

# An efficient computational framework for abstracting program slices from Java code

Aparna Kudloor Siddalingaiah<sup>1</sup>, Rajkumar Kulkarni<sup>2</sup>

<sup>1</sup>Department of Computer Science and Engineering, Global Academy of Technology affiliated to Visvesvaraya Technological University, Belagavi, India

<sup>2</sup>Department of Computer Science and Engineering, Ballari Institute of Technology and Management, Ballari, India

## Article Info

### Article history:

Received Apr 17, 2025

Revised Jul 23, 2026

Accepted Aug 12, 2026

### Keywords:

Artificial intelligence

Java code

Large language models

Program slicing

Software reengineering

Static analysis

## ABSTRACT

Over the past decades, Java-based applications have formed the backbone of enterprise, web, and mobile software systems, often containing complex business logic and domain-specific requirements embedded in older codebases (Java SE 1.2-8). Maintaining and migrating such systems is difficult due to unstructured code, missing documentation, and outdated design artifacts. To address these challenges, the paper introduces a computationally efficient static-backward slicing framework for Java programs. The framework restructures the source code, abstracts control and data flow information, computes functional dependencies using a data flow table (DFT), and extracts precise program slices. Extracted slices are further represented using object-Z notation to formalize software specifications. Furthermore, the extracted backward slices and object-Z specifications can be leveraged with artificial intelligence (AI) and large language models (LLMs) to automatically generate human-readable software requirements. This integration also ensures structured, accurate, and minimal requirements outputs, facilitating AI-assisted software modernization. Experimental evaluations on Java programs such as automated teller machine (ATM), student grading, and quadratic root calculators exhibit significant reductions in extracted code size while preserving functional dependencies. In comparison with traditional dynamic and machine learning (ML)-based slicing approaches, the proposed framework offers improved precision, computational efficiency, and scalability, making it suitable for modernizing legacy Java systems and supporting AI-driven software engineering tasks.

*This is an open access article under the [CC BY-SA](#) license.*



## Corresponding Author:

Aparna Kudloor Siddalingaiah

Department of Computer Science and Engineering

Global Academy of Technology affiliated to Visvesvaraya Technological University

Jnana Gangotri Campus, Hospet Rd, Near Allipura, Ballari 58310, India

Email: [aparna.vastrad@gmail.com](mailto:aparna.vastrad@gmail.com)

## 1. INTRODUCTION

Over the past few decades, to meet increasing demand and supply goals, several businesses have increasingly relied on automation to optimize operations and improve service efficiency [1]. This shift in organizational activities has further increased the demand for software solutions, particularly those developed using object-oriented programming languages such as Java and C++ [2], [3]. As organizations evolved, these software applications have undergone numerous need-based modifications to accommodate changing business requirements [4]. However, such incremental changes have rendered many software systems unstructured and difficult to maintain, especially when original design artifacts or developers are no longer available [5], [6].

Consequently, understanding the legacy code and migrating these applications to new modern paradigms while preserving their embedded business logic has become a critical challenge in software engineering [7].

Program slicing has emerged as an effective decomposition technique that narrows the focus to relevant portions of a program [8]. The fundamental objective of program slicing is to eliminate irrelevant statements from source code while preserving program semantics [9], [10]. Program slicing also plays a crucial role in analyzing functional dependencies within Java applications [11]. Unlike dynamic slicing, static slicing is computationally inexpensive and does not rely heavily on execution traces and extensive test cases [12]. Moreover, static slicing approaches not only simplify software maintenance, but also support security analysis, debugging, and performance optimization [13]. Its ability to provide a complete view of dependencies without program execution makes it highly suitable for extracting modern software design requirements [14].

Dynamic slicing techniques have traditionally been employed as backend mechanisms for program debugging, fault localization, and automated program repair [15], [16]. However, it has been widely observed that state-of-the-art dynamic slicing techniques face significant challenges in terms of efficiency and scalability [17]. These techniques tend to suffer as execution size and the number of executed instructions increase, leading to high runtime overhead [18], [19].

An on-demand re-execution-based dynamic slicing strategy was proposed in [2], where slices are progressively constructed by repeatedly executing the program to acquire dependency information. The approach compares on-demand re-execution with execute-once paradigms in terms of performance, highlighting trade-offs influenced by execution size and slice complexity. Comparative evaluations were conducted against tools such as JavaSlicer and Slicer4J [20], [21]. A symbolic abstraction-based dynamic slicing model for large-scale applications was introduced in [22], replacing traditional memory reference tracking with an abstract memory model operating on program symbols.

Despite these advancements, dynamic slicing remains unsuitable for production-level deployment due to the high cost of runtime instrumentation [1]. To mitigate this, a hybrid static-dynamic slicing approach was introduced, which demonstrates the recovery of a large proportion of relevant statements with limited runtime overhead. Symbolic slicing techniques based on symbolic execution have also been explored [14], but they are computationally expensive for complex Java programs with numerous execution paths, leading to scalability concerns.

Data-flow-guided strategies have been proposed to improve slicing accuracy [15], yet they struggle with complex control structures common in object-oriented Java programs, which limits their effectiveness for design specification extraction. Similar challenges related to computational overhead and scalability have been reported in large-scale Java applications with extensive object-oriented features [23], [24]. Some approaches primarily focus on automated program repair while largely overlooking the role of functional dependencies in precise slice extraction, which can restrict effective program abstraction [23].

