JOURNAL OF BASIC SCIENCE AND ENGINEERING

A Software Re-Engineering Framework for Effective Requirements Abstraction in Java-Based Systems

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

[1]Research scholar, Department of CSE, Ballari Institute of Technology and Management(Affiliated to Visvesvaraya Technological University), Belagavi, India
Assistant Professor, Department of Computer Science & Engineering, Global Academy of Technology, Bengaluru India.
[2]Professor and HOD, Department of CSE, Ballari Institute of Technology and Management, Ballari, India

DOI: https://doie.org/10.10399/JBSE.2026787101

Abstract. Over the past few decades, rapid advancements in software and hardware technologies have driven automation in various domains, with Java-based applications playing a crucial role in modern computing. However, many organizations continue to rely on legacy systems written in Java, C, and other older languages, making maintenance, comprehension, and migration increasingly complex. Frequent modifications in these systems often introduce structural inconsistencies, posing challenges for developers, especially in the absence of original contributors. Moreover, these legacy systems encapsulate critical business rules accumulated over decades, which must be preserved during migration to modern paradigms. Program slicing in software re-engineering offers an effective approach for understanding, modifying, and migrating essential code segments while reducing maintenance complexity. However, existing slicing techniques in software re-engineering frameworks struggle with Java's dynamic features, such as encapsulated classes, inheritance hierarchies, and dynamic method calls, leading to imprecise slicing and inefficient control flow analysis. Furthermore, current slicing tools often lack inter-procedural precision and fail to support legacy system refactoring. This paper proposes a novel and computationally efficient model-driven approach for abstracting key components and requirements from Java-based systems, thereby enhancing program slicing for seamless legacy-to-modern migration. The proposed method improves slicing precision by effectively handling variable references, dynamic bindings, and structural dependencies. Experimental evaluations demonstrate that this framework significantly enhances software adaptability while ensuring the integrity of embedded business logic. Unlike existing methods, the proposed approach optimizes computational complexity in search operations and facilitates efficient requirement extraction for modern software system development.

Keywords: Program Slicing, Object-Z Representation, Requirement Extraction, Java Programming Systems, Software Verification, Re-Engineering

Introduction

Over the past few decades, to meet increasing demand and supply goals, several businesses are highly relying on automation to optimize operations and improve service efficiency [1]. This shift in organizational activities further raised the high demand for developing software solutions, preferably using object-oriented programming languages such as Java, C++ etc [2].As the organizations evolved, these software applications, have undergone numerous need-based changes over time to meet ongoing business requirements. However, these incremental modifications had resulted in major changes in the software, making the code more unstructured, complex and difficult to maintain [3]. As a result, developers often struggle to understand and make updates to these software systems, specifically when the original design specifications and developers are no longer available in the same organization [4]. Therefore, understanding the code and migrating these software applications to new modernized paradigm while preserving their embedded business logic has become a critical challenge in software engineering [5].

It is important to note that while Java systems continue to function reliably, they struggle to keep pace with the advancements in modern computing technologies. Over time, continuous updates and inadequate maintenance have degraded their readability and comprehension, making them increasingly difficult to manage. Additionally, deriving software design specifications has also become a significant challenge for meeting modern software development requirements [6] [7].

Therefore, software engineers must formulate a systematic approach for re-engineering these systems to effectively derive the key functional attributes and software design requirements. The primary goal is to isolate functionally independent modules and extract abstract functionalities, for the ease of structured migration of

the Java systems to the contemporary technologies [8]-[10]. Thereby, to address the aforementioned challenges, the need arises for developing techniques that can systematically restructure and analyse Java programs to enhance their maintainability and also helps in deriving key design specifications from different layers of program abstractions. The popularity of program slicing has emerged as such decomposition approach, that narrow down the focus of attention to relevant parts of a program. The core idea of program slicing is to eliminate irrelevant statements from source codes while preserving the semantics of the Java program. It also plays crucial role in analysing functional dependencies within Java applications [11]-[12]. A slicing criterion can be defined as a pair of $\langle p, V \rangle$ where p refers to the program and V is a subset of program variables. This program slicing approach has been widely used in many software activities including software testing, debugging, re-engineering, program analysis and comprehension and so on [13].

The idea of static program slicing is still referred to analyse functional dependencies in Java programs as it helps understanding the structural aspects of a program without requiring execution, making it highly effective for re-engineering purposes. It's capability towards identifying the control and data dependencies also helps in refactoring unstructured Java code into modular and maintainable components and also helps effectively deriving software design and requirement specifications [14]. Unlike, dynamic slicing, static slicing is computationally inexpensive and do not much rely on actual execution traces and broader test cases. Also, static slicing approaches not only simplifies software maintenance, but also supports security analysis, debugging and performance optimization [15]. Its ability to provide complete view of dependencies without execution makes it highly relevant even in extracting modern software design requirements [15].

It has been observed that modern software applications, especially enterprise-level software and distributed systems are composed of millions of lines of code (LOC). Traditional slicing techniques mostly do not ensure adequate scalability factor due to high computational cost of analysing larger programs. The approaches mostly suffer from memory and performance bottlenecks and also handling precision vs. performance trade-offs still remain crucial and unsolved for real-time applications [16] [17]. The dynamic slicing approaches rely on specific execution traces, making it difficult to cover all possible program behaviours, especially in multi-threaded and concurrent systems. Also, hybrid approaches often struggles with computational overhead and complexity problems. Also, modern days Java programs contains dynamic method calls which makes it difficult for the traditional slicing approaches to trace control and data dependency effectively [18]. The problem of higher-order functions also complicates the dependency tracking while aspect-oriented programming also adds complexity to the slicing tools. Various tools use different intermediate representations (IRs) and dependency graphs, making the interoperability difficult [19]. Thereby, the need arises to design and develop a robust and computationally efficient software re-engineering framework that integrates precise dependency tracking, efficient control/data flow abstraction and real-time feasibility to reduce the software maintenance efforts, cost and modernization challenges.

The proposed work, thereby introduces a simplified and unique proposition of software re-engineering framework to abstract various components from the input Java programming system and also aims to facilitate seamless migration to new technologies or computing paradigms. 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:

- **Restructure Java Source Code** – Improve code maintainability and modularity for better abstraction.
- **Abstract Design Information** – Extract high-level architectural and functional details while preserving semantics.
- **Abstract Functional Dependencies** – Identify and analyze control and data dependencies within Java programs.
- **Identify Closure between Attribute Set and Program Slice** – Establish relationships between program attributes and their impact on slicing accuracy.
- **Abstract Software Specifications and Requirements** – Facilitate precise requirement extraction using formal methods, specifically the Object-Z formal method.

The working flow of the proposed framework integrates restructuring of the input Java program while eliminating the irrelevant attributes in the first stage whereas in the second stage, it extracts design information in the form of abstracted control flow and data flow representations. Leveraging this design abstraction approach

the framework further identifies the core functional dependencies and also narrows down to the essential functional dependencies which reduce the cost of computation and aid in the generation of program slices that serve as formal software specifications. The program slices are further identified and represented in formal notations which are further used to generate specifications. The computational framework introduces a unique combination of static and backward slicing approach which is designed based on the established static program slicing principle of Weiser (1981) [11]. However, the proposed work also addresses the limitations of over-approximates slices in the approach of Weiser (1981) [11] that leads to redundant code inclusion with a unique computational methodology that combines strength factors of both static and backward slicing approach.

The proposed re-engineering framework also combines computational processes of re-structuring, design information extraction, computation of data flow table (DFT) to compute the functional dependencies with reduced and optimized cost. The functional dependences further applied to a minimal cover algorithm (MCA) to compute the minimal attributes which influences the outcome of program slicing for extracting formal software specifications. In the proposed framework static-backward slicing approaches play crucial role in effectively deriving program abstraction and formal software specifications. Here the approaches basically analyse the dependencies and extract relevant portions of the Java program. The backward slicing approach here helps tracking the dependencies across control and data flow which also influence abstracting relevant components of Java program. It also computes only relevant and essential code fragments that simplifies the program comprehension. The backward slicing approach also isolates the functional logic and its scope lies in re-engineering the Java program to new architecture or paradigms. On the other hand the approach of static slicing helps defining the system behaviour in a structured manner and the program slices also serve as input for formal methods to verify the correctness. This approach eliminates irrelevant Java program codes which not only reduces the complexity but also enhance functional integrity.

The proposed software re-engineering framework is also realized as an automated tool for abstraction of the design information from the input Java program. The experimental outcome clearly shows that the automated tool can derive essential software characteristics from input Java program considering a set of optimized operations and also the proposed tool is tested for its correctness and completeness considering different set of programs of varying complexities and sizes. The outcome also shows that the computational framework has produced expected outcome for varying test conditions and for different given set of larger programs. The proposed slicing approach not only preserved the embedded business logic but also minimized the complexity in functional dependencies which has resulted in optimal set of attributes using which the Java program can be comprehended in a better manner. It also offers optimized slicing operations for extracting modern software design specifications.

