Formal Abstraction of Program Slices for Specification and Requirement Extraction Using Object-Z

K. S. Aparna
Department of CSE, Ballari Institute of Technology and Management (Affiliated to Visvesvaraya Technological University), India | Global Academy of Technology, Bangalore, India
aparna.vastrad@gmail.com (corresponding author)

R. N. Kulkarni
Department of CSE, Ballari Institute of Technology and Management, Ballari, India
rnkulkarni@bitm.edu.in

Received: 10 July 2025 | Revised: 12 August 2025 | Accepted: 22 August 2025

Licensed under a CC-BY 4.0 license | Copyright (c) by the authors | *DOI: https://doi.org/10.48084/etasr.13285*

ABSTRACT

Program slicing is widely used in software engineering for code comprehension, debugging, and system migration. It enables developers to extract and analyze relevant code segments based on predefined criteria, improving software maintenance. However, traditional slicing approaches often lack structured formal representations, making it difficult to derive precise software specifications, especially in complex Java-based systems. This paper introduces an Object-Z-based formal approach to program slicing, ensuring precise abstraction and systematic derivation of software specifications and requirements. The proposed model employs a backward slicing technique, where Object-Z schemas provide mathematically sound and verifiable representations of program slices. This approach effectively handles Java-specific challenges, including dynamic method invocations, encapsulated class structures, and inter-procedural dependencies. The proposed Object-Z-based formal representation framework ensures functional correctness while facilitating software migration and re-engineering. Experimental validation on large-scale Java programs demonstrates that this method improves precision, reduces computational complexity, and enhances the efficiency of requirement extraction compared to traditional formal techniques. The findings highlight that integrating formal methods with program slicing enables automated specification derivation, thereby improving software reliability, maintainability, and adaptability.

Keywords-program slicing; Object-Z representation; requirement extraction; Java systems; software verification; re-engineering

I. INTRODUCTION

Software systems have undergone rapid evolution over the past few decades, especially in object-oriented languages like Java and C++. However, continuous modifications to these systems occur, as modern applications require adaptation to changing business needs and technological advances [1]. This has resulted in unstructured and complex code, making maintenance and system migration to modern paradigms challenging, especially when their structure becomes complex due to frequent modifications [2]. Developers often struggle with system updates, particularly when original specifications and developers are unavailable. Migrating these applications to modern paradigms while preserving embedded business logic requires precise extraction of software specifications and requirements, especially in complex Java-based systems [3, 4].

Program slicing has gained popularity as a decomposition approach that aims to extract relevant segments of the program, eliminating unnecessary statements but ensuring preservation of program semantics. In Java applications, it plays a crucial role in analyzing functional dependencies and improving code comprehension [5]. Among program slicing approaches, static program slicing methods are highly praised as highly effective in analyzing the functional dependencies in Java programs. In addition, their ability to provide a complete view of dependencies without execution makes it highly relevant even when extracting modern software design specifications and requirements [6]. The formal representation of program slices is also essential for deriving precise software specifications and requirements, particularly in complex and evolving systems. This representation improves precision, correctness, and efficiency in obtaining software specifications, ensuring preservation of semantics, validation of requirements, and enabling automation to improve re-engineering scenarios in modern computing systems [7-10]. Several techniques, such as Program Dependence Graphs (PDG), System Dependence

Graphs (SDG), Predicate Logic-based Slicing, Formal Methods (Object-Z, VDM, B-Method), and many others, have been developed to represent program slices, ensure their correctness, and facilitate effective program analysis, verification, and requirements extraction [11, 12].

II. LITERATURE REVIEW

Program slicing and control/data flow abstraction have been extensively studied in the context of software verification, dependency analysis, and formal specification extraction.

A. Program Slicing in Formal Methods

In [13], an approach to slicing Object-Z specifications considered temporal logic formulas as slicing criteria. This technique identifies and eliminates specification segments that are irrelevant to the properties being verified, thereby simplifying the verification task. This method also constructs a dependence graph to trace and exploit both data and control dependencies within the specifications. The graph serves as a foundation to determine which specification component influences the slicing criterion. Incorporating the slicing approach significantly reduces the size of Object-Z specifications, leading to a more efficient verification process. Object-Z extends the Z notation with object-oriented features, making it suitable for modeling complex software systems. This slicing technique also aligns with object-oriented principles, making it more relevant for modern Java-based software verification. However, this method cannot effectively deal with contextual loss, limited generalization, and implementation complexity, restricting its adoption in a broader spectrum. It should be noticed that this work emphasized the importance of reducing specification complexity but was limited to formal specification languages rather than executable Java code.

In [14], predicate logic-based slicing was integrated with formal verification to analyze execution traces in distributed systems, using computation slicing to isolate relevant portions without exhaustive state exploration. This approach also demonstrates the role of slicing in the analysis of simulation outputs. However, although this approach improves verification efficiency, it struggles with imprecise predicates, unpredictable execution behaviors, and scalability in large or adaptive systems. It should be noted that both approaches were bound to specific modeling notations and did not address practical challenges in large-scale Java program slicing.