Execution-guided and predictive slicing approaches relying on dynamic dependence learning have also been investigated in [25]. However, their heavy reliance on execution data restricts applicability in scenarios where execution traces are incomplete, unavailable, or costly to obtain. Machine learning (ML)-based slicing techniques have been proposed to derive formal design specifications from Java programs [19], [26], [27], but these approaches often introduce additional training overhead and depend on large labelled datasets. Nevertheless, these studies collectively highlight the importance of static slicing in identifying implicit and non-trivial dependencies in complex object-oriented programs.

Recent advancements in artificial intelligence (AI)-assisted program analysis and specification translation further demonstrate a growing convergence between static program slicing and machine intelligence [28], [29]. These developments reinforce the need for efficient, dependency-aware static slicing frameworks that support precise program abstraction and serve as a reliable foundation for downstream AI-driven software engineering tasks. Thereby, to address these limitations, there is a need to design and tailor a simplified and computationally efficient program slicing method that could accurately capture implicit control and functional dependencies, also minimize redundant computations towards extracting precise formal software specifications from Java programs, while retaining a balance between precision vs performance trade-offs.

In recent years, AI and large language models (LLMs) have increasingly supported static code analysis, program comprehension, and automated software documentation [29], [30]. The proposed static-backward slicing framework aims to complement such AI-driven tools by producing precise, minimal program slices and object-Z specifications, which can be automatically translated into human-readable requirements, enabling downstream reasoning for requirement abstraction, test generation, and software modernization. Therefore, an efficient static-backward slicing framework is needed that precisely captures both implicit control and functional dependencies while minimizing redundant computations and achieving an optimal balance between slicing accuracy and computational efficiency.

Motivated by these challenges, the proposed work thereby introduces a computational framework that systematically abstracts program slices using formal dependency modeling and structured program analysis. It combines computational processes of restructuring, design information extraction, and construction of a data flow table (DFT) to compute the functional dependencies with reduced and optimized cost. Compared to symbolic and graph-based slicing approaches, the proposed framework emphasizes a lightweight, DFT-driven static-backward slicing that prioritizes computational efficiency and precision for core Java constructs. While tools like Javasilcer and Slicer4J offer general-purpose slicing, they often incur higher overhead or fail to capture non-trivial dependencies in legacy Java programs.

The proposed static-backward slicing approach contributes towards enhancing program abstraction and formal specification extraction by addressing the limitations of traditional slicing methods. Unlike dynamic, hybrid, and ML-based slicing, which suffer from high computational costs and complexity, this method relies on static Java program analysis, making it cost-effective, scalable, and efficient. By integrating a DFT, it systematically organizes variable interactions, eliminates false dependencies, removes redundant code, and improves slicing accuracy. The combination of backward slicing for relevance and static slicing for refinement ensures high precision in extracting formal specifications. Additionally, its runtime-independent execution makes it ideal for large-scale Java programs, software re-engineering, and formal verification. Here, the slicing framework serves as a pre-processing step that generates precise and minimal code representations, whereas the proposed study also envisions an extension where AI / LLM components translate these formal representations into structured human-readable requirements, making the system useful for modern software engineering tasks. Therefore, the proposed slicing framework can further integrate with LLMs to translate formal object-Z specifications into structured natural-language requirements. Here, the alignment between static slicing and AI-based specification translation bridges formal program analysis with intelligent requirement generation.

## 2. METHOD

The work proposes a simplified and computationally efficient static-backward slicing approach that is analytically tailored with a series of effective computing steps to extract precise formal software specifications from Java programs. The proposed computational framework, realized as a tool, involves a set of core computing processes viz. restructuring the input program with a pre-processing stage, abstraction of control flow information, abstraction of DFT, abstraction of functional dependencies from DFT, obtaining minimal cover of the functional dependencies, and extraction of the backward slice from the minimal cover attributes. These steps are intended to optimize the performance of static-backward slicing for deriving precise formal software design specifications with reduced cost and effort. The following Figure 1 shows the block-based architecture of the proposed slicing approach.

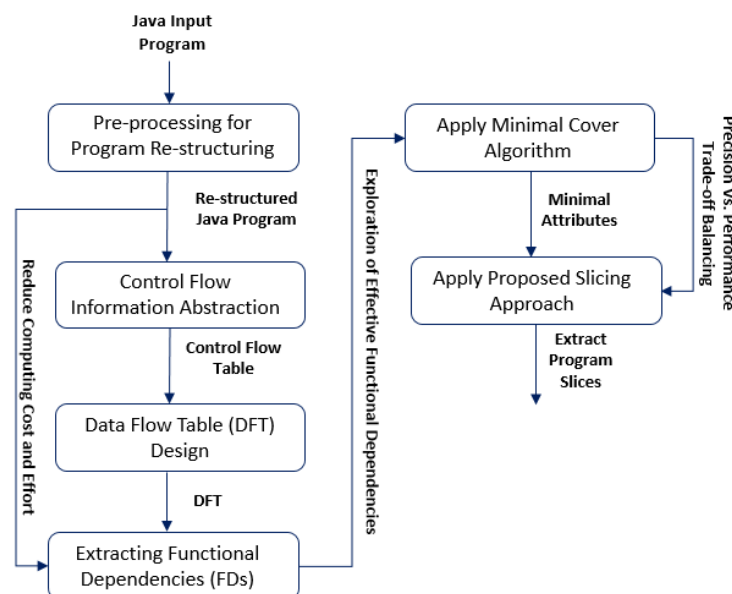


Figure 1. Core computing steps involved in proposed static-backward slicing

### 2.1. Pre-processing stage for Java program restructuring

The proposed computational framework integrates a pre-processing stage in Java program restructuring. The prime motive of this computing step is to ensure that the input program is in structured and well-defined format to positively influence the performance of program slicing, as without proper re-structuring the program slicing become error-prone, inefficient and computationally more expensive. This stage not only helps eliminate the irrelevant and unstructured code but also makes the Java program suitable to improve the control flow representation and data flow analysis. Let  $P_{java}$  be the input Java program which is represented as a sequence of lines using (1), and transformation process is also defined as a function using (2).