The proposed work integrates static-backward slicing approach in a software re-engineering framework in deriving program abstractions and formal software specifications. It has got several advantages over the existing baseline/traditional re-engineering strategies. The contribution of the proposed work is listed as follows:

- *Cost-Effectiveness*: Traditional slicing strategies such as Dynamic Slicing, Hybrid Slicing and Machine Learning (ML)-based slicing approaches suffer from high computational costs and several other constraints. In the case of dynamic slicing, run-time dependency and execution monitoring make the process computationally expensive. Also, in the case of Hybrid slicing cost of computation increase due to test case generation and runtime analysis. Also, the ML strategies require high computation and dataset preparation costs which are avoided in the proposed static-backward slicing approach. The proposed slicing strategy doesn't rely on runtime execution and optimizes the operations purely on static Java program analysis, making the entire process very much cost effective.
- *Computational Complexity:* Existing dynamic slicing strategies in re-engineering suffers from high computational complexity as it tracks maximum possible runtime dependencies. In the case of hybrid slicing also the computational complexity is found comparatively very high due to combination of runtime monitoring and static analysis so as in the case of deep learning models for their dependency over complex feature extraction, training and validation paradigms. The computational complexity is significantly reduced in the proposed static-backward slicing approach as backward slicing limits the scope and static slicing helps fine-tuning and refining the results, making it more scalable for large-sized Java programs.
- *Inclusion of Data Flow Table:* The proposed tool influence Data Flow Table (DFT) on slicing for program abstraction which significantly enhances the accuracy, efficiency and comprehensiveness of program slicing. However, majority of the existing approaches rely on Data Flow Graphs (DFG) considering nodes (variables) and edges (dependencies) which can become more complex and harder to visualise if the Java program size increases. Also, the graph traversal operations often generate overhead in larger computing

systems. Also Control Flow Graphs (CFGs) and DFGs helps extracting relevant program components but CFGs and DFGs are not alone sufficient to deal with data dependencies spanning across different parts of the code. Thereby, inclusion of DFT in the proposed slicing approach complements CFG analysis as it systematically organizes the interactions among variables within the program improving the slicing accuracy. Inclusion of DFT in the proposed work also restrict false dependencies, making program abstraction more accurate. It also helps eliminating redundant codes in program abstraction and also aids in extracting functional specifications from Java program code.

- *Precision and Accuracy:* Even though in dynamic slicing approach precision is found quite higher for specific execution traces, but restricted by test coverage. Also, it has been observed that hybrid approaches mostly suffers from overestimation of dependencies even though offers high precision in program abstraction and extracting software design specifications. In ML-based slicing approaches the precision depends on quality of training inputs and also may introduce errors. However, the proposed static-backward slicing offers high precision in extracting formal design specifications as backward slicing ensures relevance and static slicing reduces irrelevant attributes from Java programs.
- *Scalability*: Unlike existing slicing approaches the proposed slicing strategy is highly scalable due to efficient static analysis without having runtime dependencies. On the other hand, dynamic slicing strategies pose limited scalability due to runtime constraints. ML models are found difficult to scale due to increased cost of computation, training overhead and model adaptability issues.
- *Suitability in Software Re-engineering:* The scope of re-engineering is found limited in the case of dynamic slicing as it relies on execution traces that makes it unsuitable for early-stage re-engineering and program abstraction for deriving design specifications. However, in hybrid strategy the complexity increases with system size. The scope of ML models is also found limited as these models must be trained for different software paradigms. The proposed work addresses these limitations and offers structured code abstraction and specification extraction with optimized flow of execution.

The findings of this research are expected to contribute significantly to the development of next-generation software systems. By integrating formal methods with program slicing in proposed software re-engineering framework, the approach facilitates automated specification derivation, thereby enhancing software reliability, maintainability, and adaptability.The organization of the manuscript is as follows: Section 2 presents a discussion of existing studies while a discussion of research methodology is carried out in Section 3. Section 4 illustrates the result accomplished while the conclusion is stated in Section 5.

Related work

Since dynamic slicing strategies has been evolved as backend technique for program debugging, fault localization and automated program repair. However, it has been observed that State-of-the-art dynamic slicing techniques struggle in terms of efficiency and scale. These approaches are known to suffer when execution size, the number of executed instructions grows. The authors Postolski *et al.* [2] presented a strategy of dynamic slicing considering on-demand re-execution approach. The presented approach basically progressively constructs a slice while acquiring targeted information corresponding to dependencies through repeated execution flow in the same program. The evaluation strategy considers comparing both on-demand re-execution and execute once paradigms in terms of overall performance and also study the gains and losses and these factors are affected by execution and slice size. Specifically, the comparative analysis is performed against Javaslicer [20] and Slicer4J [21]. The implementation idea flows Java programming language implementing a static dependency analysis considering CodeQL and a dynamic instrumentation for frontier tracking. The evaluation approach on the SV-COMP benchmark and Antrl4 unit tests also avoids re-compilation task per re-execution. The authors claim that on-demand re-execution approach can provide high performance gains for dynamic slice computation particularly when slice size is small and the execution size is large. However, the approach of on-demand re-execution incorporates multiple re-executions of the program for every slicing request leading to high computational overhead and performance bottlenecks, particularly for large-software systems in real time environments. The approach also fails to deal with large-scale software systems with deep dependencies and extensive branching. Also, programs with non-deterministic behaviour might produce inconsistent results across different execution scenarios with this dynamic slicing approach.

Vol. 23, No. 1, (2026)
ISSN: 1005-0930

The authors (Soifer*et al.*) [22] also focused on dynamic slicing for large scale software applications. The authors presented an Abstract Memory-Model that completely replaces memory reference registering and processing. The model operates with program symbols. The study considers implementation of an abstract dynamic slicer for C#, DynAbs and also the evaluations shows how largest and modern C# applications that can be found in GitHub can be sliced for their test cases in few minutes. The presented approach also shows that minimizing the code-to-be-sliced focus can speed up the computations and also ensures marginal relative precision loss. Overall the study attempted to bring a well-balance between precision and complexity trade-offs considering abstract representation of memory. The complexity associated with tracking variable states, object interactions and heap-memory modifications in Java applications significantly reduced which also influences better optimization of memory usage and computation time. It is also observed that the presented approach of abstract memory model enhances precision and relevance. However, in certain cases the complexity of abstract memory models increases due to additional pre-processing and analysis overhead and also its dependency over execution context restricts their generalizability across various execution environments. The study also has not much talked about handling multi-threaded larges-sized Java programs and thread synchronization issues still remain challenging.

The authors (Stoica*et al.*) [1] also identified the limitations in dynamic slicing despite an extensive body of research showing its usefulness. The authors claim that dynamic slicing is still short from production-level use due to high cost of runtime instrumentation. The authors further introduce a hybrid dynamic-static slicing approach to create program slices. It leverages hardware support for control flow monitoring and also presents a strategy of cooperative heap memory tracing to reduce the overhead typically associated with dynamic slicing. The method offers significant reduction in runtime overhead while ensures more practical approach in analysing deployed software programs. The experimental outcome shows that the presented approach recovers 94% of the program statements in dynamic slice while incur only 5% runtime overhead. However, in many varying instances, the strategy suffers from high computational cost due to test case generation and complex runtime analysis. The strategy also struggles with overestimation of dependencies.

The authors (Zhang *et al.,* 2021) [14] also introduced symbolic program slicing approach that relies on symbolic execution and can be computationally expensive for large and complex Java programs that consist numerous execution paths. This may also lead to scalability issues. SymPas approach also vulnerable to path explosion problems, especially in the case of very large codebases. Here, the slicing mechanism also do not handle the loops and recursive function calls, making precise slice extraction very challenging. Even though SymPas improves the slicing accuracy but it does so at the cost of increased computational effort. The authors (Seghir *et al.,*2018) [15] also presented a Data-Flow Guided Slicing strategy that struggles with complex control structures in Java programs which impacts the accuracy of slice extraction for deriving effective design specifications. The presented method does not fully capture Java's dynamic constructs and also heavily depends on data-flow analysis which is computationally expensive for large-scale Java applications. This also resulted in performance bottlenecks. The accuracy of slices depends on precise data flow information. Thereby, the presented method's fuzziness in data-flow analysis leads to incomplete and incorrect slices computation which also affects the reliability.

