International Journal of Computer Information Systems and Industrial Management Applications.
ISSN 2150-7988 Volume 15(2023) pp. 43-58
© MIR Labs, www.mirlabs.net/ijcisim/index.html

Submitted: 5 Jan, 2023; Accepted: 20 May, 2023; Publish: 9 June, 2023

Extending Semantic based Techniques for Policy-based Slicing of Database Program

Anwesha Kashyap, Angshuman Jana

Indian Institute of Information Technology Guwahati, Guwahati, India
{anwesha.kashyap21, angshuman}@iiitg.ac.in

Abstract: **The innovation of the program slicing technique brings a revolution to address several issues (e.g. code understanding, debugging, maintenance, testing, etc.) in the more complex and large information systems. Over the past, many slicing techniques have been proposed, however, all these existing approaches did not consider the external database states. Moreover, the majority of the proposed slicing approaches are syntax-based and they do not consider the properties of variables and database attributes. Therefore, existing techniques are not directly applicable to data-intensive programs in information system scenarios and majority of them fails to compute precise slicing results. In this work, we propose a policy-based slicing framework for data-intensive programs. The policies are integrity constraints that are defined on data as per the business goal and kept in a backend database. We design our propose framework using the data dependency graph of data-intensive programs. We refine (by removing false alarms) the data dependency graph using semantics-based techniques, Condition-Action Rules and Hoare Logic. This refined graph yields precise slicing results w.r.t the policies of data-intensive programs.**

Keywords: Database Program, Program Slicing, Dependency Graph, Semantics Analysis.

I. Introduction

Static analysis is acknowledged as a key method for gathering data on how computer systems behave for all potential inputs without actually executing any code [1], [2]. Even though many non-trivial problems about program behaviour remain intractable in reality, decades of continuous and ongoing study in this field have made it possible to solve them [1], [3]. Majority of the static analysis approaches in the literature leverage dependency information (either explicitly or implicitly) between program statements and variables to address a variety of software engineering issues. Program slicing [4], information-flow analysis [5], optimization [6], code-understanding [7], code-reuse [8], taint analysis [9] are some examples of static analysis techniques.

The concept of program slicing [10] evolved as a helpful tool for extracting statements from programs that are relevant to a specific behaviour. Mark Weiser initially introduced the slicing approach in 1984 [11]. In this work, the author described a static program slicing that is an executable subset of program statements that maintains the behaviour of the original program at a certain program point for a subset of program variables for all program inputs. It contributes to the program's simplicity by focusing on the specified criteria. The slicing technique eliminates a section of the program, resulting in a subset of the original program with no effect on the semantics of the slice.

Data dependencies are unavoidable in the program when one piece of data needs another piece of data for execution. Therefore, one of the major components of program slicing is to recognise these dependencies and include them into the slices produced, maintaining the program's accurate syntactic representation. Numerous slicing approaches for the various programming languages have been developed based on this concept [12, 13, 14, 15]. Majority of the existing literature on slicing approaches exclusively mentions only imperative languages [4], and a very few important studies have been done on the slicing of database applications [12, 16, 14, 17]. Initially, Sivagurunathan et al. [15] dealt with the presence of external I/O states by putting pseudo variables inside the program to alert the slicer of the hidden I/O state. Tan and Ling [16] extended this concept to programs requiring database operations and used a similar approach as of [15]. Willmor et al. [12] suggested the Database-Oriented Program Dependence Graph (DOPDG), a variant of the Program Dependence Graph, which takes into account two new forms of data dependencies: (*i*) Program-Database Dependence between a program and a SQL query statement and (*ii*) Database-Database Dependence between two SQL statements. Cleve's work [14] is primarily concerned with the creation of System Dependence Graphs (SDG) for both the host and embedded languages.

There are several applications of slicing. Some of the applications are as follows:

(a) *Software maintenance.* Software maintenance is frequently followed by a re-engineering work, in

which a system is changed in order to enhance it. The concept of program slicing is applied to the software maintenance problem by expanding the notion of a program slice to a decomposition slice. A part of the software is extracted which is actually required. Using this technique also reduces the time and effort.

(b) *Error debugging.* When debugging programs, programmers mentally slice the code. The invention of an automated approach to aid in debugging operations by condensing the issue to a few lines or program statements was motivated by the need to support this mental debugging. That is, the program would eliminate any code that was unrelated to the collection of variables from which the error came and hence could not have been the cause of the error. This enables the debugger to focus on the lines or statements important to the error.

(c) *Code understanding.* The majority of the code in a program is unrelated to what you are interested in while studying it. Program slicing offers a practical method for removing "irrelevant" code. Using slicing we can extract a particular function or a set of statements that will help us to understand the flow of the program.

(d) *Testing.* Slicing facilitates program decomposition, which speeds up and improves the effectiveness of testing. Slicing is done according to certain slicing criteria. It allows for the identification of inter-related modules, which may subsequently be evaluated independently without disrupting the remaining program. Program slicing makes it possible to understand programs by breaking down them into slices, so that different testers may be given the duty of testing.

Therefore, slicing plays an important role for several software engineering activities. A common challenge in all of the aforementioned application contexts is to obtain more precise analysis results, that can be fulfilled if we are able to compute more precise slicing results. However, a major drawback of static analysis is that it gives false results which considerably reduces the development speed. The best strategy to decrease false-positives is to enable the analytic behaviour to be tuned to particular requirements.
Over the past, many slicing techniques have been proposed, however, all these existing approaches did not consider the external database states. Moreover, the majority of the proposed slicing approaches are syntax-based and they do not consider the properties of variables and database attributes. Observe that, the syntax based approaches account the dependencies based on the syntactic presence of variables in the definition of other variables. However, if we focus on the actual values in place of the variables then we experience that it is a false alarm [18, 19]. For instance, the value assigned to x does not depend on y in the statement $x = z + y - y$, although y occurs in the expression. Therefore, the syntactic approach, may fail to compute the optimal set of dependencies [18]. On the other hand, in the literature, we did not found much study of slicing on data-intensive programs in information system scenarios. Therefore, this encourages researchers to use semantics-based calculations to produce precise slicing results.
We intend to address the following two major research aims in this paper [1]:

- To develop a slicing framework for data-intensive programs in information system scenarios where it considers the external database states, and
- To obtain more precise slicing results.