B. Program Dependence and Semantic Modeling

Parallel Semantics Program Dependence Graph (PS-PDG) [15] 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 adapt to the formal representation of program slices. Since PS-PDG mostly focuses on parallel program optimization, this full-fledged adaptiveness toward program slicing remains questionable. In addition, this semantics-based execution plan may introduce overhead when formally extracting slices. When mapping PS-PDG with Object-Z, formal abstraction may disregard inter-thread dependencies if not modelled explicitly. PS-PDG also suffers from increased computational complexity in analysing large-scale concurrent Java applications.

In [16], NeuralPDA was presented for program dependence analysis and formal specification extraction. NeuralPDA uses deep learning to predict program dependencies with high accuracy, which also helps to derive the formal representation of program slices. While traditional approaches rely on rule-based dependency graphs, NeuralPDA offers AI-based modeling, leading to more precise specification extraction from Java code. This approach relies on deep learning models with an 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. Thus, researchers still struggle to verify AI-generated slices mathematically. NeuralPDA also requires large annotated datasets, making the system resource-intensive for complex Java systems. Therefore, although these methods enhance dependency representation, they often incur high computational complexity and require specialized graph traversals, which can be impractical for large Java systems without structured abstraction.

C. Slicing in Java Programs

In [17], PDGs were examined for the extraction of slices while traversing the dependencies. This study showed that the Java System Dependence Graph (JSysDG) and dub-Statement Linear Dependence Graph (SSLDG) often struggle to produce complete slices when object variables are involved. This problem occurs due to inadequate handling of dependencies between partial definitions of objects. This method aimed to capture flow dependencies between object variables and their data members. However, this method suffers from computational complexities due to object-flow dependencies, which negatively impact the slicing process. Although this method improves accuracy, its performance in large-scale, complex Java systems with extensive object interactions requires rigorous empirical validation. Thus, although this approach is effective for small to medium-sized codebases, it still relies heavily on graph-based dependency representations, leading to performance bottlenecks in large-scale applications.

D. Combined Modeling and Transformation

In [18], Architecture Analysis and Design Language (AADL) was integrated with Object-Z to create a formal model called OZIA. This method exploits object-oriented principles for AADL modeling utilizing class inheritance and polymorphism to extract commonalities. It also further defines transformation rules from AADL-Object-Z to OZIA for formal verification. The strengths of this approach include the effective representation of formal specifications for comprehensive analysis and the support of reusing functional specifications while enhancing modeling efficiency. However, this method suffers from the complexity in integrating different modeling languages. It has also been observed that this method is not directly applicable to extracting control/data flow abstractions from restructured code.

E. Aspect-Oriented and Context-Based Requirement Modeling

In [19], a context-based aspect-oriented requirement engineering model was proposed to manage cross-cutting concerns in the specification of requirements. This study demonstrated how contextual modeling could help improve the modularity, traceability, and adaptability of requirements. However, while effective in high-level requirement structuring, this method lacked integration with formal slicing techniques for executable programs, limiting its applicability in precise control/data flow abstraction.

F. Identified Research Gap

Existing approaches, such as [13-19], highlight the significance of slicing and dependency analysis but often rely on complex PDGs or language-specific notations, limiting their scalability and cross-language applicability. Moreover, limited attention has been paid to integrating requirement-oriented abstraction into the slicing process in the form of a tool, particularly in static backward slicing of restructured Java programs to ensure precision, maintainability, and traceability in large systems. The identified research challenges highlight the limitations of traditional formal methods in software verification, program slicing, and specification derivation. These methods struggle with precise abstraction and structured representation of program slices, especially in large-scale Java systems. Graph-based, logic-based, and state-based models, such as Z, VDM, and D-Method, become computationally expensive as program complexity increases. Additionally, traditional slicing techniques fail to effectively distinguish between control-flow and data dependencies, making specification extraction more resource-intensive. They also lack efficiency in handling object-oriented features, such as dynamic method calls, inter-procedural dependencies, and aliasing effects, which impact slicing accuracy and performance. This work aimed to address these challenges by designing a computationally efficient approach for the formal abstraction of program slices that involves a set of computational operations and facilitates specification and requirement extraction using the Object-Z formal method. This approach:

- Introduces a Data Flow Table (DFT) structured in control flow order to replace complex graph embeddings.
- Integrates functional dependency minimization to reduce redundancy.
- Aligns slicing with requirement abstraction to bridge code-level dependencies with specification-level intent.

This work presents a formal and structured approach to extract precise specifications from Java code slices, focusing on combining the advantages of program slicing and formal specifications through the integration of Object-Z. The formal abstraction framework leverages the Object-Z formalism to enhance the backwards-slicing of Java programs, aiming to precisely extract software specifications by analyzing variable dependencies and program structure while reducing computational complexity. It also offers a mathematically structured, scalable solution to specification and requirement abstraction. The contributions of this study are: (i) It presents a cost-effective multitier formal abstraction framework that integrates an efficient backward slicing technique and aids in structured formal representation of Java program slices, (ii) unlike existing formal methods, the proposed approach does not rely on extensive runtime dependencies and improves scalability for large Java programs, (iii) DFT enhances the program abstraction accuracy from slices while reducing the computational complexity, (iv) offers precise extraction of software design specifications and requirements while preserving relevant program segments using the Object-Z formal method.