The authors (Vidziunas*et al.,*2024) [23], (Stiévenart*et al.,*2023) [24] also talked about similar problems in the program slicing approaches and also struggled with additional computational overheads in large-scale Java applications with extensive object-oriented features. However, the approach in (Vidziunas et al.,2024) [23] mostly emphasized on automated program repair whereas mostly ignored the importance factors of functional dependencies towards program slicing in large-scale Java applications. The presented method could incur challenges while adapting similar reduction techniques for slicing Java programs. The authors (Yadavally *et al.,* 2024) [25] introduced a predictive program slicing approach via execution knowledge-guided dynamic dependence learning. The method heavily relies on execution data via knowledge and dynamic dependence learning which restricts its applicability in the crucial scenarios where execution data could be incomplete, unavailable or difficult to collect. This learning-based predictive slicing approach also incur high computational cost while applying to large Java applications with deep dependencies and complex control flows. The predictive program slicing method also do not ensure effectiveness towards handling object-oriented features in Java programs. Since the method is trained on specific datasets, hence struggle to generalize effectively to the unseen Java applications which involve unique coding patterns or domain-specific architectures. Also, the predictive slicing approach performance could be affected due to risk of overfitting to training data, leading to reduced effectiveness in deriving precise formal design specifications with high computation cost. The slicing approaches by (Soremekun *et al.,* 2021) [26], (Yadavally *et al.,*2024) [27] and (Shahandashti *et al.,* 2024) [19]

Vol. 23, No. 1, (2026)
ISSN: 1005-0930

also worked in the similar line of research to derive precise formal design specifications of software considering Java programs using machine learning (ML) approaches. However, the study of (Yadavally *et al.,*2024) [27] mostly encouraged the scope of static program slicing towards improving the handling of implicit dependencies in object-oriented Java programs. The authors also claim that static program slicing could be effective in identifying non-trivial dependencies in complex Java programs.

The authors (*Homerding* et al.) [28] introduced parallel semantics program dependence graph (PS-PDG) which addresses the limitations of traditional PDGs in parallel programming contexts. PS-PDG basically extends the functionalities of PDGs by incorporating parallel execution semantics that can easily adapts to the formal representation of program slices. Since PS-PDG mostly emphasized towards parallel program optimization hence, their full-fledged adaptiveness towards program slicing remains questionable and also its semantics-based execution plan may introduce overhead when formally extracting slices. While mapping PS-PDG with Object-Z, formal abstraction may disregard inter-thread dependencies if not modeled explicitly. PS-PDG also suffers from increased computational complexity in analysing large-scale concurrent Java applications. (*Yadavally* et al.) [30] presented NeuralPDA for program dependence analysis and formal specification extraction. The approach of NeuralPDA basically leverages deep learning to predict program dependencies with high accuracy which also helps in deriving the formal representation of program slices. While traditional approaches rely on rule-based dependency graphs, but NeuralPDA offers AI-based modeling leading to more precise specification extraction from Java code. However, the presented approach relies on deep learning models which adopts inherent black-box nature which makes it less deterministic and difficult to validate against formal specifications. Formal methods such as Object-Z require explicit and verifiable models in which AI-generated dependence graphs are less interpretable. Therefore, researchers still struggle to verify AI-generated slices mathematically. NeuralPDA also requires large annotated datasets making the system resource-intensive for complex Java systems.

Thereby, to address these limitations, there is a need to design and tailor a simplified and computationally efficient effective software re-engineering framework that could accurately capture implicit control and functional dependencies while minimizing redundant computations towards extracting precise formal software specifications from Java programs yet retains a balance between precision vs performance trade-offs.

The proposed re-engineering framework aims to integrate program restructuring, design information extraction, and Data Flow Table (DFT) computation to efficiently extract functional dependencies at an optimized cost. It further aims to employ the Minimal Cover Algorithm (MCA) to refine functional dependencies by identifying minimal attributes. Additionally, the framework applies an effective backward slicing approach, ensuring a structured formal representation using the Object-Z method, which aids in deriving precise software specifications and requirements which is further elaborated in the next section.

Research Method

The study analytically models the softwarere-engineering framework to integrate efficient algorithms for restructuring Java source code, abstract design information, and abstract functional relationship, identify closure between attribute set for precise program slice extraction and abstraction of software specifications and requirements. The proposed work offers a simplified and computationally efficient re-engineering framework that integrates a novel static-backward slicing approach which is analytically tailored with a series of effective computing steps to extract precise formal software specifications from Java programs. The proposed method got inspiration from the baseline static slicing approaches in software re-engineering towards analysing the functional and control dependencies without relying on program execution. It incorporates effective data-flow analysis and dependency tracking mechanisms which not only minimizes irrelevant program slices but also improve abstraction accuracy and enhance automation in deriving formal specifications. This will significantly influence software comprehension verifications and seamless migration of the large and complex Java applications to modern computing paradigms (Fig. 1).

Vol. 23, No. 1, (2026)
ISSN: 1005-0930

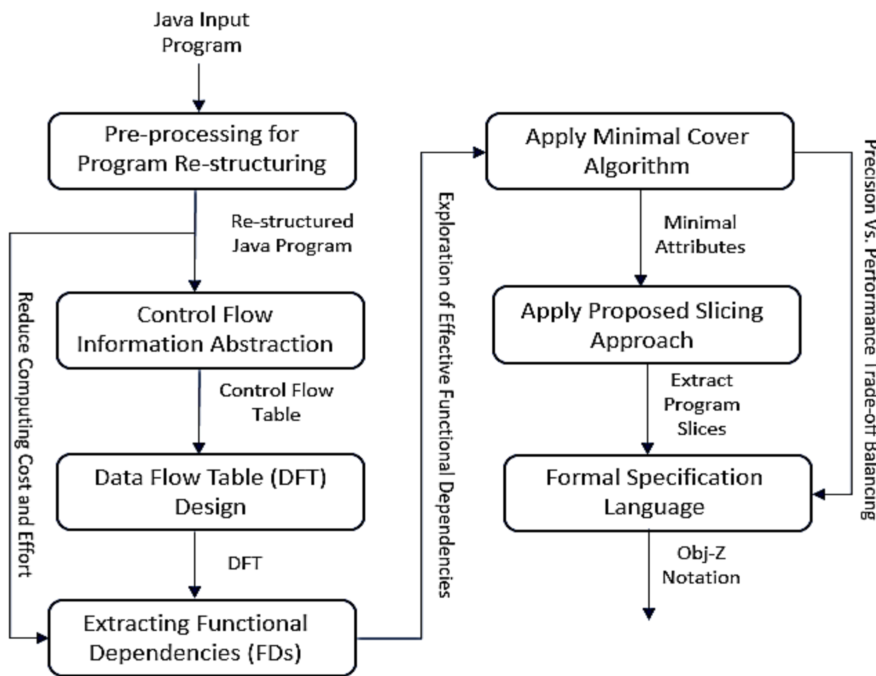


Fig. 1. Proposed Architecture for Re-Engineering Modeling

The proposed formal abstraction framework in re-engineering modeling is analytically designed using a set of computing processes to enhance program slice abstraction through efficient operations with the Object-Z formal method. Object-Z is used to formally model slicing behavior, aiding in precise specification and requirement extraction for software re-engineering. The framework employs computationally efficient processes to eliminate irrelevant statements while preserving program semantics. It also provides an unambiguous specification of the sliced program, facilitating debugging, testing, and verification. By integrating Object-Z with computationally efficient operations, the framework derives precise formal software specifications by abstracting key program properties. Additionally, it ensures functional correctness while balancing the trade-off between precision and computational performance for large Java programs. The proposed re-engineering framework also reduces the presence of over-approximate slices and redundant code inclusions with precise dependency abstractions. It also offers a simplistic approach to handle object-oriented constructs in Java programs and solves the scalability problems for large codebases.

The proposed software re-engineeringframework is realized as a tool involves a set of core computing processes viz. restructuring input program with pre-processing stage, abstraction of control flow information, abstraction of data flow table (DFT), abstraction of functional dependencies from DFT, obtaining minimal cover of the functional dependencies, extraction of the backward slice from the minimal cover attributes and finally representation of the program slice using the formal specification language. 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 Figure 1 shows the block-based architecture of the proposed software re-engineering approach for effective requirement extractions.

3.1 Pre-processing stage for Java Program Restructuring

The proposed computational framework integrates 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 eliminating 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. And this input java program can be

represented as sequence of lines using Equation (1) and the transformation proves is also defined as a function using Equation (2).