To summarise, the following are our contributions to this paper:

1. Applying more suitable semantics based approach to refine the data dependency graph.
2. Defining policy and computing slicing w.r.t policies and experiencing the preciseness of the slicing results.
3. Experimental evaluation on a collection of open-source, database-driven JSP benchmark codes.

Our initial theoretical proposal [20] simply takes into account the condition action rule based approach. Although, it is an efficient semantics based analysis, however, it has some major drawbacks which needs to be considered. Therefore, we need more precise semantics based analysis to identify and remove the false dependencies present in the slicing results. In this paper, we extend our previous work [20] to make it more powerful by considering more suitable semantic based approach Hoare Logic to obtain optimised slicing results. We consider the syntax-based dependency graph and refine it (by removing more false dependencies) using Hoare Logic. Then we perform the slicing w.r.t the policy on this refined dependency graph that leads to more precise slicing result.
The structure of the paper is organised as follows: Section II shows the preliminaries and background of our work. In Section III, we discuss the literature's current state of the art. An illustration of our running example is provided in Section IV. In Section V we describe our proposed framework where we have shown the evolutions of dependency graph and then proposed semantic-based approaches for the refinement purpose. The experimental results are shown in Section VI. Section VII contains the discussion of the work. Finally, Section VIII concludes our work.

II. Preliminaries

In this section, we illustrate the fundamentals of program slicing through examples. Then we discuss policies in terms of integrity constraint of database applications w.r.t the goals of organisations.

[1] This work is a revised and extended version of [20].

A. Slicing

Program slicing is a process where statements from a larger source code that are connected to a certain behaviour are extracted [11]. The program's sections that are unrelated to those values are removed to create a slice. Typically, the point of interest is noted in the program by adding line numbers that correspond to each primitive statement and branch point. A slice is useful for program debugging, testing, software maintenance, measurement, program parallelization, program comprehension and other tasks because it is an independent program that is ensured to accurately reflect the original program in the context of the designated subset of behaviour. These applications of program slicing results from two characteristics that they possess. We simply need to understand a portion of a program while changing it, not the entire thing. This characteristic leads to applications such as maintenance, debugging, and testing. Rather than grasping the entire program immediately, we might first comprehend specific pieces, then comprehend the association between the sections. This characteristic enables measurement, program parallelization, program understanding, and so on. Slicing is computed using either backward or forward slicing algorithm w.r.t the slicing criteria. Let us discuss the slicing criteria, backward and forward slicing as follows [11]:

Slicing Criterion [11]: A program P's slicing criterion is a tuple with the values <i, V>, where i is a statement and V is a subset of P's variables. We will see the illustration of slicing in example 1.

Backward Slicing [11]: Backward slicing is a technique used in program slicing, a method for reducing the size of a program by removing parts of the code that are not relevant to a particular analysis or manipulation. Backward slicing starts from a specified set of program points (e.g. a set of statements) and traces backwards through the program, identifying all program points that are needed to compute the values of the specified points. The resulting slice consists of all the program points that were traversed during the backward slicing process. Backward slicing is used to answer questions such as: "What parts of the code could have affected the value of a particular variable?" or "What is the minimum amount of code necessary to understand a particular error or bug?" By removing unneeded parts of the code, backward slicing makes it easier to understand and analyze the behavior of a program.

Forward Slicing [21]: Forward slicing is a program slicing technique that concentrates on the effect that a change in a program's input or initial conditions will have on the values of its outputs. It involves creating a slice of the program that includes all statements that might affect the value of the program's outputs, and removing all statements that are not relevant to the calculation of those outputs. This allows a programmer to isolate the part of the code that is responsible for producing a particular result, making it easier to understand and modify the program. Forward slicing involves starting from a particular output, and tracing the flow of data backwards through the program to identify the statements that contribute to the calculation of that output. The resulting slice consists of the statements that must be executed in order to produce the specified output, and can be used to simplify the program or to isolate the cause of a bug. Forward slicing is typically performed using data flow analysis, which tracks the flow of data through the program to determine which statements are necessary to calculate a particular result.

Example 1 *Consider a program P and the slices of P w.r.t the slicing criteria s1 = <9, sum> and s2=<10, mul>.*

1. int i, sum, mul;	1. int i, sum;	1. int i, mul;
2. i = 1;	2. i = 1;	2. i=1;
3. sum = 0, mul =0;	3. sum = 0;	3. mul = 1;
4. while (i≤10)	4. while(i≤10)	4. while(i≤10)
5. sum = sum + i;	5. sum = sum + i;	
6. mul = mul * i;		6. mul = mul * i;
7. i++;	7. i++;	7. i++;
8. printf ("%", i);		
9. printf ("%", sum);	9. printf ("%", sum);	
10.printf ("%", mul)		10. printf ("%", mul)

Figure. 1: Program P | Slicing criteria s1=<9, sum> | Slicing Criteria s2= <10, mul>

The slices in the preceding example are computed with the original program P in Fig. 1. In this scenario, we look at the variables *sum and mul* as well as program points 9 and 10. As a result, only those statements from the original program are taken out that have a direct influence on those variables. The slice section of P with respect to the slicing criteria s1=<9, sum> and s2= <10, mul> is illustrated in Fig. 1.
As we can see, slicing aids in providing statements that are directly or indirectly engaged in the computation, reducing the cost of going through each line of code in the original program to discover the relevant lines to edit. As a result, with large software with thousand lines of code, it is extremely difficult to determine the origin of a defect or which line's update is necessary. We may lessen the effort required to handle this issue by using program slicing, which provides us with a portion of the program to work with, without interfering with the original program in any way. It saves time and money while also improving program understanding.