III. RESEARCH METHOD

The proposed formal abstraction framework was 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. 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 formal abstraction framework integrates program restructuring, design information extraction, and DFT computation to efficiently extract functional dependencies at an optimized cost. It also employs the Minimal Cover Algorithm (MCA) to refine functional dependencies by identifying minimal attributes. Additionally, an effective backwards-slicing approach ensures a structured formal representation using the Object-Z method, aiding in deriving precise software specifications and requirements. This formal abstraction framework initially derives the minimal cover attributes considering a minimal cover algorithm, as shown in Figure 1. The aim is to reduce the computational cost and effort with optimized operations using a set of computational steps toward exploring the effective functional dependencies.

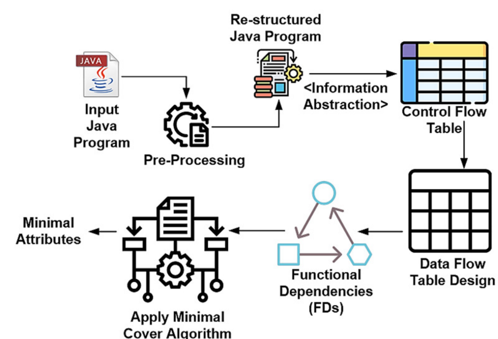

Fig. 1. Computation of minimal attributes in a formal abstraction framework.

A. Extraction of Minimal Cover Attributes in Formal Method

The proposed formal abstraction framework includes a pre-processing stage for Java program restructuring to enhance slicing efficiency. This stage eliminates irrelevant code, restructures program components, and improves control and data flow representation. The transformation process restructures the input Java program P_{java} into P'_{java} by appending relevant packages, classes, and interfaces. Further refinements include removing comments, blank lines, and unused variables, merging multiple declarations into single statements, and assigning unique line numbers to executable statements.

1) Java Program Restructuring and Preprocessing

Initially, the input Java program P_{java} is preprocessed and restructured into P'_{java} to improve control and data flow abstraction, along with slicing efficiency. This computing step basically removes comments, blank lines, and unused declarations, merges multi-variable statements, and ensures consistent control and data flow representation. Formally, the Java program is modeled as an ordered sequence of lines:

$$P_{java} = \{L_i\}_{i=1}^{n}, F: P_{java} \rightarrow P'_{java} \quad (1)$$

where L_i represents the i-th line in the source code and P'_{java} refers to the reconstructed Java program after appending necessary packages, classes, and interfaces.

$$P'_{java} = P_{java} \oplus S(P_U, C_U, I_U) \quad (2)$$

The transformation then eliminates all comment lines $\varepsilon\left(P'_{java}\right)$ and blank likes, producing P''_{java}, where the normalized code is suitable for precise control flow and data flow analysis. Here P''_{java} refers to a set or ordered list of all lines from the Java program after previous preprocessing steps. L_i represents a line of code from P''_{java} indexed by i from 1 to n. It also transforms multiple declaration statements into a single declaration statement using a function of transformation $\mathfrak{f}_{trans}(P''_{java})$. $s = \{s_i\}_{i=1}^{n}$ represents a set of individual declarations in P''_{java} . Then the function $\mathfrak{f}_{trans}(s \in P''_{java})$ processes $s = \{s_i\}_{i=1}^{n}$ and merges multiple declarations into a single statement considering the following mathematical expression:

$$s_{con} = Type \oplus \langle \{v_i\}_{i=1}^{\xi} \rangle \quad (3)$$

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

These optimizations ensure accurate, efficient, and scalable program slicing in further stages, making the transformed program more suitable for further analysis. For multiple data types, the transformation is applied independently for each type, considering the following mathematical expression:

$$\mathfrak{f}_{trans}(s) = \bigcup_{Type_i} \left(Type_i \oplus \sum_{j-1}^{\xi_i} v_{i,j} \right) \quad (4)$$

where $\bigcup(*)$ ensures that each type's variables are clustered separately and ξ_i is the number of variables of $Type_i$. The $\mathfrak{f}_{trans}(s)$ transformation normalizes code for formal slicing. It also ensures uniform representation of data types and enhances the static analysis.

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

$$L'_i = (l_{no.}(i) \rightarrow L_i) \quad \forall L_i \in e(P''_{java}) \quad (5)$$

where L_i refers to the executable statements from the preprocessed Java program. In addition, $l_{no.}(i)$ ensures that a line number is assigned to the i-th executable line, and $\rightarrow$ 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.

Restructuring plays a crucial role in the proposed formal abstraction framework, as program slicing via backward dependency tracing within this framework relies on accurate control and data flow representation. Unstructured or cluttered Java code might introduce false dependencies and missing references, leading to incorrect slices. Therefore, by standardizing the code format, removing irrelevant elements, and enforcing consistent syntax, restructuring ensures that subsequent data flow analysis and minimal cover extraction operate on a clean, symmetrically accurate program model, thereby improving both precision and computational efficiency. Here, clean and standardized code ensures precise extraction of program slices by capturing true dependencies and state changes that reflect the original program's intended behavior. This accuracy is essential while formalizing those slices in Object-Z schemas for specification and requirement extraction.