$$P_{java} = \{L_i\}_{i=1}^n \quad (1)$$

$$F: P_{java} \rightarrow P'_{java} \quad (2)$$

Where  $L_i$  represents the  $i$ -th line in the source code and  $P'_{java}$  refers to the restructured Java program. The program restructuring approach in the proposed work initially performs the appending of user-defined packages, classes, and interfaces to an input Java program  $P_{java} \cup S(P_U, C_U, I_U)$  and it is performed with the function of  $A(P_{java})$ . The mathematical expression using (3) to (5) represents this flow. Let  $\varepsilon(P'_{java})$  be the set of all comment lines in  $P'_{java}$ . Further, the computing process  $P''_{java}$  eliminates the comment lines while only considering the executable statements and ignoring the comment lines in  $P'_{java}$ . Here,  $\varepsilon(P'_{java}) = \{L_i \in P'_{java} \mid \forall L_i \in sL_C, mL_C\}$ . The computing is further performed using (6).

$$P'_{java} = A(P_{java}) \quad (3)$$

$$P'_{java} = P_{java} \cup S(P_U, C_U, I_U) \quad (4)$$

$$P'_{java} = P_{java} \oplus S(P_U, C_U, I_U) \quad (5)$$

$$P''_{java} = P'_{java} \setminus \varepsilon(P'_{java}) \quad (6)$$

Where  $P''_{java}$  refers to the transformed Java program which is obtained while eliminating the comment lines. In continuation, the computing process further also enables elimination of the blank lines in the pre-processing stage from the transformed code of  $P''_{java}$  considering as in (7).

$$P''_{java} = P''_{java} \setminus \{\{L_i\}_{i=1}^n \in P''_{java} \mid L_i = ""\} \quad (7)$$

Where  $P''_{java}$  refers to a set or ordered list of all lines from the Java program after previous pre-processing steps.  $L_i$  represents a line of code from  $P''_{java}$  indexed by  $i$  from 1 to  $n$ . The condition  $L_i = ""$  identifies lines that are empty or blank lines, whereas  $\setminus$  represents a set difference operator. The condition  $L_i = ""$  also implies that remove all lines  $L_i$  that are empty strings.

It also transforms multiple declaration statements into a single declaration statement using a function of transformation  $f_{trans}(P''_{java})$ . If  $s = \{s_i\}_{i=1}^n$  represents set of individual declarations in  $P''_{java}$ . Then the function  $f_{trans}(s \in P''_{java})$  process  $s = \{s_i\}_{i=1}^n$  and merge multiple declarations into a single statement, considering as in (8).

$$s_{con} = \text{Type} \oplus \langle \{v_i\}_{i=1}^\xi \rangle \quad (8)$$

Here  $s_{con}$  refers to the consolidated statement, i.e. the computed final single-line declaration. Here the Type refers to the data types that could be either int, float and char. It considers the data type shared by all variables being declared. The concatenation operation symbolically joins items together into one syntactic unit. Also, the term  $\{v_i\}_{i=1}^\xi$  is a tuple of variable identifiers where  $\xi$  represents the total number of variables being declared. For multiple data types, the transformation is applied independently for each type, considering the (9).

$$f_{trans}(s) = \bigcup_{\text{Type}_i} \left( \text{Type}_i \oplus \sum_{j=1}^{\xi_i} v_{i,j} \right) \quad (9)$$

Here  $\bigcup(*)$  ensures each type's variables are clustered separately and  $\xi_i$  is the number of variables of  $\text{Type}_i$ .

The approach further also transforms the multi-line statements into a single statement line and allot line numbers to executable statements. If  $e(P''_{\text{java}})$  is a set of executable statements in a program  $P''_{\text{java}}$  such as  $e(P''_{\text{java}}) = \{L_i \in P''_{\text{java}} | L_i \rightarrow \text{exec}_{\text{statement}}\}$ . Then it assigns a unique line number to the executable statement using (10).

$$L'_i = (l_{no.}(i) \rightarrow L_i), \text{ where } \forall L_i \in e(P''_{\text{java}}) \quad (10)$$

Where  $L_i$  refers to the executable statements from the pre-processed Java program. Also  $l_{no.}(i)$  ensures that a line number is assigned to the  $i$ -th executable line.  $\rightarrow$  represents a mapping function that pairs a line number with its corresponding line. Finally,  $L'_i$  forms the new representation of the line  $i$  now tagged with the line number.

Further, the computing process identifies and removes the unused variables and makes the transformed Java program  $P''_{\text{java}}$  more suitable for further analysis. The program restructuring is modeled here with an optimized flow of execution, which is an essential step to ensure accurate, efficient, and scalable program slicing. It has to be noted that the related user-defined packages are processed and incorporated into the current program if the Java program contains external linked classes imported from other packages.