B. Integrity Constraints

The integrity constraint is represented in terms of policies and is imposed in an information system as per the business goal. Policies are collection of rules or predefined rules that maintains the quality and consistency of data in a database. Consumer needs might shift from time to time, causing policies to change. Basically, policies are business rules or integrity constraints to fulfill the business goal [22]. For instance, certain banks in a financial system have a policy stating that the minimum amount of a specific account must be more than or equal to $50. Consider another example, in the telecom industry if they want to incorporate a change in a sin-

gle plan, it will be easier to extract the part of the code which will be affected by the change. It will be easier this way since it will reduce the time to find that single piece of code where change must be done, from the large code the original program included. Hence, changing of policies can be incorporated seamlessly since it will be easier and efficient without hampering the original program in any way [23]. In this work, the policy is considered as an slicing criteria and w.r.t this criteria, we extract the set of semantically equivalent statement to maintain the policies.

Example 2 *Consider the program Q in Fig. 2 which is an example of a bank account that calculates the simple interests of customer accounts. Initially, the rate of interest was 7%. The bank decided to increase its interest rate from 7% to 7.5%. After the new policy has been employed the variable which is responsible for the calculation of the interest rate will be changed. Hence, statement 3 and statement 5 will have to be modified since the rate of interest has been increased. Therefore the important lines of the code should be taken care of which can be computed using slicing if we consider this type of constraints as a policy.*

```
1. float interest(float p, float r, float t)
2. {
3. r = 7;
4. float si;
5. si = (p * r * t)/100;
6. return si;
7. }
```

Figure. 2: Program Q

III. Related Works

The concept of program slicing which was first put out by Mark Weiser [11] in 1984, has evolved over time. The produced slices consist of statements that have the potential to change the values of the slicing criteria variable. Since then, researchers have utilised this idea to compute slices of several programming paradigms [24], [25], [21], [26]. Despite the fact that [25], [21] focuses mostly on the application and advancements of program slicing, [27] attempts to discover an implementation for an interactive software development environment.

Program slicing applications have now evolved into powerful software tools that are used at various phases in the life cycle of software development, such as program understanding, automated computation, code verification, testing and software maintenance, quality assurance, program integration, cohesion etc. [28], [25], [29]. Several methods have been proposed for slicing, including dynamic slicing [11], [25], amorphous slicing [25], [30], static slicing [11], [25], quasi-static slicing [25]. The Program Dependence Graph (PDG) may be used to approach static slicing [27], [6] by including both data and control dependencies of the program. A PDG has edges that reflect the control and data dependencies between statements and vertices that represent program statements. The slice with relation to the slicing criterion is computed from a vertex in the PDG that is originally associated with the slicing criterion. To enumerate the slices by compiling statements and predicates, the control flow graph (CFG) or PDG of the program is traversed backward beginning with the programmer-specified slicing criterion. Another technique given by Bergeretti and Carre [31] is to define slices in terms of information-flow relations obtained from a programme in a syntax-directed manner. The results of Denning and Denning [32] in secure information flow are formalised in this connection. Numerous PDG representations have also been proposed throughout the years [28], [25], depending on the desired usage. During the past few decades, several program slicing methodologies based on PDG representation or modifications, such as Dynamic Dependence Graph, System Dependence Graph etc. have been created [25], [28]. In general, data dependence is typically used to explain the essential data flow of the program, whereas control dependency is derived from the original control graph and illustrates just the program's critical control interactions. PDG-based slicing, on the other hand, is rather restricted since it must be computed or implemented in relation to the slicing criteria established at the given program point. Because the concepts of program slicing and data provenance are similar, it may be utilised to exploit the usefulness to transfer ideas, tools, and approaches for future study. Danicic et al. [33] proposed a parallel technique for calculating backward, static slices. The CFG is transformed into a network of parallel processes to do this. Messages naming the necessary sets of variables are sent and received by each process. Specification-based approaches were proposed by Chung, Lee et al. [34] to make program slicing easier. By deleting assertions that are not pertinent to the specification for the slice, one may use the specification to produce slices that are more exact. Their approach is pre/post condition-based. This specification-based slicing is useful for extracting reusable components, reconstructing programs, and other tasks.

In [6], authors have used PDG to present and show efficient and powerful program transition for compiler optimization. The importance of it is that it allows the programmer for potential parallelism. The authors expect PDG as the basis for process partitioning for multiprocessors. The authors of [35] proposed an application of program slicing in context to data provenance as well. Provenance can be defined as the understanding and troubleshooting of database queries by giving us an explanation of the results through its input. Program slicing gives us a concise explanation of the error in the program or any aberrated behaviour present in the program in the form of slice which shows only the part

relevant to the error. Data provenance has a compelling analogy to program slicing since it also tries to explain part of the result of a query using the relevant part of the input database. Willmor et al. [12] introduced a PDG version termed as Database Oriented Program Dependency Graph (DOPDG) which is in accordance to the traditional PDG, however, it has two extra dependencies which is discussed in the further section. However, to obtain finer slices, [12] addressed the issue of false alarm which is a drawback in case of syntax-based DOPDGs. There needed to be semantics based analysis to overcome such false alarms.
Evaluations of various slicing methods have also been made as more and more varieties of slicing methods have been developed. These assessments [36, 37, 38, 39, 21, 40, 41] largely focused on the characteristics of slicing, including its effectiveness, the size of the slices that arise, applications, and how slicing interacts with other types of source code analysis. Another new approach called Symbolic Program Slicing (SymPas) has been developed in [42] where the authors perform dataflow analysis on LLVM and offer a lighter-weight alternative to traditional slicing methods. This paper examines the SymPas approach and its potential benefits. In [17], authors proposed a new approach called srcClone that can detect both syntactic and semantic code clones using a slice-based scalable method. Their approach determines code segment similarity by analyzing the similarity of the corresponding program slices. They utilize a lightweight and readily available program slicer, which enables their clone detection approach to scale more efficiently.

IV. Running Example

Let us take a database program fragment "`Prog`" shown in Fig. 3. The program implements several functionality of an Employee Management System (EMS). And it performs various kinds of operation.
The database code snippet "`Prog`" raises employees' salaries depending on their years of employment with the company. The percentage increase in the salary is stored in two variables 'x' and 'y' which are depicted in statements 2 and 3 respectively. The increase in the salary is 5% for all the employees serving for a minimum of 3 years and a maximum of 45 years, which is shown in statement 8, and 15% for employees serving for more than 20 years which is shown in statement 5. Finally, the average of the salary is computed for all the employees in that organisation. Note that we illustrate our propose framework using this example in the subsequent sections.