2) Control and Data Flow Abstraction

The work integrates control flow abstraction to enhance static backwards slicing for formal software specification extraction. This step ensures accurate slicing by tracing dependencies among conditional statements and function invocations in the restructured Java program P''_{java}. In this core computing step, control flow abstraction is performed on the restructured Java program P''_{java} to ensure precise and complete program slicing. Control flow abstraction helps capture execution paths dictated by conditional branches, loops, and function calls, which are essential for accurate backward slicing and specification derivation. Without this step, slicing results may omit dependent statements or misrepresent the program's intended behavior, especially when code structure varies. This method models P''_{java} as a sequence of statements in the form of $\{s_i\}_{i=1}^{k}$, categorized as:

- Conditional statements ($s_i \in c_s$)
- Computational statements ($s_i \in comp_s$)

- Function invocations ($s_i \in f(*)_{call}$)

Therefore, the program is represented as a sequence of statements categorized into conditional, computational, and function invocation statements. A structured Control Flow Information (CFI) table is introduced, mapping execution paths for precise slicing and data flow analysis. Unlike complex DFGs with high computational overhead $O(n)$ or $O(n^2)$, the proposed DFT offers structured, efficient processing with $O(1)$ or $O(\log(n))$ complexity, ensuring scalability for large Java programs. In this approach, the control flow information logic is further derived in the form of a 4-col structured tabular form, defined as follows:

$$CF_I\left(P''_{java}\right) = \{\langle s_{start}, s_{end}, Tr_1, Tr_2\rangle | s_{end} \in s_i, Tr_1, Tr_2 \in P''_{java}\} \quad (6)$$

The proposed work formulates the DFT in control flow order using $CF_I\left(P''_{java}\right)$ to track variable dependencies efficiently. Here, s_{start} marks entry, s_{end} refers to the control statement, Tr_1/Tr_2 represent true/false execution targets. In comparison with the conventional data flow graphs, which become complex and error-prone for large Java programs, the proposed tabular DFT represents fast lookup with high precision, cross-language compatibility, making it suitable for scalable software analysis. The proposed formal abstraction framework recognizes that the data variable interactions are inherently governed by the program's control flow.

Accordingly, the control flow information $CF_I\left(P''_{java}\right)$ computed in the previous step is further leveraged to construct the DFT (T_{DFT}) in the exact order of execution. In T_{DFT}, each row corresponds to a program statement and includes three attributes, line number, defined variables, and referenced variables, in the form:

$$\{\langle L'_i, v_d^i, v_r^i\rangle | i \in [1, n]\}$$

$$T_{DFT}(i) = \{\langle L'_i, v_d^i, v_r^i\rangle | i \in [1, n]\} \quad (7)$$

This structured approach enhances dependency tracking, maintains control flow order, and optimizes functional dependency abstraction, ultimately improving slicing and formal abstraction performance. From a requirement abstraction perspective, by preserving the control flow order in the DFT, the proposed formal method captures how program variables are introduced, transformed, and utilized across execution paths, which in turn reflects the program's underlying operational requirements. By explicitly mapping these relationships in control flow order, the proposed approach ensures that the derived specifications remain faithful to both the business logic and execution semantics of the original Java program.

3) Abstraction of Functional Dependencies from DFT

This approach identifies functional dependencies by analyzing variable relationships in a Java program. A functional dependency exists when a variable defined in one statement depends on variables from previous statements. This is mathematically represented as a relation extracted from the DFT:

$$f(L, v) = \{\langle L'_i, L'_j, v_c\rangle | v_c \in v_d^i \cap v_r^j, i > j\} \quad (8)$$

The abstraction process helps track dependencies across program statements, ensuring accurate slicing, optimization, and that all relevant variable interactions are preserved. By systematically capturing these dependencies, the framework supports both precise backward slicing and subsequent optimization stages, including the extraction of minimal cover attributes in the next step.

4) Finding Minimal Cover of the Functional Dependencies

The next stage refines this by extracting minimal cover attributes for improved efficiency. The proposed work minimizes functional dependencies $f(L, v)$ by eliminating redundancies, ensuring that only essential attributes are used for program slicing. This step reduces the functional dependency set of $f(L, v)$ to its minimal equivalent of $f \rightarrow f_{min}$, eliminating redundancies while retaining only essential dependencies for precise program slicing. A dependency of $\langle L'_j, L'_k, v_c\rangle$ is removed if it can be transitively derived from other dependencies or if its variables are irrelevant to the slicing goal. 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. Minimizing the dependency set is essential for requirement abstraction as it ensures that only the most significant variable relationships are preserved, those that directly link and influence the system's intended behavior. This approach in the proposed formal abstraction framework eliminates ambiguity introduced during maintenance, improves computational efficiency, and also enhances the traceability between program logic and formal specifications. Overall, this improves maintainability and efficiency in abstracting formal representation from Java program slices.

B. Structured Formal Representation Using Object-Z

To derive precise formal specifications in Object-Z, the proposed method first extracts computationally minimal and semantically relevant program slices. This is achieved by considering an enhanced static backward slicing algorithm that incorporates minimal cover attributes, ensuring that only statements that influence the target variables are retained. Such precision in slice extraction is crucial to eliminate irrelevant dependencies, thereby enabling a structured and concise mapping from code segments to Object-Z constructs.

