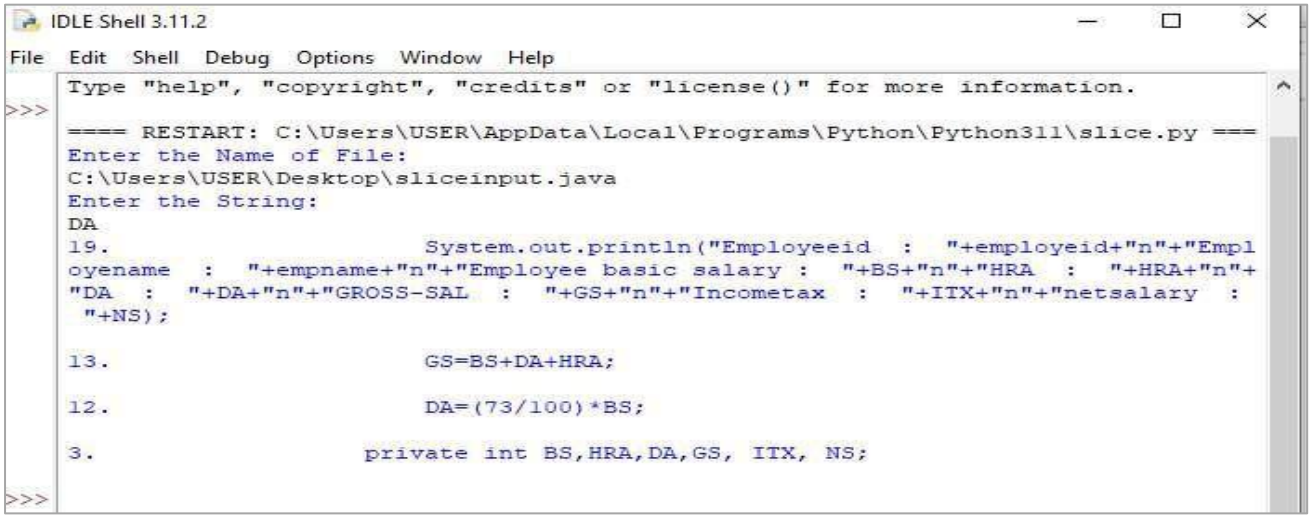

Figure 3: Backward Slice for The Attribute DA

The formal notation (Object-Z) is extracted for the above slice and the natural language specification is extracted from the user defined LLM models as shown in the Figure 5. These extracted slices are represented using the OBJ-Z formal notations, which gives more precise, mathematical and unambiguous representation of the program. These notations highlight only the required constructs like inputs, outputs, pre-conditions and post-conditions statements, from the extracted back slice of the program. In Object-Z representation, the input variables are represented using the '?' symbol and output variables are represented using '!' symbol and the derived variables are represented using the delta symbols, as in Figure 4. These notations are further used for extracting the software specifications and user requirements.