V. Proposed Framework

In this section, we design our framework using semantic based methods which improves the syntactic Database Dependency Graph (DDG) to obtain more precise and accurate slicing results. Let us recall the syntax-based DDG construction from [12, 43].

A. Evolutions of Dependency Graph

At first, we discuss the evolution of dependency graph. In general, a database dependency is a constraint that governs how attributes relate to one another. A dependency graph is a program intermediate form that reveals both data and control interdependence between program statements [6]. Nodes represent program statements in dependency graphs, whereas edges indicates data and control dependencies. Program Dependence Graph (PDG) is playing crucial roles in a wide range of software-engineering activities, e.g., program slicing, code-reuse, language-based information flow security analysis [44] code-understanding. The graphical depiction of control flow or computation during the execution of programs or applications is called a control flow graph (CFG). Basically a control flow graph is a visual representation of a program's control flow, which shows the sequence of operations executed in the program and the conditions that control the flow of execution. Data dependencies are typically represented as directed edges between operations in the control flow graph, with the direction of the edge indicating the flow of data from one operation to another.

1. *Directed Graph:* A pair (S, E) that consists of a set of nodes S and a set of edges $(E \subseteq S \times S)$ is referred to as a directed graph. A path P connecting node n_1 and n_j is a non-empty sequence of nodes $\langle n_1, \ldots, n_j \rangle \in S^+$ where $(n_i, n_{i+1} \in E \; \forall \; i \in \{1, \ldots, j\text{-}1\}$.
2. *Control flow graph (CFG):* A CFG with defined(D) and used(U) sets is a tuple $(S, E, \mathrm{D}, \mathrm{U})$ in which (S, E) is a directed graph, S has two distinct nodes which is start and stop, and defined, used: $S \rightarrow \wp(\mathbb{V})$ are functions that return the node's defined and used variables, respectively.
3. *Data Dependencies:* Data dependencies refer to the relationships between operations in a program where the value of one operation is used as input to another operation. These dependencies describe the flow of data through the program and affect the order in which the operations must be executed. Knowing the data dependencies in a program is important for analyzing and optimizing its performance, as well as for parallelizing its execution for improved efficiency on multi-core and multi-processor systems.
4. *Control Dependencies:* In the context of a control flow graph, control dependencies refer to the relationships between operations (nodes in the graph) and the control flow statements (if-then-else, while loop, etc.) that determine the order in which the operations are executed. The control dependencies specify which operations must be executed before a particular control flow statement is executed, and which operations are executed as a result of that control flow statement. This information is used to determine the order of execution of operations in the graph and ensure the correct behavior of the program.

```
1. START
2. float x = 0.05;
3. float y = 0.15;
4. Statement con = DriverManager.getConnection("jdbc sql: . . . ", "scot", "tig").createStatement();
5. con.executeQuery("UPDATE Emp SET sal = sal + y * sal WHERE yrs_of_employment ⩾ 20 ");
6. ResultSet rs1=con.executeQuery("SELECT AVG(sal) FROM Emp WHERE yrs_of_employment ⩾ 20 ");
7. ResultSet rs1=con.executeQuery("SELECT AVG(sal) FROM Emp WHERE yrs_of_employment < 20 ");
8. con.executeQuery("UPDATE Emp SET sal = sal + x * sal WHERE yrs_of_employment BETWEEN 3 AND 45 ");
9. ResultSet rs1=con.executeQuery("SELECT AVG(sal) FROM Emp WHERE yrs_of_employment BETWEEN 3 AND 45 ");
10. END
```

Figure. 3: DB Code "Prog"

5. *Program Dependency Graph:* A program dependency graph (PDG) is a directed graph that represents the relationships between elements in a program, such as variables, statements, and functions [45]. It shows how the values of one element depend on the values of other elements, and is used to analyze and understand the behavior of a program. Each node in a PDG represents a program element, such as a variable or statement, and the edges represent the relationships between elements. For example, an edge from one statement to another might indicate that the first statement has an impact on how the second statement is carried out. PDGs can be used for a variety of purposes, such as analyzing the impact of changes to a program, identifying potential bugs, or optimizing the performance of a program.

   The construction of a PDG typically involves data flow analysis, which tracks the flow of data through the program to determine the relationships between elements. The PDG is updated as the program is executed, and can be used to create program slices, which are smaller parts of the program that represent the minimum set of statements necessary to produce a particular output.

Fig. 4 shows a simple program and its corresponding PDG where control dependency is shown by solid lines, whereas data dependence is represented by dashed lines.

```
1. x = v();
2. while(q>0) {
3. y=u();
4. if(y>0)
5. z=x;
6. else
7. w=z;
8. }
9. m=w;
```

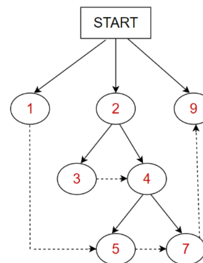

Figure. 4: A small program (left side) and its corresponding PDG (right side)

However, in [12] the notion of Data Dependency Graph (DDG) first proposed by the authors in the context of database systems that contain query language. The DDG is generated of this database code in place of the conventional PDG and is equivalent to the former with two additional dependencies:

Definition 1 (Program-Database (PD) dependence) [43]. If the following three conditions are met, a database statement D is PD dependent on a program statement I for a variable x (denoted $I \xrightarrow{x} D$): (i) I determines x, (ii) x is used by D, and (iii) x is not redefined by I and D.

Definition 2 (Database-Database(DD) dependence) [43]. Consider D.INS, D.SEL, D.DEL and D.UPD denotes database operations which are insert, select, delete and update respectively by statement D. For an attribute 'a', a database statement D1 is reliant on statement D2 in the same database (denoted $D1 \xrightarrow{a} D2$) if the following hold: (i) D1.SEL ∩(D2.INS ∪ D2.UPD ∪ D2.DEL) = ∅, and (ii) The impact of D2 cannot be undone by reversing the execution on the route p between D2 and D1 (exclusive).