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

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

Here L_i represents the i-th line in the source code and P'_{java} refers to the reconstructed java program. The program restructuring approach in the proposed work initially perform appending of packages, classes and interfaces to an input file of transformed java program and it is performed with the function of $A(P_{java})$. The mathematical expression using Equations (3 to 5) representing this flow. Let $\varepsilon(P'_{java})$ be the set of all comment lines in P'_{java}. Further the computing process eliminate the comment lines while only consider the executable statements and ignore the comment lines in P'_{java}. Here, $\varepsilon(P'_{java}) = \{L_i \in P'_{java} | \forall L_i \in sL_C, mL_C\}$. The computing is further performed using Equation (6).

$$P'_{java} = A(P_{java}) \quad (3)$$
$$\text{or} P'_{java} = P_{java} \cup S(P_U, C_U, I_U) \quad (4)$$
$$\text{or} P'_{java} = P_{java} \oplus S(P_U, C_U, I_U) \quad (5)$$
$$P''_{java} = P'_{java} \backslash \varepsilon(P'_{java}) \quad (6)$$

Here 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 the following expression (7).

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

In the above equation (7) 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 blanck lines whereas $\backslash$ 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 single statement considering the following mathematical expression (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, char etc. It considers the data type shared by all variables being declared. The concatenation operation symbolically join items together into one syntactic unit. Also the term $\{v_i\}_{i=1}^{\xi}$ is a tuple of variable identifiers where ξ represents the total number of variables being declared. For multiple data types the transformation is applied independently for each type considering the following mathematical expression (9).

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

Here $\cup(*)$ ensures each type's variables are clustered separately and ξ_i is the number of variables of Type_i.

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

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

In equation (10) 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. →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 identify and remove the unused variable and make the transformed Java program $\mathrm{P}''_{\mathrm{java}}$ more suitable for further analysis. The program re-structuring is modeled here with 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.

3.2 Abstraction of Control Flow Information from Re-structured Java program

The proposedwork alsoconsiders abstraction of control flow information from the re-structured Java program. After analysing the baseline studies, it has been observed that control flow abstraction plays a crucial role in program slicing. Thereby, the proposed work customizes a module for abstraction of control flow information in static-backward slicing approach for deriving formal software specifications. Here this approach explores and traces dependencies among statements such as conditional branches and function invocations in re-structured Java program $\mathrm{P}''_{\mathrm{java}}$ and helps ensuring accurate backward slicing in consecutive steps of the work as without control flow abstraction, program slicing might yield incomplete and incorrect slices, if the Java program structure varies. Precise slicing approach require knowledge of execution paths dictated by conditional and looping constructs. Thereby the proposed work enables this step of computation to identify reachable and dependent statements that also improves the slice precision towards deriving the formal specifications. Thereby it is a crucial step of computation, serves as a foundation for effective program slicing, efficient extraction of dependent statements while maintaining integrity of the Java program execution and embedded business logic.

The core analytical steps associated with the abstraction of control flow information from $\mathrm{P}''_{\mathrm{java}}$ in proposed work represent $\mathrm{P}''_{\mathrm{java}}$ as a sequence of statements such as $\mathrm{P}''_{\mathrm{java}} = \{s_i\}_{i=1}^{k}$ where each statement s_i belongs to one of the following categories.

- Conditional Statement:$s_i \in c_s$ where c_s represents conditional statements in $\mathrm{P}''_{\mathrm{java}} = \{s_i\}_{i=1}^{k}$
- Computational Statement:$s_i \in comp_s$where $comp_s$ represents computational statements
- Function Invocations:$s_i \in f(*)_{call}$, where $f(*)_{call}$ represents function calls or invocations in $\mathrm{P}''_{\mathrm{java}}$

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 Equation (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 control statement is true for particular inputs and fourth column Tr_2 represents the statement where the false statement gets executed as shown in the Table 2.

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

This tabular representation in the proposed work captures the control flow transitions in the form of CF_I within $\mathrm{P}''_{\mathrm{java}}$ and further helps formulating Data Flow Table (DFT) in the control flow order. The control flow table thereby created after exploring the design information from the Java program. However, the proposed work addresses the limitations in Data Flow Graphs as they are mostly complex with graph embeddings and hard to visualize in large programs and also tends to introduce slicing errors. It also requires complex graph traversal with an increased complexity of $O(n)$ or $O(n^2)$ where as applicable to program-specific structures only. Here n represents number of statements of line of code (LOC) in Java programs. The proposed slicing approach addresses this limitation and considers DFT which is highly structured and easy to implement. Also,

it offers high computational efficiency due to fast lookup and filtering with complexity of $O(1)$or $O(\log(n))$. It also offers high-precision, cross-language compatibility and can be easily processed in structured models for large-scale Java software applications.

3.3 Abstraction of Data Flow Table (DFT) in Control Flow Order

The study realises that the data variables in the program is based on the control flow order and further the proposed work considers $CF_I(\mathrm{P''}_{\mathrm{java}})$ of $\mathrm{P''}_{\mathrm{java}}$ and aims to formulate the DFT in the order of control flow. The proposed work considers DFT as a structure tabular representation in the form of T_{DFT} in which each row corresponds to a program statement and also consists of three attribute susing in Equation (12) as follows.

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

In the above equation 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 longer run influences the slicing approach as well.

3.4 Abstraction of Functional Dependencies from DFT

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 variable. In the proposed work the functional dependency in program $\mathrm{P''}_{\mathrm{java}}$ is mathematically represented using Equation (13) as a relation of $\mathfrak{f}$ which is extracted from the $T_{DFT}(i)$.

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

In the above expression of (13), L'_i refers to the defining program statement where the v_d^i is defined whereas L'_j refers to the dependent program statement line where the v 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 identifying 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.

3.5 Finding Minimal Cover of the Functional Dependencies

In this computing step the proposed work aims to minimize $\mathfrak{f}(L, v)$ such as $\mathfrak{f} \rightarrow \mathfrak{f}_{min}$ while eliminating the redundant dependencies. This approach ensures that only minimal set of functional attributes should be extracted from $\mathfrak{f}(L, v)$ which are essential for program slicing. The study assumes that if $\langle L'_i, L'_j, v_c\rangle$ is considered redundant functional dependencies that means that v_c appears in multiple functional dependencies with different program statement numbers. However, for program slicing only few significant dependencies are required to extract minimal cover attributes. Minimization is done to remove the redundant variables occurring in the code. This redundancy happens during the maintenance phase of the software as it is handled by the multiple developers. To avoid the ambiguity and to attain clarity in the program, these functional dependencies have to be minimized to get minimal attributes. These minimal attributes are used for program slicing in the next step. The minimal cover rules eliminate redundant dependencies if both $\langle L'_i, L'_j, v_c\rangle$ and $\langle L'_j, L'_k, v_c\rangle$ exist and $\langle L'_i, L'_k, v_c\rangle$ can be derived then eliminate$\langle L'_j, L'_k, v_c\rangle$. If dependency contains a variable that is not required for slicing then minimal cover algorithm removes that variable as well to optimize the computational flow.

This approach also helps reducing the computational complexity and cost in proposed static-backward slicing approach. It also improves the clarity and maintainability of larger Java programs.

The proposed re-engineering framework enables a set of effective computing steps to derive minimal attributes in formal abstraction framework as shown in the Fig.2.

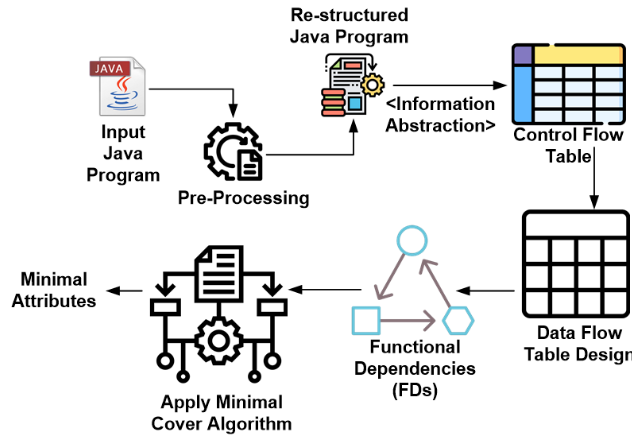


Fig. 2. Computation of Minimal Attributes in Software Re-Engineering Framework

The proposed work minimizes functional dependencies $f(L, v)$by eliminating redundancies, ensuring only essential attributes are used for program slicing. Redundant dependencies arise when a variable appears in multiple functional dependencies with different statements, often due to software maintenance by multiple developers. The minimal cover algorithm further removes non-essential variables, optimizing computational flow and reducing complexity in static-backward slicing. This improves maintainability and efficiency in abstracting formal representation from Java program slices.

3.6 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 variables at specific program points. In the proposed work the customized steps further improve Weiser's approach for enhancing the backward slicing process in DFT Coverage (DFT-COV) Slicing.