## 2.2. Abstraction of control flow information from re-structured Java program

The proposed work also considers abstraction of control flow information from the re-structured Java program. After analyzing the baseline studies, it has been observed that control flow abstraction plays a crucial role in program slicing. Thereby, in the proposed work the control flow information logic is further derived in the form of 4-col structured tabular form which is defined using (11). Here  $s_{start}$  column represents the initiation of the Java program, whereas the second column  $s_{end}$  represents the occurrence of the first encountered control or conditional statement. Third column  $Tr_1$  represents the statement where the control jumps if the control statement is true for particular inputs and the fourth column  $Tr_2$  represents the statement where the false statement gets executed.

$$CF_I(P''_{\text{java}}) = \{ \langle s_{start}, s_{end}, Tr_1, Tr_2 \rangle | s_{end} \in s_i, Tr_1, Tr_2 \in P''_{\text{java}} \} \quad (11)$$

This tabular representation in the proposed work captures the control flow transitions in the form of  $CF_I$  within  $P''_{\text{java}}$  and further helps formulating DFT in the control flow order.

## 2.3. Abstraction of data flow table in control flow order

The study realizes that the data variables in the program is based on the control flow order and further, the proposed work considers  $CF_I(P''_{\text{java}})$  of  $P''_{\text{java}}$  and aims to formulate the DFT in the order of control flow. The proposed work considers DFT as a structured tabular representation in the form of  $T_{DFT}$  in which each row corresponds to a program statement and also consists of three attributes using (12).

$$T_{DFT}(i) = \{ \langle L'_i, v_d^i, v_r^i \rangle | i \in [1, n] \} \quad (12)$$

Where  $L'_i$  refers to the  $i$ -th program statement with line number for  $n$  number of program statements. Also, here both  $v_d^i, v_r^i$  columns represent set of variables defined at statement number of lines  $i$  and set of variables referred at statement number of lines  $i$ . In the proposed slicing approach, the inclusion of DFT offers efficient tracking of variable dependencies, which also preserves the control flow order. It significantly optimizes and enhances the performance of further tasks such as abstraction of functional dependencies and also in the longer run influences the slicing approach as well.

## 2.4. Abstraction of functional dependencies from data flow table

The proposed work also assumes that functional dependency exists when a variable defined at a certain statement depends on variables referred to in the previous computing steps or statements. The functionality can be abstracted by finding the functional dependencies existing between the defined and referred variables. In the proposed work, the functional dependency in a program  $P''_{\text{java}}$  is mathematically represented using (13) as a relation of  $f$  which is extracted from the  $T_{DFT}(i)$ .

$$f(L, v) = \{ \langle L'_i, L'_j, v_c \rangle | v_c \in v_d^i \cap v_r^j, i > j \} \quad (13)$$

Here  $L'_i$  refers to the defining program statement where the  $v_d^i$  is the set of variables defined.  $L'_j$  refers to the dependent program statement line where the  $v_r^j$  was previously referred. Also  $v_c$  refers to the dependent common variables establishing the functional dependency. The condition of  $i > j$  implies that the

dependency follows the Java program execution order with increasing program statements. Here, in the proposed work this abstraction helps identify the functional dependencies across the program, and it is crucial for slicing and optimization. In continuation, the next stage further extracts the minimal cover attributes of the functional dependencies.

## 2.5. Finding minimal cover of the functional dependencies

In this computing step, the proposed work aims to minimize  $f(L, v)$  such as  $f \rightarrow f_{min}$  while eliminating the redundant dependencies. This approach ensures that only a minimal set of functional attributes should be extracted from  $f(L, v)$  which are essential for program slicing. The study assumes that if  $\langle L'_i, L'_j, v_c \rangle$  is considered a redundant functional dependency, that means that  $v_c$  appears in multiple functional dependencies with different program statement numbers. However, for program slicing, only a few significant dependencies are required to extract minimal cover attributes. This approach also helps reduce the computational complexity and cost in the proposed static-backward slicing approach. It also improves the clarity and maintainability of larger Java programs.

## 2.6. Extracting the backward slice from minimal cover attributes

The proposed work customizes an improved static-backward slicing approach based on Weiser's algorithm. Similarly, the study considers program slicing as computing a subset of program statements that contributes towards the value of a variable at specific program points. The proposed slicing, called backward slicing, searches and extracts only those program statements which are contributing to the calculation of the slicing variable, and it can be represented at (14).

$$S = \{L_n\} \cup \{L_i | \langle L'_i, L'_j, v_{slice} \rangle \in f_{min}, L'_j \in S\} \quad (14)$$

The above process computes the last occurrence of  $L_n$  of the slicing variable  $v_{slice}$  and further computes the statements that define  $v_{slice}$  in the previous dependencies. In (14)  $S$  represents the set of all lines included in the program slice.  $L_n$  refers to the slicing criterion line and  $f_{min}$  helps deriving the minimal set of dependencies relevant to  $v_{slice}$ . Further, the algorithm also enables optimized traversal considering the mathematical (15).

$$S = S \cup \{L_i | \exists \langle L'_i, L'_j, v_{slice} \rangle \in f_{min}, L'_j \in S\} \quad (15)$$

Here  $S$  represents the current slice i.e. the set of lines influencing the value of the target attribute  $v_{slice}$ , the minimal set of relevant data dependencies also explored via  $f_{min}$ . Here  $\exists$  implies existence, and further the process keeps expanding the slice  $S$  by including any  $L_i$  that defines the slicing variable  $v_{slice}$ . It continues adding the contributing statements iteratively. Finally, these line numbers are extracted from the DFT and the corresponding statements from the restructured input program are sliced, ignoring the irrelevant statements. Further, these slices are used to abstract the software formal specification of the given program. The same procedure is followed for all minimal cover attributes, obtaining multiple slices for any given program. Here the output  $S(v) \in P_{slice}$  represents the set of backward slices for the slicing variable  $v_{slice}$ . Further, the extracted backward program slices  $S(v) \in P_{slice}$  are represented using object-Z notation, ensuring mathematical precision and structured representation. It has to be noted that the proposed slicing framework operates with linear complexity relative to the number of executable statements in the analyzed Java programs. While no pseudocode is included, the complete slicing workflow is systematically represented through formal derivations and corresponding schematic figures ensuring methodological clarity and reproducibility.