Observe the running example "Prog". Figure 5 demonstrates the syntax-based Database Dependency Graph (DDG) of "Prog". The dependencies between the imperative and control statements are determined in the same way as traditional PDG. To compute the PD-dependency and DD- dependency we have to compute the defined (D) and used (U) variable of the statements:

D(2) = {x} D(3) = {y} D(4) = {sal, yrs_of_employment}
D(5) = {sal} U(5) = {y, yrs_of_employment}
U(6) = {sal, yrs_of_employment}
U(7) = {sal, yrs_of_employment}
U(8) = {x, yrs_of_employment}
U(9) = {sal, yrs_of_employment}

Based on the above information we can compute the PD- and DD- dependences. For Example: edges $4 \xrightarrow{yrs_of_employment} 5$, $5 \xrightarrow{sal} 6$ etc. represents the DD-dependency. Similarly, edges $2 \xrightarrow{x} 8$, $3 \xrightarrow{y} 5$ represent the PD-dependency. The syntax-based DDG is illustrated in Fig. 5 where the red edges represent the false dependencies present in the program.

Limitations: In the code snippet "Prog" in running example (Section IV), even though statement 5 and statement 7 are syntactically reliant i.e dependent on one another, however if we focus on value there is no dependency since statement 7 does not use the database-part declared in line 5. Similarly, we can observe the same false dependency between defined attribute of state-

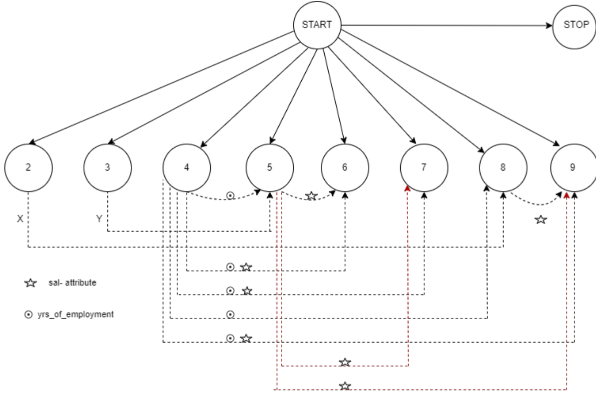

Figure. 5: Syntax-Based DDG of Program "`Prog`"

ment 5 and used attribute of statement 9. Since false dependencies exists, a more precise semantic-based approach is required to identify and eliminate those.

B. Semantics Refinement: Condition-Action Rules

To address the issue of spurious dependency in the syntax-based database dependency graph, we offer a semantic-based analysis based on the Condition Action rule-based method proposed by Elena and Jennifer in [46]. The suggested Condition Action rule-based method which is introduced in [46] is for expert database systems and is expressed using relational algebraic expressions. First, we recall from [47] the formulation of the semantics of database query languages.

Formal Syntax. Table 1 depicts the syntactic sets and the abstract syntax of database statements in Backus-Naur form. The syntax of SQL-embedded database programs from [44] is summarized here. The imperative aspect of the language also includes operations like assignment, iteration, skip, and conditional. The variables used in the language can be divided into two categories: application variables ($\mathbb{V}_a$) and database variables ($\mathbb{V}_d$). The GROUP BY and ORDER BY clauses in the SQL are represented by the functions g($\vec{e}$) and f($\vec{e}$), respectively, where $\vec{e}$ is a sequence of arithmetic expressions. The aggregate functions in the SELECT query, namely AVG, MAX, MIN, SUM, and COUNT, are represented by the symbol s. The ordered sequence of aggregate functions is denoted by $\vec{h}(\vec{x})$ where $\vec{x}$ is a sequence of arguments.
The regulations are written in the form $E_\phi \rightarrow E_A$, where E_A denotes the action part and E_ϕ denotes a condition of the statement. When it comes to database codes, each database statement q can be expressed in two parts: (i) action part E_A (ii) condition part E_ϕ. It may be formalised as q = $< E_A, E_\phi >$. E_A includes four database statements namely, *SELECT, INSERT, UPDATE, DELETE* which can be denoted by E_{Asel}, E_{Ains}, E_{Aupd}, E_{Adel}. The condition in the *WHERE* clause falls under E_ϕ which is a first order logic formula. The algorithm analyses condition and action and predicts how one statement's action might impact the condition of the other statement.

Additionally, the *attribute extension* operator lets us add a new attribute to a relational expression E. This method is utilised for aggregate functions as well as modification actions. ϵ is an unary operator which is applied to a relational expression E producing a small result with schema{E} ∪ {x}. ϵ is expressed as ϵ[x=expr] where, *expr* is an expression that is evaluated over each tuple t of E. Let us consider a *SELECT* statement, "SELECT salary FROM Emp WHERE salary≤ 10000;". The E_ϕ and E_{Asel} part of the above statement is:

$$E_\phi = \pi_{salary}(\sigma_{salary \leq 10000} \text{ Emp}) \quad \text{and} \quad E_{Asel} = null$$

Secondly, an *INSERT* statement "INSERT INTO Emp(EmpID, FName, LName, Dept) VALUES('1001', 'Rakesh', 'Sarma', 'Accounts');" is being considered. The E_ϕ and E_{Ains} part of the above statement is:

$$E_\phi = null \quad \text{and} \quad E_{Ains} = <\text{'1001','Rakesh','Sarma','Accounts'}>$$

Thirdly, let us take an *UPDATE* statement, "UPDATE Emp SET salary = salary + salary * bonus WHERE salary ≥ 50000";

$$E_\phi = \pi_{salary}(\sigma_{salary \geq 50000} \text{ Emp}) \quad \text{and}$$

$$E_{Aupd} = \epsilon[\text{salary=salary+salary*bonus}](\sigma_{salary \geq 50000} \text{ Emp})$$

Lastly, a *DELETE* statement "DELETE FROM Emp WHERE age> 60;" is being considered. The E_ϕ and E_{Adel} part of the above statement is:

$$E_\phi = \pi_{age}(\sigma_{age>60} \text{ Emp}) \quad \text{and} \quad E_{Adel} = \pi_{age}(\sigma_{age>60} \text{ Emp})$$