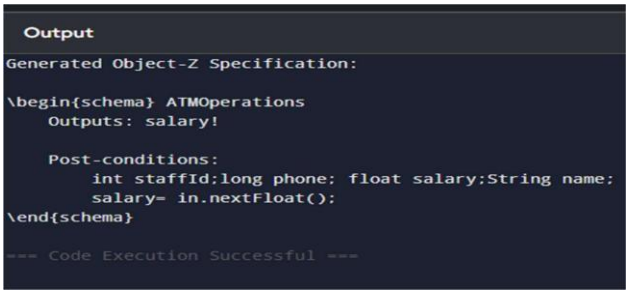

Figure 4: Object Z notation for the backward slice

***Corresponding Author:** Aparna K. S

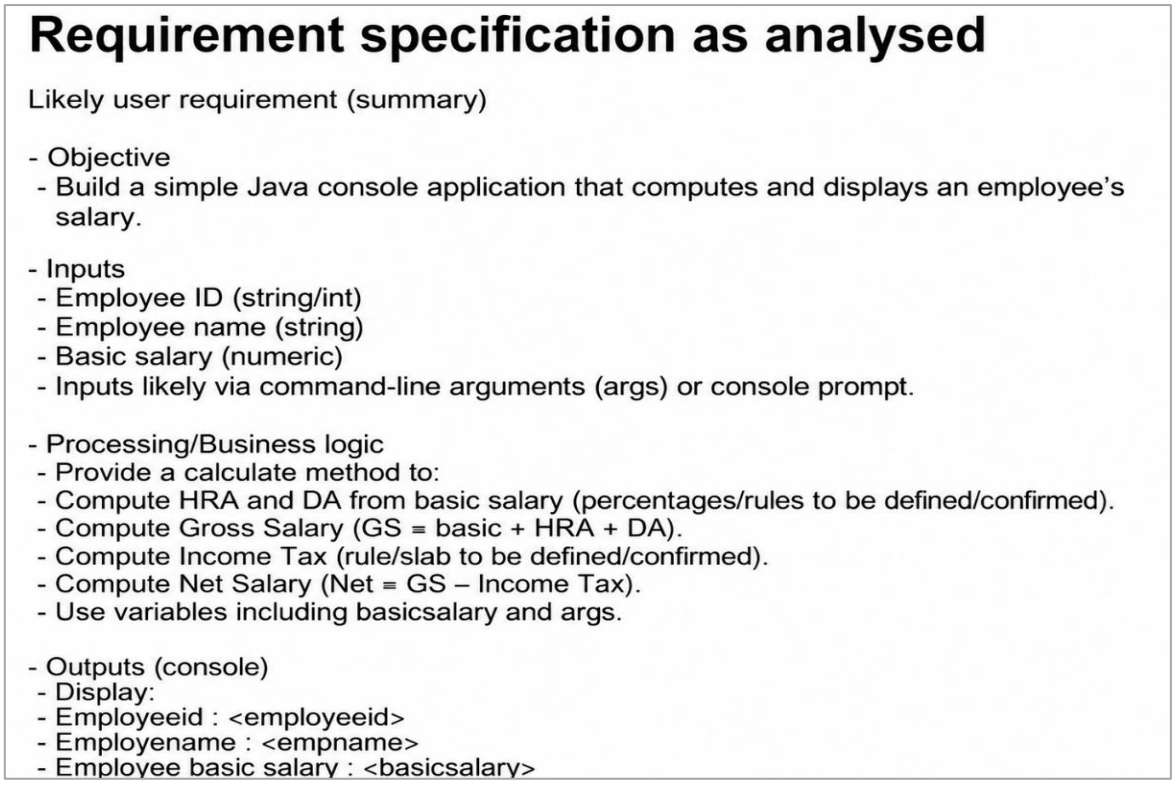

Figure 5: Software requirements from the Object-Z notation

CONCLUSION

This paper presents an automated tool for abstracting design information and software specifications from a given Java program. The tool analyses the program and represents its behaviour in the form of Control Flow Graph (CFG) and Data Flow Graph (DFG), which are then stored as Control Flow Tables (CFT) and Data Flow Tables (DFT). A comparative analysis of the proposed methodology is provided along with relevant results. Functional dependencies are extracted from the DFT and subsequently minimized to obtain an optimal set of attributes, which assist in understanding the core logic of the program. Based on these attributes, the various functional components of the input Java program are abstracted. Using the backward slicing techniques, the tool identifies the relevant slices of the program that contribute to specific functionalities. These precise slices are formally represented and further used to derive the software specifications and requirements, enabling easier program comprehension and re-engineering. The tool has been evaluated for Core Java programs with varying levels of complexity. Experimental results confirm that the tool consistently generates accurate and expected outputs, demonstrating its correctness and completeness. As a future scope, these specifications can be reengineered and migrated to advanced versions

REFERENCES

1. Stoica, B. A., Sahoo, S. K., Larus, J. R., & Adve, V. S. (2021). Statistical program slicing: A hybrid slicing technique for analyzing deployed software. *arXiv preprint arXiv:2201.00060*.
2. Postolski, I., Braberman, V., Garbervetsky, D., & Uchitel, S. (2022). Dynamic slicing by on-demand reexecution. *arXiv preprint arXiv:2211.04683*.
3. Molnar, A.-J., & Motogna, S. (2021). A study of maintainability in evolving open-source software.
4. *Communications in Computer and Information Science*, 261–282.
5. Kovács, Z., & Vujic, A. (2024). Open source prover in the attic. *arXiv preprint arXiv:2401.13702*.
6. Assunção, W. K. G., Marchezan, L., Egyed, A., & Ramler, R. (2024). Contemporary software modernization: Perspectives and challenges to deal with legacy systems. *arXiv preprint arXiv:2407.04017*.
7. Miskell, C., Diaz, R., Ganeriwala, P., Slhoub, K., & Nembhard, F. (2023). Automated framework to extract software requirements from source code. *Proceedings of the International Conference on Natural Language Processing and Information Retrieval*, 130–134.
8. Irani, Z., Abril, R. M., Weerakkody, V., Omar, A., & Sivarajah, U. (2023). The impact of legacy systems on digital transformation in European public administration: Lessons learned from a multi-case analysis. *Government Information Quarterly*, 40(1), 101784.
9. Bianchi, A., Caivano, D., Marengo, V., & Visaggio, G. (2003). Iterative reengineering of legacy systems. *IEEE Transactions on Software Engineering*, 29(3), 225–241.
10. Lano, K., Haughton, H., Yuan, Z., & Alfraihi,