Algorithm: Improved DFT-COV Slicing Using Minimal Cover Attributes

Input:f_{min} , $T_{DFT}(i)$, P''_{java}
Output: $S(v) \in P_{slice}$
Begin
1. Init $\langle L_n, v\rangle$, $S = \{L_n\}$
2. Enable Backward Traversal Using Minimal Cover Attributes
3. **For** each $\langle L'_i, L'_j, v_{slice}\rangle \in f_{min}$
4. **Check If**$L'_j \in S$
5. add$L'_i \rightarrow S$ (L'_i contributes to L'_j)
6. **End**
7. **End**
8. Repeat (Line-3 to 7) recursively → until no new statements are added
9. Extract the statements from the P''_{java}
10. Using $S = \{L_n\}$ extracts the statements from P''_{java}
11. Store $S \leftarrow S(v) \in P_{slice}$
12. Repeat for all minimal cover attributes
13. **Return**$S(v) \in P_{slice}$
End

Here $\mathfrak{f}_{min}$ refers to the minimal functional dependencies in the form of $\langle L'_i, L'_j, v_c\rangle$ from above computing steps. Here L'_i represents the defining variable v, L'_j represents the statement that refers to v and v_{slice} refers to the set of slicing variables. In the algorithm the process initializes the slicing criterion where L_n refers to the last occurrence of v_{slice}. Also, it initializes the slice $S = \{L_n\}$, this enables initializing the slice for a given variable. The proposed work computes the backward slices considering the minimal cover attributes using the above optimized algorithm. The proposed slicing called as backward slicing searches and extracts only those program statements, which are contributing to the calculation of the slicing variable and it can be represented with the following mathematical expression of (14). The above algorithm 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.

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

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 represets the set of all lines included in the program slice. $L_n$refers to the slicing criterion line and $\mathfrak{f}_{min}$helps deriving the minimal set of dependencies relevant to v_{slice}.Further the algorithm also enables optimized traversal considering the mathematical expression (15).

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

Here S epresents the current slice i.e. the set of lines influencing the value of target attribute v_{slice}, the minimal set of relevant data dependencies also explored via $\mathfrak{f}_{min}$. Here $\exists$ implies existence and further the process keep 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}.

3.7 Representation of the Program Slice using Formal Specification Language

Further the extracted backward program slices $S(v) \in P_{slice}$ are represented using Object-Z notation ensuring mathematical precision and structured representation. The Object-Z representations gives more precise, mathematical and unambiguous representation of the program. In our methodology, only the required constructs like inputs, outputs, pre-conditions and post-conditions statements of Object-Z notations are used, which are mandate for extracting the software specifications. In Object-Z representation, the input variables are represented using the '?' symbol and output variables are represented using '!' symbol and the derived variables are computed using input variables, represented using the delta symbol and further using these formal representations, the software can be comprehended by abstracting the specifications and user requirements.

The program slice is defined as $S_c \subseteq P$. Here S_c consists of the statements relevant to the slicing criterion C. Here S_c refers to the minimal subset of statements that affects $v \rightarrow l$. Here v refers to variables and l refers to the location of P. In the proposed formal method the Object-Z notation for slice representation follows a standard slice schema of [19] and derive the proposed slice schema as S_δ as shown in expression (16)

$$S_\delta \triangleq \mathfrak{f}(?:\mathbb{V}, !:\mathbb{V}, \Delta_s, P(\mathbb{V}), Q(\mathbb{V})) \quad (16)$$

Here $\mathbb{V}$ represents set of program variables where as Δ_s implies the state changes in S_c. The pre-condition here ensures input validity as $In(?:\mathbb{V}) \Longrightarrow \exists S_c, S_c \subseteq P$. Here the post condition ensures the output correctness as $Out(!:\mathbb{V}, \Delta_s) \Longrightarrow \exists S_\zeta$. Here S_ζ represents the correct slice transformation. This formal method in the proposed work basically helps extracting the software requirements and specifications considering pre-conditions which ensure valid inputs to sliced function, post conditions that ensure expected outputs to the inputs and also preserve the software logic while abstracting the specification of the program slice. The proposed formalized Object-Z schema which follows the baseline principle of [19] can be used for complex software verification, migration and maintenance.

Vol. 23, No. 1, (2026)
ISSN: 1005-0930

3.8 Case Study using Large Java Programs

In order to evaluate above the methodology of software re-engineering, this case study aims to apply the algorithms in real-world Java programs of different sizes. The proposed study evaluated the system for different programs, however this segment shows the case study for Student.java program that consists n number of student's personal information including enrolment, grades computation and academic performance tracking. The aim is to analyse the functional dependencies and maintainability where the structure of the program contains of multiple independent classes, methods and key-functionalities. The considered program basically undergoes all the necessary computing steps in the context of effective static-backward program slicing as mentioned in the mentioned Fig.1. Firstly, the program Student.java is pre-processed and restructured as shown in the following Fig.3.

```
1   import java.io.*;
2   class Stud {
3       String susn;
4       String sname;
5       String  sbranch;
6       Long  sphone;
7   float savg;
8       void sread ()
9   {
10      Scanner in = new Scanner (System.in);
11          System.out.println("Enter USN:");
12          susn = in.nextLine ();
13      System.out.println("Enter Name:");
14          sname = in.nextLine ();
15      System.out.println("Enter Branch:");
16          sbranch = in.nextLine ();
17      System.out.println("Enter Phone Number:");
18          sphone = in.nextLong();
19       scalc-Avg ( );
20      }
21      void print ()
22  {
23      System.out.println(usn + "\t| " + name + "\t| " + branch + "\t| " + phone+"\t"| +grade);
24      }
25  Void  scalc-Avg ( )
26  {
27  int M1, M2, M3, savg=0;
28      System.out.println("Enter marks:");
29      M1 = in.nextLine();
30  M2 = in.nextLine();
31  M3= in.nextLine();
32  savg= (M1+M2+M3)/3;
33   If (savg>75)
34      System.out.println("student is meritorious:");
35  }
36  }// class ends
37          Public static void main (String args [])
38  {
39              int i, n;
40              Student  s[]=  new Student[100];
41              Scanner in = new Scanner (System.in);
42              System.out.println("How Many Students Details you want to Enter:");
43              n = in.nextInt();
44              System.out.println("Enter details of " + n + " students\n");
45              for(i = 0; i<n; i++)
46  {
47              s[i]= new Student ();
48              s[i]. read ();
49          s[i].print ();
50              }
51
        System.out.println("****STUDENTS DETAILS ARE****\n");
52              System.out.println("USN \t\t| NAME \t\t| BRANCH \t\t| PHONE |\n");
53
    }//main ends
```

Fig. 3. Restructured Java Program Student,java

The proposed system further performs abstraction of control flow information from the above restructured Java program as shown in the following Table 1. The Table 1 basically consists of four columns which are Start(s), End (E), Trans1 (tr1) and Trans2 (tr2) respectively. Here these attributes basically represent the control flow and data dependency transitions within Java program, likely extracted for DFT abstraction which further also influences the program slicing analysis. This structured mapping also can be utilized for backward slicing to determine the formal specifications of software.

Table 1. Control Flow Table (CFT) of the Java program Student.java

Start(S))	End(E)	Trans1(tr1)	Trans2(tr2)
1	45	46	51
46	48	8	49
8	19	25	20
25	33	34	36
34	36	20	
20	20	49	
49	49	21	50
21	24	50	
50	52E		

Here the CFT abstraction performs program scanning in structured recursion-based execution model that systematically enables initial transitions whereas efficiently manages handling the function calls within the looping structure. The execution also follows nested function invocation flow, traversing through conditional branches, while dynamically updating transitions states. The CFT table also tracks each function invocation which also ensures that dependencies and control paths are accurately maintained. Once, all the nested invocations are processed then the control resumes at the corresponding return points, preserving execution integrity. This structured approach enhances program comprehension, optimizes slicing precision, and ensures efficient handling of complex dependencies in large-scale Java programs. The proposed work further also evaluates DFT in control flow order which is also represented with the following Table 2.

Table 2. Data flow table for the above restructured program

Line no	Def-var	Ref-var
1-3	stud	
5	Sread ()	
9	susn	
11	sname	
13	sbranch	
15	sphone	
16		scalc_Avg ()
18	sprint ()	
20		sname, sbranch, sphone
22	scalc_Avg ()	
25	M1	
26	M2	
27	M3	
28	savg	M1, M2, M3
29		savg
36	S []	Student ()
39	n	
41	i	n
43	S[i]	Stud ()
44	S[i]	sread
45	S[i]	sprint
48		susn, sname, sbranch, sphone

The DFT in Table 2 represents the definition (Def-var) and reference (Ref-var) of variables in the considered Java program Student.java. Here the DFT plays a crucial role in effectively tracking the defined and referenced variables that ensures clear identification of data dependencies across the program. It also provides structured insights into the function calls and variable interactions, improving the readability of the program's execution flow. The clear mapping of variable usage also allows for performance optimization, redundancy elimination and better modularization of the above Java program. In the next section the study further offers analysing the results obtained from the proposed tool and further the comparative analysis is also shown to prove the effectiveness of the proposed tool in handling modern Java programming paradigms.