The π and σ used above are relational algebra operators for attribute projection and selection.
Now, we use a Condition-Action rule-based approach to express the semantics of the Running Example code snippet, as shown below:

$$E^5_\phi = \pi_{sal}(\sigma_{yrs_of_employment \geq 20} \text{ Employee})$$

$$EA^5_{upd} = \epsilon[\text{sal} = \text{sal} + \text{y} * \text{sal}](\sigma_{yrs_of_employment \geq 20} \text{ Employee})$$

$$E^6_\phi = \pi_{sal}(\sigma_{yrs_of_employment_{\geq 20}} \text{ Employee}) \quad EA^6_{sel} = \text{null}$$

$$E^7_\phi = \epsilon_{sal}(\sigma_{yrs_of_employment_{\leq 20}} \text{ Employee}) \qquad EA^7_{sel} = \text{null}$$

$$E^8_\phi = \pi_{sal}(\sigma_{yrs_of_employment \geq 3 \wedge yrs_of_employment_{\leq} 45} \text{ Employee})$$

$$EA^8_{upd} = \epsilon[\text{sal=sal+x*sal}](\sigma_{yrs_of_employment \geq 3 \wedge yrs_of_employment_{\leq} 45} \text{ Employee})$$

$$E^9_\phi = \pi_{sal}(\sigma_{yrs_of_employment \geq 3 \wedge yrs_of_employment_{\leq} 45} \text{Employee})$$

$$EA^9_{sel} = \text{null}$$

After computing the condition and action part of the database statements of the code snippet "`Prog`" we now use that information to compute the dependencies be-

Syntactic Sets	**Abstract Syntax**
$n : \mathbb{Z}$ (Integer)	$e ::= n \mid k \mid v_a \mid v_d \mid op_u\ e \mid e_1\ op_b\ e_2$, where $op_u \in \{+,-\}$ and $op_b \in \{+,-,*,/,\dots\dots\}$
$k : \mathbb{S}$ (String)	$d ::= e_1\ op_r\ e_2 \mid \neg d \mid d_1 \vee d_2 \mid d_1 \wedge d_2 \mid true \mid false$, where $op_r \in \{=,\geq,\leq,<,\dots\}$
$v_a : \mathbb{V}_a$ (Application Variables)	$g(\vec{e}) ::= \texttt{GROUP BY}(\vec{e}) \mid id$
$v_d : \mathbb{V}_d$ (Database Attributes)	$r ::= \texttt{ALL} \mid \texttt{DISTINCT}$
$e : \mathbb{E}$ (Arithmetic Expressions)	$s ::= \texttt{AVG} \mid \texttt{SUM} \mid \texttt{MAX} \mid \texttt{MIN} \mid \texttt{COUNT}$
$d : \mathbb{D}$ (Boolean Expressions)	$h(e) ::= s \circ r(e) \mid \texttt{DISTINCT}(e) \mid id$
$A : \mathbb{A}$ (Action)	$h(*) ::= \texttt{COUNT}(*)$
$\tau : \mathbb{T}$ (Terms)	$\vec{h}(\vec{x}) ::= \langle h_1(x_1),\dots,h_n(x_n)\rangle$, where $\vec{h} = \langle h_1,\dots,h_n\rangle$ and $\vec{x} = \langle x_1,\dots,x_n\rangle$
$a_f : \mathbb{A}_f$ (Atomic Formulas)	$f(\vec{e}) ::= \texttt{ORDER BY ASC}(\vec{e}) \mid \texttt{ORDER BY DESC}(\vec{e}) \mid id$
$\phi : \mathbb{W}$ (Pre-condition)	$q ::= \langle E_A, E_\phi \rangle$
$Q : \mathbb{Q}$ (SQL statements)	$A ::= \texttt{SELECT}(v_a,\ f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi,\ g(\vec{e})) \mid \texttt{UPDATE}(\vec{v_d},\ \vec{e}) \mid \texttt{INSERT}(\vec{v_d},\ \vec{e}) \mid \texttt{DELETE}(\vec{v_d})$
$I : \mathbb{I}$ (Imperative statements)	$\tau ::= n \mid k \mid v_a \mid v_d \mid f_n(\tau_1,\tau_2,\dots,\tau_n)$, where f_n is an n-ary function.
$c : \mathbb{C}$ (Statements)	$a_f ::= R_n(\tau_1,\tau_2,\dots,\tau_n) \mid \tau_1 = \tau_2$, where $R_n(\tau_1,\tau_2,\dots,\tau_n) \in \{true, false\}$
	$\phi ::= a_f \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \forall x_i\ \phi \mid \exists x_i\ \phi$
	$I ::= skip \mid v := e$
	$c ::= Q \mid I \mid$ if b then c_1 else $c_2 \mid$ while b do c

Table 1: Syntax and semantics of query languages

tween the statements. We will first find out the defined and used part of each statement and finally use that data to calculate the DD-dependency depending on semantics.

1) Computing the Defined and Used Part:

We must determine the database components of the statements that will be defined or used by the statements of q in order to compute the dependence between two database statements, q1 and q2. Hence, The semantics of q are divided into three categories, $\psi_\circ = < q_t, q_f, q_m >$. In this case, q_t stands for the true component, q_f for the false part, and q_m for the modified section. Now, let's specify two functions $Func_d$ and $Func_u$. Let Dq and Uq indicate, respectively, the *Defined* and *Used* database components of the database statement q:

$$D_q = Func_d\ (q, \psi_\circ) = < q, q_t, q_m >$$

$$U_q = Func_u\ (q, \psi_\circ) = < q_t >$$

2) Computing Dependency:

Two statement, for example A_1 and A_2 are said to be dependent on each other iff $D_q^{A_1} \cap U_q^{A_2} \neq \emptyset$. We will first generate the pairwise dependency of two statements. It will have four possibilities to identify the independency and the semantic dependency among them. $D_q^{A_1}$ can be expressed by two components, $D_q^{A_1} = < E_A^{A_1}, E_\phi^{A_1} >$, according to the condition-action rule based method where, $E_A^{A_1}$ represents the action part and $E_\phi^{A_1}$ represents its condition part. Also, $U_q^{A_1}$ can be represented by, $< Q_\phi^{A_1} >$. Similarly, for $D_q^{A_2}$ and $U_q^{A_2}$ can be represented as: $D_q^{A_2} = < E_A^{A_2}, E_\phi^{A_2} >$ and $U_q^{A_2} = < Q_\phi^{A_2} >$.