***Corresponding Author:** Aparna K. S

H. (2024). Agile model-driven re-engineering.
11. *Innovations in Systems and Software Engineering*, 20(4), 559–584.
12. Mohottige, T. I., Polyvyanyy, A., Buyya, R., Fidge, C., & Barros, A. (2024). Microservices-based software systems reengineering: State-of-the-art and future directions. *arXiv preprint arXiv:2407.13915*.
13. Nandini, A., Singh, R., & Rathee, A. (2024). Code smells and refactoring: A tertiary systematic literature review. *International Journal of System of Systems Engineering*, 14(1), 83–143.
14. Cordeiro, J., Noei, S., & Zou, Y. (2024). An empirical study on the code refactoring capability of large language models. *arXiv preprint arXiv:2411.02320*.
15. Alaküla, A. R., Hedin, G., Fors, N., & Pop, A. (2024). Property probes: Live exploration of program analysis results. *Journal of Systems and Software*, 211, 111980.
16. Riouak, I., Fors, N., Hedin, G., & Reichenbach, C. (2024). IntraJ: An on-demand framework for intraprocedural Java code analysis. *International Journal on Software Tools for Technology Transfer*, 26(6), 687–705.
17. Hejderup, J., Beller, M., Triantafyllou, K., & Gousios, G. (2022). Präzi: From package-based to call-based dependency networks. *Empirical Software Engineering*, 27(5), 102.
18. Weiser, M. (1984). Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4), 352–357.
19. Tip, F. (1995). A survey of program slicing techniques. *Journal of Programming Languages*, 3, 121–189.
20. Gallagher, K. B., & Lyle, J. R. (1991). Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8), 751–761.
21. Zhang, Y. (2019). SymPas: Symbolic program slicing. *arXiv preprint arXiv:1903.05333*.
22. Seghir, M. N. (2018). Data-flow guided slicing. *arXiv preprint arXiv:1808.01232*.
23. Stoica, B. A., Sahoo, S. K., Larus, J. R., & Adve, V. S. (2021). Statistical program slicing: A hybrid slicing technique for analyzing deployed software. *arXiv preprint arXiv:2201.00060*.
24. Seghir, M. N. (2018). Data-flow guided slicing. *arXiv preprint arXiv:1808.01232*.
25. Vidziunas, L., Binkley, D., & Moonen, L. (2024). The impact of program reduction on automated program repair. *arXiv preprint arXiv:2408.01134*.
26. Yadavally, A., Li, Y., & Nguyen, T. N. (2024). Predictive program slicing via execution knowledge-guided dynamic dependence learning. *Proceedings of the ACM on Software Engineering*, 1(FSE), 271–292.
27. Soremekun, E., Kirschner, L., Böhme, M., & Zeller, A. (2021). Locating faults with program slicing: An empirical analysis. *Empirical Software Engineering*, 26(3).
28. Shahandashti, K. K., Mohajer, M. M., Belle, A. B., Wang, S., & Hemmati, H. (2024). Program slicing in the era of large language models. *arXiv preprint arXiv:2409.12369*.
29. Ferrarotti, F., Moser, M., & Pichler, J. (2020). Stepwise abstraction of high-level system specifications from source code. *Journal of Computer Languages*, 60, 100996.
30. Miskell, C., Diaz, R., Ganeriwala, P., Slhoub, K., & Nembhard, F. (2023). Automated framework to extract software requirements from source code. *Proceedings of the International Conference on Natural Language Processing and Information Retrieval*, 130–134.
31. Shivaram, A. M., & Handigund, S. M. (2015). An ameliorated methodology for the abstraction of object-oriented features from software requirements specification. *Procedia Computer Science*, 62, 274–281.
32. Weiser, M. (1984). Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4), 352–357.

***Corresponding Author:** Aparna K. S