## 2.7. Potential artificial intelligence / large language model integration

While the current study focuses on static-backward slicing and object-Z specification extraction, the resulting formal representations can serve as input to AI-driven tools for modern software engineering tasks. In particular, extracted object-Z specifications can be normalized into simpler tokens and processed by LLMs to generate structured, human-readable requirements (e.g. actors, actions, pre-conditions, post-conditions, and invariants) in JSON format, enabling automated documentation, requirement extraction, and software modernization. Post-processing with rule-based checks can also ensure correctness, and optional fine-tuning on paired object-Z and requirement datasets can further improve accuracy. This integration remains a prospective extension of the framework and does not affect the current experimental evaluation. LLMs would automatically translate formal program slices into human-readable requirements. This prospective integration remains future work and is not experimentally validated here. Algorithm 1 summarizes the key steps of the proposed static-backward slicing framework. It also provides a compact procedural view of the proposed static-backward slicing framework.

**Algorithm 1.** DFT-driven static-backward slicingInput: Java source code, slicing criterion  $\langle v_{slice}, L_n \rangle$ 

Output: Minimal backward slice, object-Z specification

Begin

1. Pre-process and restructure core Java program  $P_{java}$  to obtain structured program  $P''_{java}$
2. Abstract control flow information  $CF_I(P''_{java})$
3. Construct data flow table  $T_{DFT}(i)$  in control-flow order
4. Compute functional dependencies  $f(L, v)$  from  $T_{DFT}$
5. Compute the minimal cover  $f_{min}$  by eliminate false and redundant dependencies
6. Apply backward slicing w.r.t  $\langle v_{slice}, L_n \rangle$
7. Generate minimal slice  $S(v)$  and generate the corresponding object-Z specification

End

**3. RESULTS AND DISCUSSION**

The experimental evaluation of the formulated tool is performed for various Java programs such as automated teller machine (ATM), student grading, and Karnataka common entrance test (KCET), considering the integrated development environment (IDE) of PyCharm. Here the proposed algorithms are scripted in a single file implementation considering Python. The evaluation framework considers internal memory of 12 GB and a 64-bit Windows operation system supported with the x64-based processor. For experimental evaluation the study considers a Java program for finding quadratic roots which is shown in Figure 2. The framework was evaluated on representative Java programs ranging from small utilities to medium-sized modules. Each slicing task was executed once under identical static analysis settings, focusing on control- and data-dependency preservation rather than runtime profiling.

```

1  import java.util.Scanner;
2
3  public class Quadratic
4  {
5      public static void main(String[] args)
6      {
7          double a, b, c, det, root1, root2;
8          Scanner in = new Scanner(System.in);
9          System.out.println("Enter the value of a: ");
10         a = in.nextDouble();
11         System.out.println("Enter the value of b: ");
12         b = in.nextDouble();
13         System.out.println("Enter the value of c: ");
14         c = in.nextDouble();
15         if (a == 0 || b == 0 || c == 0)
16         {
17             System.out.println("Invalid input: Enter non-zero
coefficients.");
18             System.exit(0);
19         } else
20         {
21             det = b * b - 4 * a * c;
22             if (det > 0)
23             {
24                 root1 = (-b + Math.sqrt(det)) / (2 * a);
25                 root2 = (-b - Math.sqrt(det)) / (2 * a);
26                 System.out.println("Roots are real and distinct.");
27                 System.out.println("Root1 = " + root1);
28                 System.out.println("Root2 = " + root2);
29             } else if (det == 0)
30             {
31                 root1 = root2 = -b / (2 * a);
32                 System.out.println("Roots are real and equal.");
33                 System.out.println("Root1 = Root2 = " + root1);
34             } else
35             {
36                 double realPart = -b / (2 * a), imaginaryPart =
Math.sqrt(-det) / (2 * a);
37                 System.out.println("Roots are complex and imaginary.");
38                 System.out.println("Root1 = " + realPart + " + i" +
imaginaryPart);
39                 System.out.println("Root2 = " + realPart + " - i" +
imaginaryPart);
40             }
41         }
42         in.close();
43     }
44 }

```

Figure 2. Input quadratic Java program after restructuring

The experimental results for the Java program for finding quadratic roots is illustrated in Figure 3. Figure 3(a) clearly shows the control flow table (CFT) as obtained from evaluating the mentioned (11) for the Java program for quadratic roots. It also shows the DFT in Figure 3(b) for the same Java program.

| Start | End | Trans1 | Trans2 |
|-------|-----|--------|--------|
| 1     | 14  | 15     | 19     |
| 15    | 17  | 42     |        |
| 19    | 21  | 22     | 29     |
| 22    | 28  | 41     |        |
| 29    | 29  | 30     | 35     |
| 30    | 34  | 41     |        |
| 35    | 40  | 41     |        |
| 41    | 44E |        |        |

(a)

| Sl.no | DEF-VAR                    | REF-VAR                 |
|-------|----------------------------|-------------------------|
| 2     | class                      |                         |
| 6     | a, b, c, det, root1, root2 |                         |
| 9     | a                          |                         |
| 11    | b                          |                         |
| 13    | c                          |                         |
| 14    |                            | a, b, c                 |
| 20    | det                        | a, b, c                 |
| 21    |                            | det                     |
| 23    | root1                      | b, a, det               |
| 24    | root2                      | b, a, det               |
| 26    | root1                      |                         |
| 27    | root2                      |                         |
| 29    |                            | det                     |
| 31    | root1, root2               | a, b                    |
| 33    |                            | root1                   |
| 36    | realPart, imaginaryPart    | a, b, det               |
| 38    |                            | realPart, imaginaryPart |
| 39    |                            | realPart, imaginaryPart |