The semantic independency and dependency are shown below:

1. $E_\phi^{A_1} \cap Q_\phi^{A_2} \neq \emptyset \wedge E_A^{A_1} \cap Q_\phi^{A_2} = \emptyset$
2. $E_\phi^{A_1} \cap Q_\phi^{A_2} = \emptyset \wedge E_A^{A_1} \cap Q_\phi^{A_2} \neq \emptyset$
3. $E_\phi^{A_1} \cap Q_\phi^{A_2} \neq \emptyset \wedge E_A^{A_1} \cap Q_\phi^{A_2} \neq \emptyset$
4. $E_\phi^{A_1} \cap Q_\phi^{A_2} = \emptyset \wedge E_A^{A_1} \cap Q_\phi^{A_2} = \emptyset$

Fig. 6 shows a visual depiction of the aforementioned four circumstances. Red color indicates $Q_\phi^{A_2}$, blue indicates $E_\phi^{A_1}$ and green indicates $E_A^{A_1}$. From this figure, we can see that semantic independency between statement A_1 and A_2 is observed only in case of condition 4. The remaining three cases shows the semantic dependency between the statements.
Now, by employing Condition-Action rule-based strategy, we will illustrate our running example to remove our false dependencies [47] present in the code snippet. We can see that $E_{A_{upd}}^{5}$ acts on a part of data which is not used by E_ϕ^7, as a result, it is a false dependency. Similarly, there is a faulty relationship between $5 \rightarrow 9$. As a result, the Condition-Action rule-based method eliminates one of the false dependency from our running example which is shown in Fig. 7.

3) Limitations of Condition-Action rule based approach

Although, it is an efficient semantics based analysis, it has some major drawbacks which needs to be considered. Let us consider our running example from section IV. If three consecutive statements are present in a program and there exist a false dependency between them, it cannot be identified using the Condition-Action rule based approach. In our example, there is a false dependency between 5→9 as there is another update statement in statement 8 which will affect the result of statement 9. It is represented in the Fig. 8. Statement 5 is updating some amount of data, another amount of data is being updated by 8 and then after that 9 is also updating some amount of data. In this case 9 will only consider the data which was last updated by statement 8. Statement 8 is overwriting statement 5 so statement 9 will depend only on the result of statement 8. Therefore there is a false dependency between 5 and 9 which will not be identified by Condition-Action rule based approach. Also, it has high computational cost of $O(2^n)$, where n represents the number of variables in the program and is unable to generate optimal solution.
Consequently, we require more accurate semantics-based analysis which will be computationally cost effective and can completely identify and remove the false dependencies present in the program.