Results and discussion

The experimental evaluation of the formulated tool is performed for various Java programs such as ATM, student grading, KCET etc. considering the IDE of PyCharm. Here the proposed algorithms are scripted in a single file implementation considering Python. The evaluation framework considers internal memory of 12GB and 64-bit windows operation system supported with x64-based processor. For experimental evaluation the study considers a Java program for finding quadratic roots which is shown in the below Fig.4.

The experimental results are further shown for Java program for finding quadratic roots is illustrated in Fig,5. The Fig.5(a) clearly shows the Control Flow Table (CFT) as obtained from evaluating the mentioned Equation (11) for Java program of quadratic roots. It also shows the Data Flow Table (DFT) in Figure 5(b) for the same Java program.

The Quadratic Java Program is shown in Fig.6. In Fig.6(a) the functional dependencies are extracted from the DFT which are further used in extracting the backward slices for given slicing variable as shown in Fig. 6(b). This variable is further used in deriving the formal software specifications which is shown in the following Figure 5.

Vol. 23, No. 1, (2026)
ISSN: 1005-0930

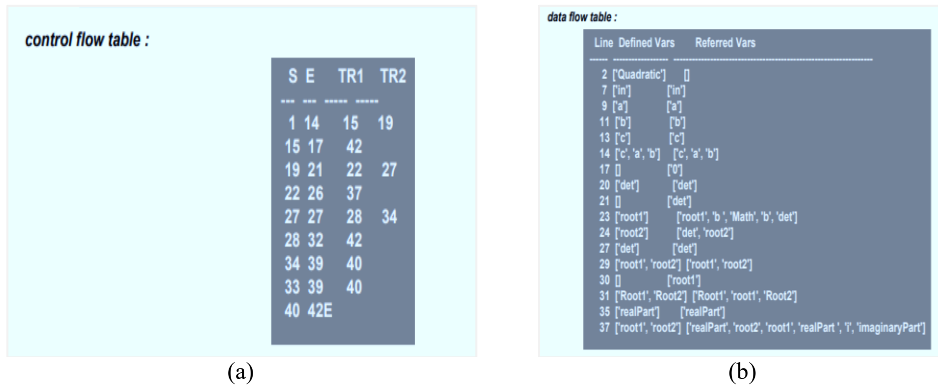
The software specification depends on the functional dependencies existing between the referred and defined variables in object Z form. The natural language specifications are designed using the Object Z representation. These specifications are further used to abstract the user requirements. The proposed software re-engineering framework finally derive the formal specification representation of the extracted backward slice which is also shown in the following in Fig.7.

The analysis of the CPU utilization in Fig.8 shows that the proposed re-engineering framework offers non-linear and fluctuating CPU utilization as the input size of the programs increases. The low CPU spikes for many input sizes (especially between 10–20 and 30–40) indicate the efficiency of using minimal cover attributes in backward slicing. This confirms that irrelevant and redundant statements are effectively ignored, reducing computational overhead. Sharp spikes (e.g., at input sizes 7, 14, 21, and 31) likely correspond to higher functional nesting or complex control flows (loops, nested calls). Even in these cases, the CPU usage remains within a modest range (max ~22%), suggesting the tool is scalable and robust for larger programs. The use of a structured CFT abstraction helps in optimizing execution flow, which reduces the CPU load in most cases by avoiding full program scans. Compared to dynamic slicing, which requires execution traces and has higher runtime costs, this approach is more predictable and lightweight.ML-based slicing may show better predictive slicing but often involves model training and inference, making them CPU-intensive, whereas the proposed tool uses deterministic logic with minimal compute.

```
import java.util.Scanner;

public class Quadratic {
    public static void main(String[] args) {
        double a, b, c, det, root1, root2;
        Scanner in = new Scanner(System.in);

        System.out.println ("Enter the value of a:");
        a = in.nextDouble();

        System.out.println("Enter the value of b:");
        b = in.nextDouble();

        System.out.println("Enter the value of c:");
        c = in.nextDouble();

        if (a == 0 || b == 0 || c == 0) {
            System.out.println("Invalid input: Enter non-zero coefficients");
            System.exit(0);
        } else {
            det = b * b - 4 * a * c;

            if (det > 0) {
                root1 = (-b + Math.sqrt(det)) / (2 * a);
                root2 = (-b - Math.sqrt(det)) / (2 * a);
                System.out.println("The roots are real and distinct: root1 = " + root1 + ",
root2 = " + root2);
            } else if (det == 0) {
                root1 = root2 = -b / (2 * a);
                System.out.println("Roots are real and equal: root1 = root2 = " + root1);
            } else {
                double realPart = -b / (2 * a);
                double imaginaryPart = Math.sqrt(-det) / (2 * a);
                System.out.println("Roots are complex and imaginary");
                System.out.println("root1 = " + realPart + " + i" + imaginaryPart);
                System.out.println("root2 = " + realPart + " - i" + imaginaryPart);
            }
        }

        in.close();
    }
}
```

Fig. 4. Input Quadratic Java Program

The Fig.9(a) shows the throughput increases steadily as input size grows, showing that the system maintains scalability and high performance even with larger programs.This validates the efficiency of the slicing algorithm in handling increased complexity without degrading performance in proposed software re-engineering framework. The peaks reaching nearly 400 operations/sec implies optimal code traversal.



control flow table :

S	E	TR1	TR2
1	14	15	19
15	17	42	
19	21	22	27
22	26	37	
27	27	28	34
28	32	42	
34	39	40	
33	39	40	
40	42E		

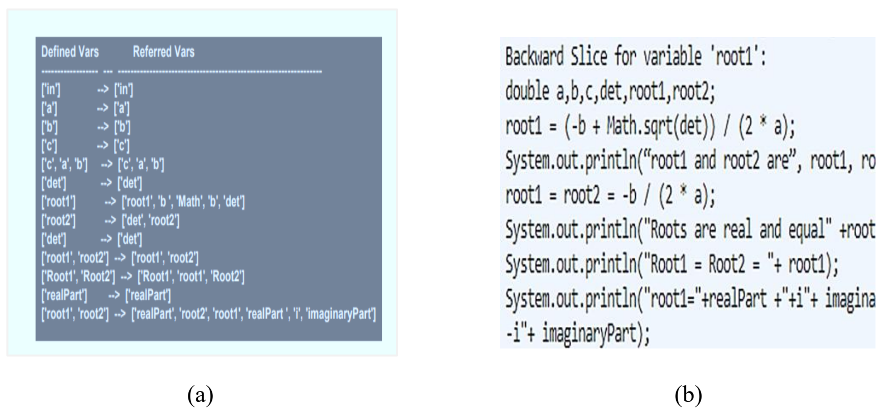
(a)

data flow table :

Line	Defined Vars	Referred Vars
2	['Quadratic']	[]
7	['in']	['in']
9	['a']	['a']
11	['b']	['b']
13	['c']	['c']
14	['c', 'a', 'b']	['c', 'a', 'b']
17	[]	['0']
20	['det']	['det']
21	[]	['det']
23	['root1']	['root1', 'b ', 'Math', 'b', 'det']
24	['root2']	['det', 'root2']
27	['det']	['det']
29	['root1', 'root2']	['root1', 'root2']
30	[]	['root1']
31	['Root1', 'Root2']	['Root1', 'root1', 'Root2']
35	['realPart']	['realPart']
37	['root1', 'root2']	['realPart', 'root2', 'root1', 'realPart ', 'i', 'imaginaryPart']

(b)