(b)

Figure 3. Quadratic Java program contains (a) CFT and (b) DFT

The software specification depends on the functional dependencies existing between the referred and defined variables in object-Z form. The natural language specifications are designed using the object-Z representation. These specifications are further used to abstract the user requirements. The extensive evaluation of the proposed tool further also considers the different size of Java programs in the proposed tool of slicing approach which considers data-flow table coverage (DFT-COV) approach such as ATM, calculator, KCET, library, emp\_salary, and stud\_grade. The original ATM program comprises 240 lines of code, while the grade program consists of 321 lines and the remaining programs vary approximately between 200 to 500 lines. It has been observed that in existing slicing traditional coverage approaches (TRAD-COV), the extracted slices consist of significantly increased lines of codes for the same Java programs whereas in the proposed slicing techniques the lines of code in the extracted slices have significantly reduced with least number of statements which are extracted from the various forms of Java programs as shown in the Figure 4.

Figure 4 illustrates how DFT explicitly separates functional dependencies from control dependencies, enabling precise backward traversal without graph expansion overhead. Let TRAD-COV denote traditional coverage that counts all syntactic dependencies. In contrast, DFT-COV measures only semantically valid functional dependencies extracted via the DFT. Empirically, DFT-COV reduces false dependencies by eliminating control-insensitive and redundant variable interactions. The experimental analysis of the proposed tool shows that, unlike existing slicing approaches [14], [26], [27], the proposed tool offers precise slice extraction with reduced cost of computation and significantly balance the precision vs performance trade-off with highly optimized flow of execution. It is also observed that the proposed tool is effective in identifying non-trivial dependencies in complex Java programs in which slicing approaches are still struggling.

Overall, the proposed work contributes towards cost effectiveness, computational efficiency and scalability factors with improvised backward-slicing approach which is designed on static methods. It is also observed that when compared with both ML-based and dynamic slicing strategies, the proposed tool offers significantly better outcome in deriving formal software specifications with reduced lines of code and yet offers computational efficiency with simplified strategies. In addition to the lines of code reduction, the extracted slices were analyzed for precision and redundancy. The proposed approach achieves comparatively high slice correctness with minimal redundant statements, demonstrating efficiency compare to baseline traditional slicing approaches. Across representative Java modules, the proposed slicing consistently reduces irrelevant lines of code by approximately 30–45%, primarily due to false dependency elimination via DFT-COV. The object-Z schema captures the functional dependency between program variables derived from the minimal backward slice as shown in the Figure 5.

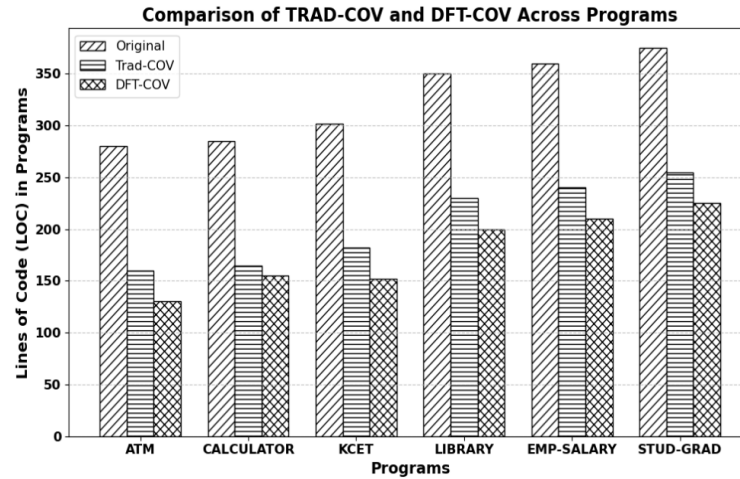


Figure 4. Comparison of TRAD-COV and DFT-COV slicing approaches

```
Generated Object-Z Specification:

\begin{schema} ATMOperations
  Outputs: root1!

  Post-conditions:
    double a,b,c,det,root1,root2;
    root1 = (-b + Math.sqrt(det)) / (2 * a);
    root1 = root2 = -b / (2 * a);
\end{schema}
```

Figure 5. Generated object-Z specification

Although current experiments focus more on static slicing however, our extended work considers resulting object-Z specifications to be fed to AI driven LLM tools for requirement extraction, highlighting broader applicability. The precision and minimalism of the generated slices suggest strong potential for downstream integration with LLMs that rely on structured, dependency-aware code representations. Unlike graph-based and symbolic slicing, the proposed DFT-based approach resolves dependencies through structured table lookups while avoid graph traversal and symbolic execution, and thereby achieves near-linear computational behavior under static analysis assumptions as shown in the Table 1. The backward slices and object-Z specifications derived through static slicing can later be leveraged by AI/LLM framework to produce structured, human readable requirements which support downstream tasks such as automated documentation and requirement analysis. This paper however reports result only for the static slicing framework.