1) Backward Slice Extraction Using Minimal Cover Attributes

This approach enhances Weiser's static-backward slicing approach to improve slice precision in DFT Coverage (DFT-COV) analysis. This improvement leverages minimal cover attributes to optimize the traversal and reduce irrelevant dependencies. The improved algorithm identifies and extracts only relevant program statements contributing to a slicing variable. It initializes the slice at the last occurrence of the variable, then recursively traverses dependencies using minimal cover attributes. By iteratively adding contributing

statements and eliminating irrelevant ones, the algorithm optimizes slicing efficiency. The final slices, derived from the restructured Java program, aid in abstracting software formal specifications, ensuring precise and efficient program analysis. The proposed slicing, called backwards slicing, searches and extracts only those program statements that contribute to the calculation of the slicing variable, which can be represented by:

$$S = \{L_n\} \cup \left\{L_i | \langle L'_i, L'_j, v_{slice}\rangle \in f_{min}, L'_j \in S\right\} \quad (9)$$

The proposed formal abstraction method further aids in deriving a structured formal representation of the program slice $S(v) \in P_{slice}$. The algorithm also enables optimized traversal considering:

$$S = S \cup \left\{L_i | \exists \langle L'_i, L'_j, v_{slice}\rangle \in f_{min}, L'_j \in S\right\} \quad (10)$$

where S represents the current slice, i.e., the set of lines influencing the value of the target attribute v_{slice}, and the minimal set of relevant data dependencies also explored via f_{min}. Here $\exists$ implies existence, and the process keeps expanding the slice S by including any L_i that defines the slicing variable v_{slice}. Here, the resulting $S(v) \in P_{slice}$ consists only of relevant statements, which are then extracted. This method generates multiple accurate backward slices for each minimal cover attribute, which forms the basis for a formal Object-Z specification abstraction.

2) *Representation of the Program Slice Using Formal Specification Language*

The extracted backward program slices are represented using Object-Z notation to ensure mathematical precision and structured representation. Object-Z provides a precise, mathematical, and unambiguous depiction of the program. In this method, only essential constructs—such as inputs, outputs, preconditions, and post-conditions—are utilized to extract software specifications. Inputs are denoted by the '?' symbol, outputs by the '!' symbol, and derived variables, computed from input variables, are represented using the delta symbol. These formal representations help abstract software specifications and user requirements effectively. The proposed formal abstraction framework offers a formal representation of program slices using Object-Z notations (Figure 2). The program slice is defined as $S_c \subseteq P$, where S_c consists of the statements relevant to the slicing criterion C. Here, S_c refers to the minimal subset of statements that affects $v \to l$, where v refers to variables and l refers to the location of P.

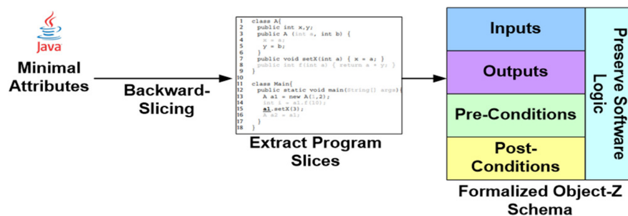

Fig. 2. Object-Z-based abstraction framework for program slices.

In the proposed formal method, the Object-Z notation for slice representation follows the standard slice schema of [19] and derives the proposed slice schema as S_δ as:

$$S_\delta \triangleq f(?:\mathbb{V}, !:\mathbb{V}, \Delta_s, P(\mathbb{V}), Q(\mathbb{V})) \quad (11)$$

where $\mathbb{V}$ represents a set of program variables, and Δ_s implies the state change 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 $Out(!:\mathbb{V}, \Delta_s) \Longrightarrow \exists S_\zeta$. S_ζ represents the correct slice transformation. This formal method helps extract the software requirements and specifications, considering preconditions that ensure valid inputs to the sliced function, postconditions that ensure expected outputs to the inputs, and also preserving the software logic while abstracting the specification of the program slice. The proposed formalized Object-Z schema, which follows the baseline principle of [20], can be used for complex software verification, migration, and maintenance.

IV. RESULTS AND DISCUSSION

The experimental assessment of the proposed formal abstraction framework considered Java programs with variable sizes, such as KCET600.java and ATMProjectout.java. The formal abstraction framework is realized as a tool for deriving specifications and requirements from program slices. The implementation considers PyCharm 2022.3 IDE, whereas the formal abstraction of program slices for specification and requirement extraction is implemented using Object-Z, with the proposed algorithms structured in a single Python script. The code analysis scripts were written using matplotlib, reportlab, graphviz, psutil, and tabulate libraries. The evaluation framework ran on a system with 12 GB of internal memory, a 64-bit Windows operating system, and an x64-based processor.

A. *Dataset Java Programs*

This study considered two different Java programs, KCET600.java and ATMProjectout.java, for experimental evaluation and validation. Both programs, with their details, are illustrated in Tables I and II. The experimental evaluation used metrics such as precision, slice extraction time (s), and reduction in the Lines of Code (LoC). Tables III and IV show the experimental results.