Fig. 5. Quadratic Java Program contains (a) CFT and (b) DFT

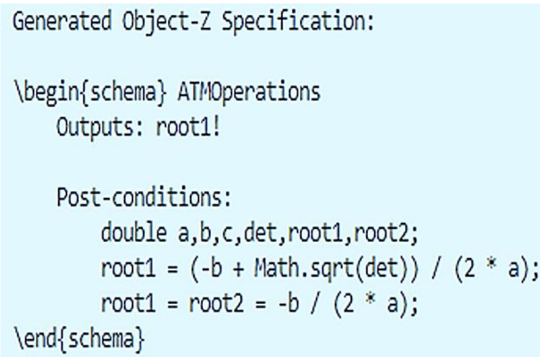


Defined Vars		Referred Vars
['in']	-->	['in']
['a']	-->	['a']
['b']	-->	['b']
['c']	-->	['c']
['c', 'a', 'b']	-->	['c', 'a', 'b']
['det']	-->	['det']
['root1']	-->	['root1', 'b ', 'Math', 'b', 'det']
['root2']	-->	['det', 'root2']
['det']	-->	['det']
['root1', 'root2']	-->	['root1', 'root2']
['Root1', 'Root2']	-->	['Root1', 'root1', 'Root2']
['realPart']	-->	['realPart']
['root1', 'root2']	-->	['realPart', 'root2', 'root1', 'realPart ', 'i', 'imaginaryPart']

```
Backward Slice for variable 'root1':
double a,b,c,det,root1,root2;
root1 = (-b + Math.sqrt(det)) / (2 * a);
System.out.println("root1 and root2 are", root1, ro
root1 = root2 = -b / (2 * a);
System.out.println("Roots are real and equal" +root
System.out.println("Root1 = Root2 = "+ root1);
System.out.println("root1="+realPart +"+i"+ imagina
-i"+ imaginaryPart);
```

(a) (b)

Fig. 6. Quadratic Java Program contains (a) Functional Dependencies and (b) Backward slice of slicing variable



```
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}
```

Fig. 7. Formal specification representation of the backward slice in proposed Re-Engineering framework

Vol. 23, No. 1, (2026)
ISSN: 1005-0930

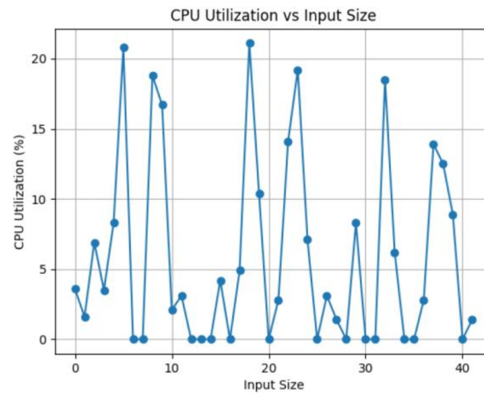
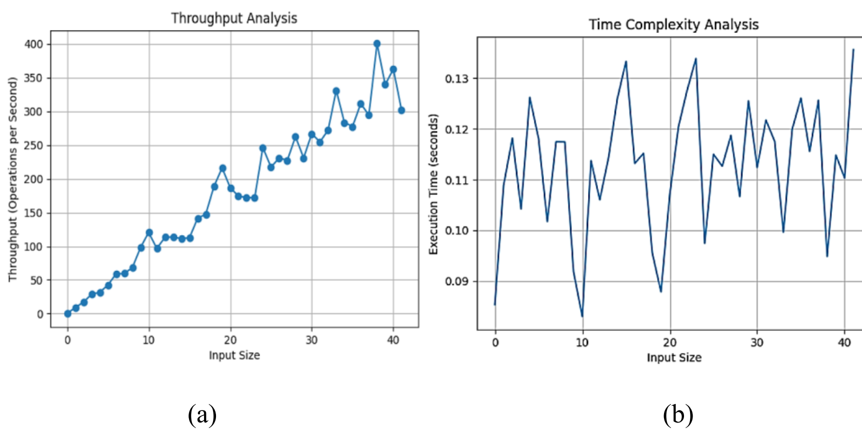


Fig. 8.CPU utilization (%) vs input size in proposed Re-Engineering framework

This was possible due to use of minimal cover attributes, structured CFT based execution and efficient slicing using Weiser's static backward slicing principles. The high-throughput outcome in the proposed framework ensures faster analysis, slicing and specification abstraction which is crucial during legacy code re-engineering. Unlike dynamic slicing which depends on runtime and may slow down due to trace generation, the proposed approach maintains high throughput. In contrast to the ML-based approaches which are mostly adaptive and often require batch processing and inference cycles reduces throughput in large-scale object oriented applications like Java.



(a) (b)

Fig. 9.(a) Throughput analysis and (b) Time complexity analysis vs input size in proposed Re-Engineering framework

The Fig.9(b) exhibits that despite increasing input size, the execution time stays mostly between 0.09–0.13 seconds, which reflects a nearly constant time complexity, or at worst sub-linear growth. This behavior is ideal for static slicing where computations rely on pre-constructed tables (CFT/DFT) rather than runtime tracing. It also shows that the early computation of minimal cover attributes and graph-based representation (CFT/DFT) minimizes the amount of real-time computation needed during slicing. This helps maintain consistent performance, regardless of structural complexity in the Java program.It has been observed that dynamic slicing often shows linear or even exponential time with increasing execution paths and runtime data, making it slower for large programs.ML-based slicing may also introduce latency due to model inference and data processing stages.

In contrast, static backward slicing using minimal cover attributes in proposed software re-engineering framework is both predictable and fast, ideal for CASE tools.

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 DFT-COV approach such as atm, grade, kcet, library, salary and stud grade.The original atm program comprises 240 lines of code, while the grade program consists of 321 lines; the remaining programs vary approximately between 200 to 500 lines each. It has been observed that in existing slicing approaches TRAD-COV the extracted slices consist of significantly increased lines of codes (LOC) for the same Java programs whereas in the proposed slicing techniques the LOC 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 Fig.10.

The experimental analysis of the proposed tool shows that, unlike existing slicing and re-engineering approaches, such as (Yadavally *et al.,*2024) [27], (Shahandashti *et al.,* 2024) [19] and (Mondal*et al.,* 2018) [31], 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.

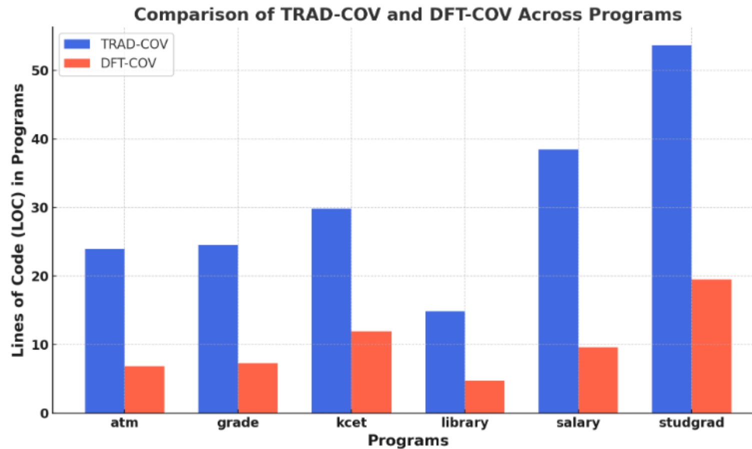
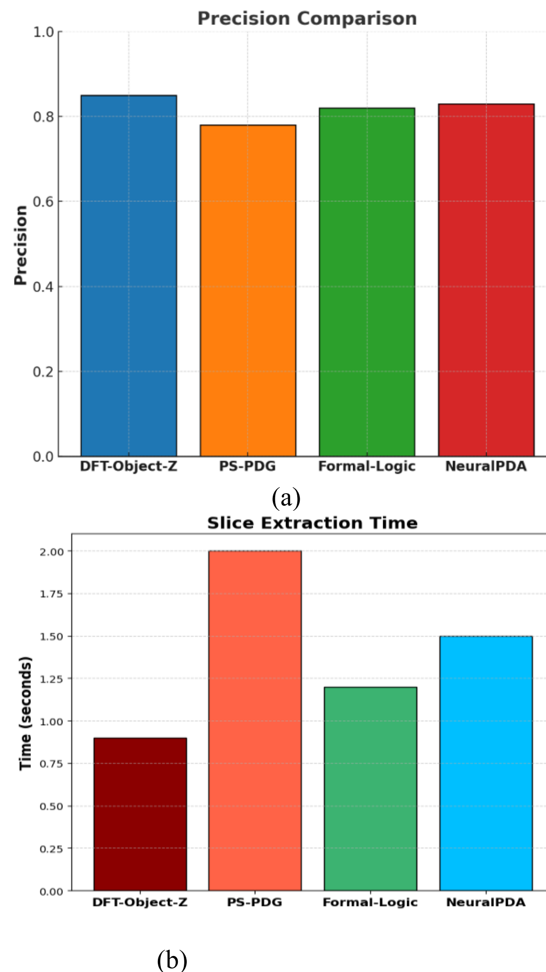


Fig. 10. Comparison of TRAD-COV and DFT-COV slicing approaches in the measure of LOC

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 LOC and yet offers computational efficiency with simplified strategies.