Table 1. Qualitative comparison of complexity

| Reference                | Dependency model | Time complexity worst case | Overhead |
|--------------------------|------------------|----------------------------|----------|
| Ahmed <i>et al.</i> [20] | Graph-based      | $O(N^2)$                   | High     |
| He <i>et al.</i> [27]    | Symbolic         | $O(N^2)$ - $O(N^3)$        | High     |
| Proposed                 | DFT-based        | $O(N)$                     | Low      |

#### 4. CONCLUSION

This work presented a computationally efficient static backward slicing framework for Core Java programs based on structured control flow and DFTs. By replacing graph-based dependency analysis with table-driven lookup operations, the proposed approach improves scalability while preserving slicing correctness and specification integrity. Experimental evaluation using case studies and program simulations demonstrates effective reduction in lines of code and a balanced trade-off between precision and computational efficiency. The framework also enables seamless generation of object-Z formal specifications from extracted program slices. Future work will focus on formal verification of slice correctness using standard benchmarks,

automated translation of object-Z specifications into human-readable requirements using LLMs, and extension of the framework to support multi-language codebases. The AI / LLM integration is considered a future research direction and is not evaluated in this study.

FUNDING INFORMATION

Authors state no funding involved.

AUTHOR CONTRIBUTIONS STATEMENT

This journal uses the Contributor Roles Taxonomy (CRediT) to recognize individual author contributions, reduce authorship disputes, and facilitate collaboration.

| Name of Author    | C | M | So | Va | Fo | I | R | D | O | E | Vi | Su | P | Fu |
|-------------------|---|---|----|----|----|---|---|---|---|---|----|----|---|----|
| Aparna Kudloor    | ✓ | ✓ | ✓  | ✓  | ✓  | ✓ |   | ✓ | ✓ | ✓ |    |    |   | ✓  |
| Siddalingaiah     |   |   |    |    |    |   |   |   |   |   |    |    |   |    |
| Rajkumar Kulkarni |   | ✓ |    |    |    | ✓ |   | ✓ | ✓ | ✓ | ✓  | ✓  |   |    |

|                       |                                |                            |
|-----------------------|--------------------------------|----------------------------|
| C : Conceptualization | I : Investigation              | Vi : Visualization         |
| M : Methodology       | R : Resources                  | Su : Supervision           |
| So : Software         | D : Data Curation              | P : Project administration |
| Va : Validation       | O : Writing - Original Draft   | Fu : Funding acquisition   |
| Fo : Formal analysis  | E : Writing - Review & Editing |                            |

CONFLICT OF INTEREST STATEMENT

Authors state no conflict of interest.

DATA AVAILABILITY

Data availability does not apply to this paper as no new data were created or analyzed in this study.

REFERENCES

[1] Q. Stievenart, D. Binkley, and C. D. Roover, "QSES: Quasi-static executable slices," in *IEEE 21st International Working Conference on Source Code Analysis and Manipulation, SCAM 2021*, 2021, pp. 209–213, doi: 10.1109/SCAM52516.2021.00033.

[2] J. Li *et al.*, "Enhancing real-time operating system security analysis via slice-based fuzzing," *IEEE Transactions on Software Engineering*, vol. 51, no. 12, pp. 3467–3485, Dec. 2025, doi: 10.1109/TSE.2025.3615642.

[3] A. J. Molnar and S. Motogna, "A study of maintainability in evolving open-source software," in *Evaluation of Novel Approaches to Software Engineering*, Cham, Switzerland: Springer, 2021, pp. 261–282, doi: 10.1007/978-3-030-70006-5\_11.

[4] M. Himansh and V. M. Manikandan, "A statistical study and analysis to identify the importance of open-source software," in *2022 International Conference on Innovative Trends in Information Technology (ICITIIT)*, 2022, pp. 1–6, doi: 10.1109/ICITIIT54346.2022.9744176.

[5] N. Jomhari, N. A. Ahmad Alias, A. A. A. Ellah, A. A. Magableh, and E. M. Ghazali, "A multi-criteria decision-making for legacy system modernization with FUCOM-WSM approach," *IEEE Access*, vol. 12, pp. 48608–48619, 2024, doi: 10.1109/ACCESS.2024.3383917.

[6] C. Miskell, R. Diaz, P. Ganeriwala, K. Slhoub, and F. Nembhard, "Automated framework to extract software requirements from source code," in *2023 7th International Conference on Natural Language Processing and Information Retrieval*, 2023, pp. 130–134, doi: 10.1145/3639233.3639242.

[7] Z. Irani, R. M. Abril, V. Weerakkody, A. Omar, and U. Sivarajah, "The impact of legacy systems on digital transformation in European public administration: lesson learned from a multi case analysis," *Government Information Quarterly*, vol. 40, no. 1, 2023, doi: 10.1016/j.giq.2022.101784.

[8] A. Bianchi, D. Caivano, V. Marengo, and G. Visaggio, "Iterative reengineering of legacy systems," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 225–241, 2003, doi: 10.1109/TSE.2003.1183932.

[9] K. Lano, H. Haughton, Z. Yuan, and H. Alfraihi, "Agile model-driven re-engineering," *Innovations in Systems and Software Engineering*, vol. 20, no. 4, pp. 559–584, 2024, doi: 10.1007/s11334-024-00568-z.

[10] T. I. Mohottige, A. Polyvyanyy, C. Fidge, R. Buyya, and A. Barros, "Reengineering software systems into microservices: state-of-the-art and future directions," *Information and Software Technology*, vol. 183, 2025, doi: 10.1016/j.infsof.2025.107732.

[11] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984, doi: 10.1109/TSE.1984.5010248.

[12] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995.

[13] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 751–761, 1991, doi: 10.1109/32.83912.

[14] K. K. Shahandashti, M. M. Mohajer, A. B. Belle, S. Wang, and H. H. Lassonde, "Program slicing in the era of large language models," in *2025 IEEE 49th Annual Computers, Software, and Applications Conference, COMPSAC 2025*, 2025, pp. 1224–1233, doi: 10.1109/COMPSAC65507.2025.00155.