TABLE I. JAVA PROGRAM STATISTICS FOR KARNATAKA COUNSELLING SIMULATION (KCET600.JAVA)

Metric	Value
Lines of Code (LoC)	~469 (excluding comments % blanks)
Number of member functions	7 methods (including constructors and static helpers)
Total cyclomatic complexity	80

TABLE II. ATMPROJECTOUT.JAVA

Metric	Value
Lines of Code (LoC)	414
Number of member functions	36
Total cyclomatic complexity	48

TABLE III. KEY PERFORMANCE METRICS

Program	LoC (Orig)	LoC (After Slice)
ATMProjectout.java	414	280
KCET600.java	469	370

TABLE IV. KEY PERFORMANCE METRICS

Program	Slice Time (s)	Precision
ATMProjectout.java	~0.91	~0.81
KCET600.java	~1.05	~0.79

Precision measures how accurately the proposed formal abstraction method extracts the slices of relevant code portions, since Object-Z is further used for formal abstraction, thereby ensuring correctness and relevance in the extracted slices is crucial. Although Object-Z prioritizes correction and specification quality over speed, the proposed formal abstraction framework attempts to balance the trade-off between precision and computing speed to suit the requirements of real-time processing of modern large-sized Java-based systems. LoC implies better abstraction, which is a goal of formal specification. The proposed formal method emphasizes reducing code length while maintaining the completeness and correctness of the preserved system logic in the extracted specifications, which also positively influences the cost of the computation aspect.

This study also offers a comparative analysis to justify the results of the formalized Object-Z schema for structured formal representation. In this regard, it considers a set of related popular baselines, such as the graph-based PS-PDG model [15], the formal-logic-based model [14], and NeuralPDA [16], which aid in the formal representation of the program slices. PS-PDG basically relies on a semantics-based execution strategy, whereas the formal-logic-based model employs first-order logic to define the program dependencies and ensure formal correctness in slice extraction. On the other hand, NeuralPDA is a deep learning solution for automatic dependence detection in program slicing, aiding the formal representation of the program slices. The evaluation also examines the correctness of the extracted specifications, considering the proposed formal abstraction framework using the Object-Z notation. It also examines whether the extracted formal models match the expected behavior and logic of the system. The evaluation also examines how much necessary program logic is retained in the extracted specifications, and also determines the consistency factors that should comply with the Object-Z's syntactic and semantic rules. It should be noted that the Object-Z formalization in the proposed formal abstraction framework ensures the correctness and abstraction of the specifications.

The experimental results in Figure 3 show that the proposed DFT-Object-Z method of formal abstraction offers precise slice extraction compared with PS-PDG and formal-logic methods with variable sizes of Java programs. It also offers a mathematically rigorous and unambiguous representation of program slices, which ensures accurate specification derivation. Mathematically, precision can be defined using set-based evaluation metrics similar to information retrieval.

$$Precision = {|s_E|}/{|s_{Tot}|} \quad (11)$$

where s_E is the number of statements in the slice that are actually needed (i.e., relevant to the slicing criterion), and s_{Tot} is the total number of statements included in the extracted slice. 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 model also produces potential logical inconsistencies, which do not ensure much precise extraction of structured formal representation from program slices. On the other hand, it was observed that, in many cases, NeuralPDA lacks explainability and struggles with unexpected Java program behaviors, which also affects the slicing precision and effective formal representation of program slices.

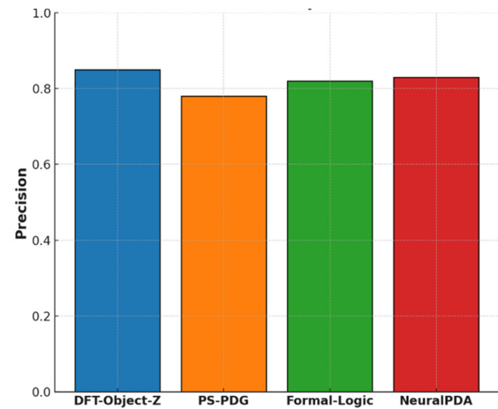

Fig. 3. Comparative analysis of precision.

Figure 4 shows that DFT-Object-Z also offers a 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 model, which relies on exhaustive traversal of program dependence graphs. Formal-logic-based abstraction implements predicate resolution and symbolic execution paradigms, increasing the computational overhead. NeuralPDA depends on large-scale training, which makes the slicing and formal abstraction process computationally expensive.

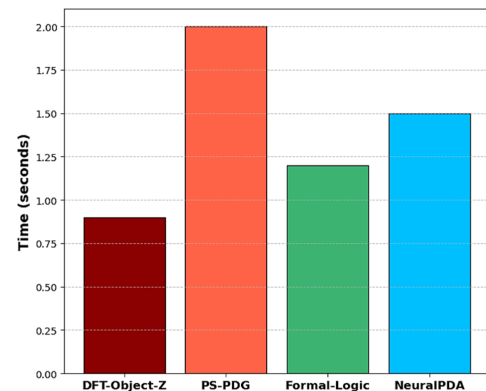

Fig. 4. Comparative analysis of slice extraction time (s).

DFT-Object-Z offers a significant reduction in slice extraction time due to its structured-state-transition model, leading to faster verification and slice computation compared to the graph-based models that rely on exhaustive traversal of program dependence graphs. Formal-logic-based abstraction models implement predicate resolution and symbolic execution

paradigms, increasing computational overhead. NeuralPDA depends on large-scale training, which makes the slicing and formal abstraction process computationally expensive.