The experimental results as shown in Fig.11(a) exhibits that the DFT-Object-Z method of formal abstraction in proposed re-engineering framework offers precise slice extraction when compared with PS-PDG [28] and Formal-Logic methods [29] with variable sizes of Java programs. It also offers mathematically rigorous and unambiguous representation of program slices which ensures accurate specification derivation. On the other hand PS-PDG suffers from the problem of over-approximation due to difficulty in precisely distinguishing control dependencies from data dependencies in complex Java programs. The Formal-Logic-based models also produces potential logical inconsistencies for which it do not ensure much precise extraction of structured formal representation from program slices. On the other hand it has been observed that NeuralPDA [30] in many cases lacks explainability and struggles with unexpected Java program behaviours which also affects the slicing precision and effective formal representation of program slices.

Fig.11(b) on the other hand shows that DFT-Object-Z also offers significant reduction in slice extraction time due to its structured-state-transition model leading to faster verification and slice computation compared with the graph-based models which rely on exhaustive traversal of program dependence graphs. Formal-logic based abstraction models implements predicate resolution and symbolic execution paradigm for which it increases the computational overhead. The NeuralPDA depends on large-scale training which makes the slicing and formal abstraction process computationally expensive.



(b)

Fig. 11. Comparing simulation results with existing system (a) precision measure and (b) slice extraction time (seconds)

Conclusion

In this paper, we have introduced a novel software re-engineering framework designed to improve the effectiveness of requirements abstraction in Java-based systems. The proposed framework incorporates a static-backward slicing method that optimizes design and data flow abstraction, contributing to better CPU utilization by reducing computational overhead during slice extraction. By utilizing the DFT-Object-Z method for formal specification extraction, the framework not only improves the precision of requirements abstraction but also significantly enhances throughput by reducing slice extraction time by 55% compared to PS-PDG and 40% compared to NeuralPDA. Moreover, the time complexity of the slicing process has been optimized, resulting in faster execution without compromising on the accuracy or completeness of the extracted specifications. These improvements collectively offer a substantial contribution to efficiency in both resource utilization and processing speed, making the framework a significant advancement in software re-engineering for Java-based systems.Future research will further explore how this framework can facilitate seamless migration to modern technologies by extending its capabilities to support cloud-based and microservices architectures. The goal is to provide a comprehensive solution that not only improves software re-engineering processes but also enables smooth transitions to cutting-edge technologies, ensuring that systems remain scalable, maintainable, and future-proof.

Authors Contribution: All the authors (Aparna K S, and Dr. R.N. Kulkarni)contributed equally.
Funding: there has been no significant financial support for this work that could have influenced its outcome

Data Availability: The data availability statement is not applicable

Code Availability: Not applicable

Declarations

Conflict of Interest: We wish to confirm that no known conflicts of interest are associated with this publication.

Ethical Approval: Not applicable

Consent to Participate: Not applicable.

Consent for Publication: Not applicable

References

1. B. A. Stoica, S. K. Sahoo, J. R. Larus, and V. S. Adve, "Statistical program slicing: A hybrid slicing technique for analyzing deployed software," arXiv [cs.SE], 2021, doi: 10.48550/arXiv.2201.00060.
2. I. Postolski, V. Braberman, D. Garbervetsky, and S. Uchitel, "Dynamic slicing by on-demand re-execution," arXiv [cs.SE], 2022, doi: 10.48550/arXiv.2211.04683.
3. A.-J. Molnar and S. Motogna, "A study of maintainability in evolving open-source software," in Communications in Computer and Information Science, Cham: Springer International Publishing, 2021, pp. 261–282, doi: 10.48550/arXiv.2009.00959.
4. Z. Kovács and A. Vujic, "Open Source Prover in the Attic," arXiv [cs.PL], 2024, doi:10.48550/arXiv.2401.13702.
5. W. K. G. Assunção, L. Marchezan, A. Egyed, and R. Ramler, "Contemporary software modernization: Perspectives and challenges to deal with legacy systems," arXiv [cs.SE], 2024. [Online]. Available: http://arxiv.org/abs/2407.04017.
6. C. Miskell, R. Diaz, P. Ganeriwala, K. Slhoub, and F. Nembhard, "Automated framework to extract software requirements from source code," in Proceedings of the 2023 7th International Conference on Natural Language Processing and Information Retrieval, New York, NY, USA: ACM, 2023, doi:10.1145/3639233.363924.
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," Gov. Inf. Q., vol. 40, no. 1, p. 101784, 2023, doi: 10.1016/j.giq.2022.101784.
8. A. Bianchi, D. Caivano, V. Marengo, and G. Visaggio, "Iterative reengineering of legacy systems," IEEE Trans. Softw. Eng., 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," Innov. Syst. Softw. Eng., vol. 20, no. 4, pp. 559–584, 2024, doi: 10.1007/s11334-024-00568-z.
10. T. I. Mohottige, A. Polyvyanyy, R. Buyya, C. Fidge, and A. Barros, "Microservices-based software systems reengineering: State-of-the-art and future directions," arXiv [cs.SE], 2024. [Online]. Available: http://arxiv.org/abs/2407.13915
11. M. Weiser, "Program Slicing," IEEE Trans. Softw. Eng., 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 Trans. Softw. Eng., vol. 17, no. 8, pp. 751–761, 1991, doi: 10.1109/32.83912.
14. Y. Zhang, "SymPas: Symbolic Program Slicing," arXiv [cs.PL], 2019. [Online]. Available: http://arxiv.org/abs/1903.05333
15. M. N. Seghir, "Data-flow guided slicing," arXiv [cs.PL], 2018. doi: 10.48550/ARXIV.1808.01232.
16. Q. Zhang et al., "A systematic literature review on Large Language Models for automated Program Repair," arXiv [cs.SE], 2024, doi:10.48550/arXiv.2405.01466.
17. J. Singh, P. M. Khilar, and D. P. Mohapatra, "Dynamic slicing of distributed Aspect-Oriented Programs: A context-sensitive approach," Comput. Stand. 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,” SIGPLAN Not., vol. 25, no. 6, pp. 246–256, 1990, doi: 10.1145/93548.93576.
19. K. K. Shahandashti, M. M. Mohajer, A. B. Belle, S. Wang, and H. Hemmati, “Program slicing in the era of large language models,” arXiv [cs.SE], 2024. [Online]. Available: http://arxiv.org/abs/2409.12369.
20. C. Hammacher, “Design and implementation of an efficient dynamic slicer for java”. Bachelor’s Thesis, Saarland University, 2008.
21. K. Ahmed, M. Lis, and J. Rubin, “Slicer4J: a dynamic slicer for Java,” in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA: ACM, 2021, doi: 10.1145/3468264.3473123.
22. A. Soifer, D. Garbervetsky, V. Braberman, and S. Uchitel, “Focused dynamic slicing for large applications using an abstract memory-model,” arXiv [cs.SE], 2022. [Online]. Available: http://arxiv.org/abs/2211.04560
23. L. Vidziunas, D. Binkley, and L. Moonen, “The impact of program reduction on Automated Program Repair,” arXiv [cs.SE], 2024. [Online]. Available: http://arxiv.org/abs/2408.01134
24. Q. Stiévenart, D. Binkley, and C. De Roover, “An empirical evaluation of static, dynamic, and hybrid slicing of webassembly binaries,” 2024. doi: 10.2139/ssrn.4927726.
25. A. Yadavally, Y. Li, and T. N. Nguyen, “Predictive program slicing via execution knowledge-guided dynamic dependence learning,” Proc. ACM Softw. Eng., vol. 1, no. FSE, pp. 271–292, 2024, doi: 10.1145/3643739.
26. E. Soremekun, L. Kirschner, M. Böhme, and A. Zeller, “Locating faults with program slicing: an empirical analysis,” Empir. Softw. Eng., vol. 26, no. 3, 2021, doi: 10.1007/s10664-020-09931-7.
27. 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, no. OOPSLA1, pp. 83–109, 2024.
28. B. Homerding et al., "The Parallel Semantics Program Dependence Graph," arXiv preprint arXiv:2402.00986, 2024. [Online]. Available: https://arxiv.org/abs/2402.00986
29. A. Sen and V. K. Garg, "Formal Verification of Simulation Traces Using Computation Slicing," IEEE Transactions on Computers, vol. 56, no. 4, pp. 511–527, Apr. 2007, doi: 10.1109/TC.2007.1011.
30. A. Yadavally, T. N. Nguyen, W. Wang, and S. Wang, "(Partial) Program Dependence Learning," in Proc. 2023 IEEE/ACM 45th Int. Conf. Software Engineering (ICSE), 2023, pp. 2501–2513, doi: 10.1109/ICSE.2023.00123.
31. S. Mondal, “Context-sensitive slicing for Java applications,” Software Quality Journal, vol. 26, no. 4, pp. 799–820, 2018.