- [15] X. Rao, C. Sun, D. Zeng, Y. Huang, and G. Tan, "Sliver: a scalable slicing-based verification for information flow security," *IEEE Transactions on Dependable and Secure Computing*, vol. 22, no. 1, pp. 457–473, 2025, doi: 10.1109/TDSC.2024.3403653.
- [16] Q. Zhang *et al.*, "A systematic literature review on large language models for automated program repair," *ACM Transactions on Software Engineering and Methodology*, 2026, doi: 10.1145/3799693.
- [17] J. Singh, P. M. Khilar, and D. P. Mohapatra, "Dynamic slicing of distributed Aspect-Oriented Programs: a context-sensitive approach," *Computer Standards and Interfaces*, vol. 52, pp. 71–84, 2017, doi: 10.1016/j.csi.2017.01.007.
- [18] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, 1990, pp. 246–256, doi: 10.1145/93548.93576.
- [19] C. Hammacher, "Design and implementation of an efficient dynamic slicer for Java," Bachelor's Thesis, Saarland University, Saarbrücken, Germany, 2008.
- [20] K. Ahmed, M. Lis, and J. Rubin, "Slicer4J: a dynamic slicer for Java," in *ESEC/FSE 2021 - 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1570–1574, doi: 10.1145/3468264.3473123.
- [21] D. Binkley and L. Moonen, "Assessing the impact of execution environment on observation-based slicing," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2022, pp. 40–44, doi: 10.1109/SCAM55253.2022.00010.
- [22] L. Vidziunas, D. Binkley, and L. Moonen, "The impact of program reduction on automated program repair," in *2024 IEEE International Conference on Software Maintenance and Evolution, ICSME 2024*, 2024, pp. 337–349, doi: 10.1109/ICSME58944.2024.00039.
- [23] Q. Stiévenart, D. Binkley, and C. D. Roover, "An empirical evaluation of static, dynamic, and hybrid slicing of WebAssembly binaries," *Journal of Systems and Software*, vol. 228, 2025, doi: 10.1016/j.jss.2025.112453.
- [24] A. Yadavally, Y. Li, and T. N. Nguyen, "Predictive program slicing via execution knowledge-guided dynamic dependence learning," *Proceedings of the ACM on Software Engineering*, vol. 1, pp. 271–292, 2024, doi: 10.1145/3643739.
- [25] E. Soremekun, L. Kirschner, M. Böhme, and A. Zeller, "Locating faults with program slicing: an empirical analysis," *Empirical Software Engineering*, vol. 26, no. 3, 2021, doi: 10.1007/s10664-020-09931-7.
- [26] A. Yadavally, Y. Li, S. Wang, and T. N. Nguyen, "A learning-based approach to static program slicing," *Proceedings of the ACM on Programming Languages*, vol. 8, pp. 83–109, 2024, doi: 10.1145/3649814.
- [27] D. He, J. Lu, Y. Gao, and J. Xue, "Selecting context-sensitivity modularly for accelerating object-sensitive pointer analysis," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 719–742, 1 Feb. 2023, doi: 10.1109/TSE.2022.3162236.
- [28] M. Abdallah, A. Ngah, A. Al-Rahamneh, and D. Staegmann, "AI-based program slicing: Techniques, tools, and comparative analysis," in *2025 12th International Conference on Information Technology (ICIT)*, 2025, pp. 334–338, doi: 10.1109/ICIT.2025.00068.
- [29] P. He, S. Wang, and T.-H. Chen, "SLICET5: static program slicing using language models with copy mechanism and constrained decoding," 2025, *arXiv: 2509.17338*.
- [30] F. Olmedo, "Static slicing for probabilistic programs: an overview," in *Principles of Verification: Cycling the Probabilistic Landscape*, Cham, Switzerland: Springer, 2025, pp. 155–178, doi: 10.1007/978-3-031-75783-9\_7.

## BIOGRAPHIES OF AUTHORS



**Aparna Kudloor Siddalingaiah**     working as an assistant professor in the Department of Computer Science and Engineering at Global Academy of Technology, an Autonomous Institution, affiliated to Visvesvaraya Technological University (VTU), Belagavi with teaching experience of 20 years. She completed her B.E. in Computer science from Rao Bahadur Y. Mahabaleswarappa Engineering College, Gulbarga University and her M.Tech. in Computer Network Engineering from Ballari Institute of Technology and Management, Ballari in 2006 with a VTU rank. Currently, pursuing a Ph.D. degree in Software engineering from VTU. Passionate about the teaching-learning process involved in many innovative projects and has patents published. Having participated in many international and national conferences. Participated actively in several workshops and seminars with many paper publications in different domains and was involved in various academic committees like NBA and NAAC. She can be contacted at email: aparnaks@gat.ac.in.



**Dr. Rajkumar Kulkarni**     holds a Ph.D. in Software Engineering from Visvesvaraya Technological University, Belagavi, Karnataka, India. Presently heading the Department of Computer Science and Engineering, Ballari Institute of Technology and Management, Ballari, Karnataka, India. He has more than 35 years of experience in teaching with passionate teaching and getting involved in many aspects of academic growth like 30 international conferences, 18 national conferences, 45 international journal papers published, 8 patents published, and 2 patents received grants. His research domains include software engineering, DBMS, soft computing, and object technology. He is an active person in the process of NBA and NAAC. He has chaired many conferences and is involved actively in various committees and present dean, School of Computer Science at Ballari Institute of Technology and Management, Ballari. He can be contacted at email: rkulkarni@bitm.edu.in.