Figure 5 shows the reduction in LoC through slice abstraction for structured formal representation and deriving the specifications and requirements. DFT-Object-Z emphasizes more on abstracting the essential components of Java programs through Object-Z representations, which include inputs, outputs, pre-conditions, and post-conditions, with optimized cost of operations that aid in reducing LoC while preserving the program logic in structured formal representations. However, PS-PDG models often generate large and complex dependency graphs for every Java program, leading to difficulties in manual verification of redundant statements in LoC. Similarly, Formal-logic-based approaches also explicitly define logical constraints for every program segment, which also increases the cost of computation for augmented LoC.

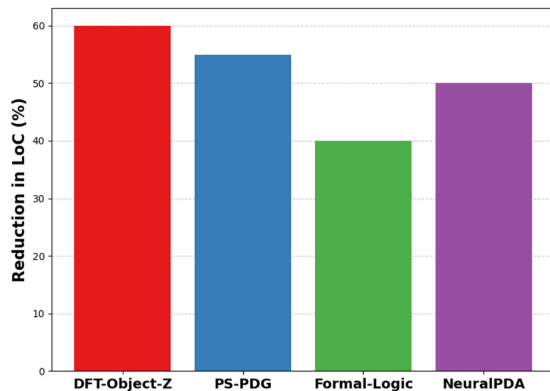

Fig. 5. Comparative analysis of reduction in LoC (%).

NeuralPDA produces slices using trained models but struggles with code generalization, which results in redundant statements in slices. This affects the performance of the formal abstraction of program slices and the verification of specifications and requirements. Overall, it was observed that DFT-Object-Z offers a significant balance between precision vs performance trade-off and also ensures semantic correctness in structured formal representations, which facilitates the derivation of automated specifications with minimal code overhead. Figure 6 shows the results of the program slice distribution for the ATMProjectout.java program, where it can be seen that by increasing the number of slices, there is a significant reduction in LoC using the proposed formal method. The experimental results exhibit the schema for ATMProjectout.java and KCET600.java in Figures 7 and 8.

TABLE V. COMPARISON OF KEY PERFORMANCE METRICS

Methods	Precision	Slice extraction time (s)	Reduction in LoC (%)
DFT-Object-Z	0.85	0.9	60
PS-PDG	0.78	2.0	55
Formal-Logic	0.82	1.2	40
NeuralPDA	0.83	1.5	50

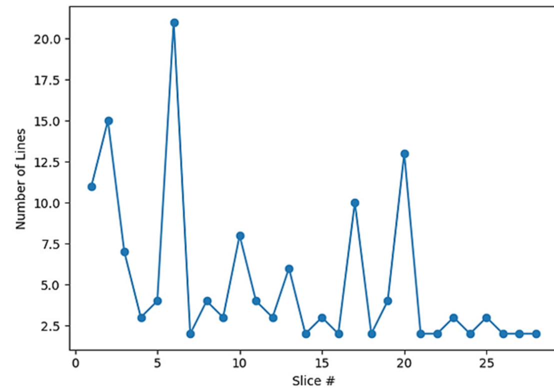

Fig. 6. Analysis of program slice size distribution.

```
ATM
------------------------------------------
accountNumber       : ℕ
pin                 : ℕ
availableBalance    : ℝ
totalBalance        : ℝ
userAuthenticated   : BOOL
userExited          : BOOL
------------------------------------------
availableBalance ≥ 0 ∧ totalBalance ≥ availableBalance
------------------------------------------
Init
------------------------------------------
accountNumber' = 0 ∧ pin' = 0 ∧
availableBalance' = 0 ∧ totalBalance' = 0 ∧
userAuthenticated' = FALSE ∧ userExited' = FALSE
------------------------------------------
ValidatePIN
Δ(ATM)
inputPIN? : ℕ
valid!    : BOOL
------------------------------------------
valid! = (inputPIN? = pin)
(valid! ⇒ userAuthenticated' = TRUE) ∧ (¬valid! ⇒ userAuthenticated' = FALSE)
------------------------------------------
Withdraw
Δ(ATM)
amount?  : ℝ
success! : BOOL
------------------------------------------
(amount? ≤ availableBalance) ⇒
  (availableBalance' = availableBalance - amount? ∧
   totalBalance' = totalBalance - amount? ∧ success! = TRUE)
(amount? > availableBalance) ⇒ success! = FALSE
------------------------------------------
CheckBalance
Ξ(ATM)
avail! : ℝ
total! : ℝ
------------------------------------------
avail! = availableBalance ∧ total! = totalBalance
```

Fig. 7. Object-Z representation schema for ATMProjectout.java.

The Object-Z schemas for both ATMProjectout.java and KCET600.java exhibit the slice specification extracted considering the proposed formal abstraction method. Unlike prior methods [13, 18], the proposed framework provides explicit formal schemas, supports scalability, and preserves semantic correctness during abstraction.

```
SeatAllocation
ΔSeatAllocationState
applicant? : Applicant
program?   : Program
quota?     : Quota
seatsAvailable : Quota ⇸ ℕ

pre  applicant? ∈ applicants ∧
     program? ∈ programs ∧
     seatsAvailable(quota?) > 0

post seatsAvailable' = seatsAvailable ⊕ { quota? ↦ seatsAvailable(quota?) - 1 }
```

Fig. 8. Object-Z representation schema for KCET600.java.

TABLE VI. COMPARISON WITH EXISTING APPROACHES

Feature	PS-PDG	NeuralPDA	Proposed DFT+MCA
Formal method	✕	✕	Object-Z
Minimal mover	✕	✕	✓
LoC reduction	Moderate	High	Higher
Complexity	$O(n^2)$	$O(n\log n)$	$O(\log n)$
Specification extraction	Partial	Learned	Mathematically precise

V. CONCLUSION

This work presents a unique approach to formal abstraction, presenting the DFT-Object-Z method that aims to extract precise formal abstraction of Java program slices to derive specifications and requirements. The results clearly show that DFT-Object-Z achieves higher precision and demonstrates an 8.97% improvement over PS-PDG, while it achieves marginal gains over other methods. It also reduces slice extraction time with an improvement of 55% over PS-PDG and 40% over NeuralPDA, making it a computationally efficient formal abstraction method. In addition, in terms of LoC, DFT-Object-Z outperforms other baseline approaches, such as Formal-logic and NeuralPDA, with 20-33% improvement. Overall, DFT-Object-Z offers structured and precise extraction of formal representations with optimized cost of operations, making it a strong solution for requirement derivation in complex Java programs. Future plans involve enhancing specification extraction by integrating Large Language Models (LLMs) to automate and improve the accuracy of deriving software requirements from program slices.

REFERENCES

[1] 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)*, Flagstaff, AZ, USA, Oct. 2024, pp. 337–349, https://doi.org/10.1109/ICSME58944.2024.00039.

[2] B. Wilber, T. D. Le, and A. Christi, "ReduJavator: A Tool to Simplify Developer-Written Java Unit Tests," in *2024 IEEE International Conference on Data and Software Engineering (ICoDSE)*, Gorontalo, Indonesia, Oct. 2024, pp. 199–204, https://doi.org/10.1109/ICoDSE63307.2024.10829914.

[3] 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, Mar. 2003, https://doi.org/10.1109/TSE.2003.1183932.

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

[5] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995.

[6] 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, Aug. 1991, https://doi.org/10.1109/32.83912.

[7] M. N. Seghir, "Data-Flow Guided Slicing." arXiv, 2018, https://doi.org/10.48550/ARXIV.1808.01232.

[8] Q. Zhang *et al.*, "A Systematic Literature Review on Large Language Models for Automated Program Repair." arXiv, 2024, https://doi.org/10.48550/ARXIV.2405.01466.

[9] J. Singh, P. M. Khilar, and D. P. Mohapatra, "Dynamic slicing of distributed Aspect-Oriented Programs: A context-sensitive approach," *Computer Standards & Interfaces*, vol. 52, pp. 71–84, May 2017, https://doi.org/10.1016/j.csi.2017.01.007.

[10] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *ACM SIGPLAN Notices*, vol. 25, no. 6, pp. 246–256, Jun. 1990, https://doi.org/10.1145/93548.93576.

[11] V. D'Silva, D. Kroening, and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008, https://doi.org/10.1109/TCAD.2008.923410.

[12] X. Chen *et al.*, "Deep learning-based software engineering: progress, challenges, and opportunities," *Science China Information Sciences*, vol. 68, no. 1, Jan. 2025, Art. no. 111102, https://doi.org/10.1007/s11432-023-4127-5.

[13] I. Brückner and H. Wehrheim, "Slicing Object-Z Specifications for Verification," in *ZB 2005: Formal Specification and Development in Z and B*, vol. 3455, H. Treharne, S. King, M. Henson, and S. Schneider, Eds. Springer Berlin Heidelberg, 2005, pp. 414–433.

[14] 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, https://doi.org/10.1109/TC.2007.1011.

[15] B. Homerding *et al.*, "The Parallel Semantics Program Dependence Graph." arXiv, Feb. 01, 2024, https://doi.org/10.48550/arXiv.2402.00986.

[16] A. Yadavally, T. N. Nguyen, W. Wang, and S. Wang, "(Partial) Program Dependence Learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, Melbourne, Australia, May 2023, pp. 2501–2513, https://doi.org/10.1109/ICSE48619.2023.00209.

[17] C. Galindo, S. Pérez, and J. Silva, "Program slicing of Java programs," *Journal of Logical and Algebraic Methods in Programming*, vol. 130, Jan. 2023, Art. no. 100826, https://doi.org/10.1016/j.jlamp.2022.100826.

[18] Z. Guo and Z. Cao, "Combined Formal Modeling and Model Transformation Based on AADL and Object-Z," *Journal of Software*, pp. 185–199, Nov. 2023, https://doi.org/10.17706/jsw.18.4.185-199.

[19] S. R. Idate, T. S. Rao, and D. J. Mali, "Context-Based Aspect-Oriented Requirement Engineering Model," *Engineering, Technology & Applied Science Research*, vol. 13, no. 2, pp. 10460–10465, Apr. 2023, https://doi.org/10.48084/etasr.5699.

[20] M. R. Laurence, "Characterizing minimal semantics-preserving slices of function-linear, free, liberal program schemas," *The Journal of Logic and Algebraic Programming*, vol. 72, no. 2, pp. 157–172, Jul. 2007, https://doi.org/10.1016/j.jlap.2007.02.008.