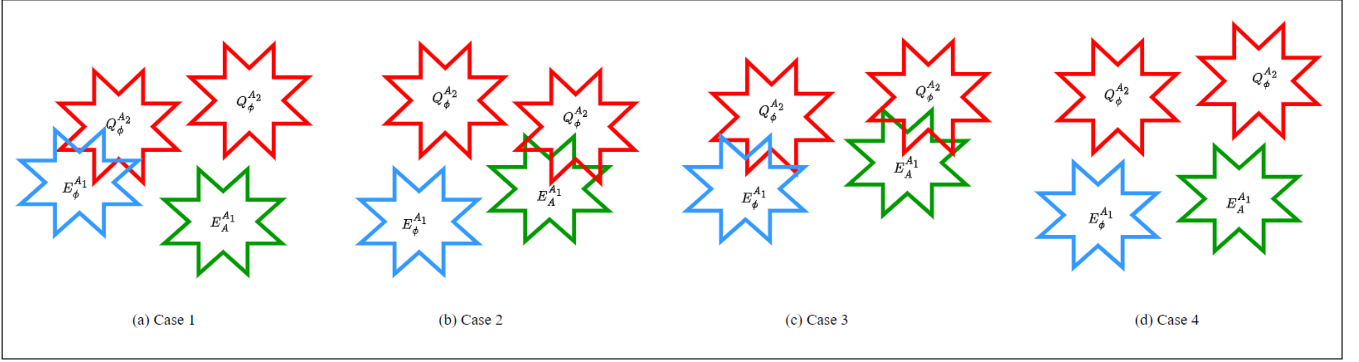

Figure. 6: Representations of dependencies and independencies

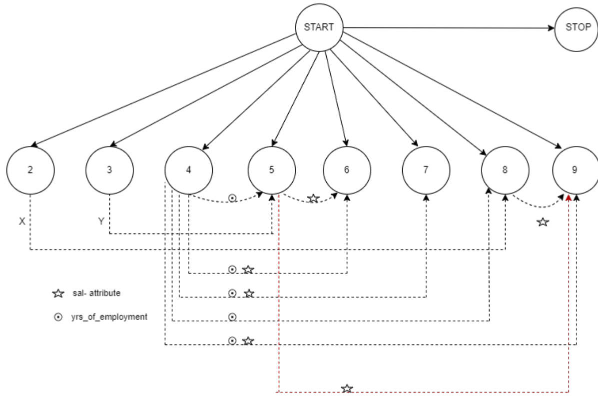

Figure. 7: Refined DDG of Program "`Prog`" using Condition-Action

C. Refinement Using Hoare Logic

Hoare logic is another formal method of refinement technique which was proposed by British computer scientist and logician *Tony Hoare* in 1969 [48]. The original idea were based on the work of *Robert W. Floyd*.
Hoare logic's primary goal is to offer a formal mechanism for assessing *program correctness*. It is like a contract between the implementation of a function and its clients. The main feature of hoare logic is its *Hoare Triple*. It can be represented by: *{P} S {Q}*, where P is the *precondition*, Q is the *postcondition*, and S is the statement to be executed. The precondition is a predicate that explains the condition on which the function relies for proper execution and must be met by the client. The postcondition is a predicate that describes the condition that the function creates after it has been executed successfully. We take a broad view of a situation in which the SQL instructions are included into another high-level host language.
If the precondition is satisfied shortly before the function is executed and the postcondition is satisfied if the function finishes, the implementation of a function is *partially correct* with regard to its definition. Similar to this, the implementation is 100% accurate if it is assumed that the precondition is true before the function executes, the function is bound to terminate, and the postcondition is true at the time of termination. Total correctness is thus the sum of partial correctness and its termination.
It should be emphasised that a client can occasionally call a function without completing its precondition, in which case the function can act in any way and still be valid. As a result, it is critical that the function be error-tolerant, and the precondition must incorporate the potential of erroneous input, while the postcondition should define what should happen in the event of such mistakes. For example, one such method could be throwing of exceptions to handle such errors. The weakest precondition WP [48] of an iteration free imperative program is computed as follows, given a program statement *'stmnt'* and a postcondition φ:

$$\{WP(skip, \varphi) = \varphi \qquad WP(v{:=}\ e, \varphi) = \varphi[e/v]\}$$

$$\{WP(stmnt_1; stmnt_2, \varphi) = WP(stmnt_1, WP(stmnt_2, \varphi))\}$$

$$\{WP(\text{if } d \text{ then } stmnt_1 \text{ else } stmnt_2 \text{ endif}, \varphi) =$$

$$(d \land wp(stmnt_1, \varphi)) \lor (\neg d \land WP(stmnt_2, \varphi))\}$$

To handle database programs, we define **WP** (*Weakest Precondition*) [48] on database statements as shown below.

$$\{WP(skip, \varphi) = \varphi \qquad WP(v{:=}\ e, \varphi) = \varphi[e/v]\}$$

$$\{WP(assume\ \varphi_1, \varphi_2) = \varphi_1 \implies \varphi_2 \qquad WP(assert\ \varphi_1, \varphi_2) = \varphi_1 \land \varphi_2\}$$

$$\{WP(<RS{:=}\ SEL(f(\vec{e}),\ r(\vec{h}(\vec{x})), \varphi,\ g(\vec{e})), \phi>, \varphi) = (\varphi[F(\vec{a})/RS] \land \phi) \lor (\varphi \land \neg \phi)\}$$

$$\{WP(<UPD(\vec{e}), \phi>, \varphi) = ((\varphi \land \neg\phi) \land \varphi[\vec{e}/\vec{a}] \land \phi))\}$$

$$\{WP(<\vec{a}{:=}INS(\vec{e}), false>, \varphi) = \varphi[\vec{e}/\vec{a}] \lor \{\varphi\}\}$$

$$\{WP(<\vec{a}{:=}DEL(), \phi>\varphi) = \varphi \land \neg\phi))$$

$$WP(stmnt_1; stmnt_2, \varphi) = WP(stmnt_1, WP(stmnt_2, \varphi))\}$$

$$\{WP(\text{if } d \text{ then } stmnt \text{ endif}, \varphi) = (d \land WP(stmnt, \varphi)) \land (\neg d \land \varphi)\}$$

$$\{WP(\text{if d then } stmnt_1 \text{ else } stmnt_2 \text{ endif}, \varphi) = (d \land WP(stmnt_1, \varphi)) \lor (\neg d \land WP(stmnt_2, \varphi))\}$$

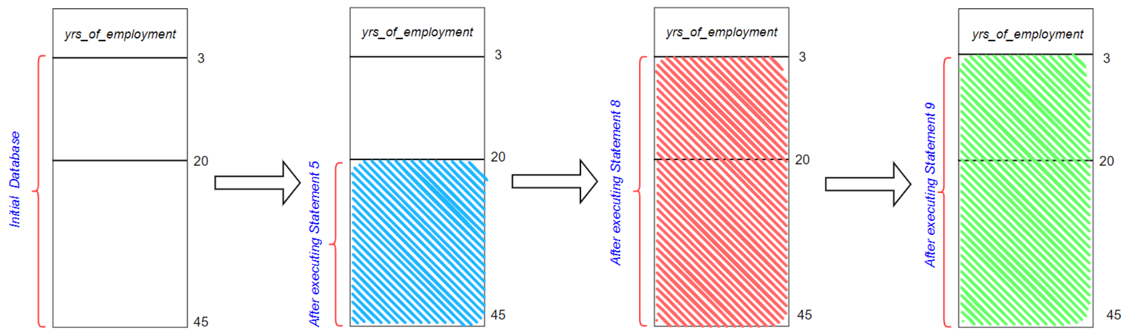

Figure. 8: Representation of overlapping part of the database after execution of the three subsequent statements 5, 8 and 9 of "`Prog`".

In order for a program to be accurate, it must start in a state where P is true, execute S, and end in a state where Q is true. This is what "hoare triple" means. Let us take an example, consider the Hoare triple {x=10}x = x*3{x>0}. It is evident that the triple is accurate, because if x=10 and we multiply it by 3 it will be x=30, which is more than 0, i.e, it is satisfying the postcondition. Although true, this hoare triple's postcondition is not particularly exact. We can provide more stronger postcondition which will be more informative and will pin down the value of x to a certain range. For example, instead of writing {x>0} , a more precise postcondition would be {x>20 && x<50}. However, the strongest postcondition would be if {x=30}. Formally, Q is the strongest postcondition of S with regard to P if {P}S{Q} for any Q such that {P}S{Q}, Q $\implies$ Q. Similarly, for the above example, x=30 is not the only valid precondition. It can also be x>0 or x> 5 etc. Although, declaring a constant gives us the guaranteed postcondition, they are not technically correct. They are more restrictive about the values of x for which the function is guaranteed to be correct. We usually want to use the precondition that guarantees correctness for a broader set of inputs. Therefore, technically we need the weakest precondition, i.e, the most general precondition to construct the precondition. Therefore, we can write it as {x>0}x=x*4{x>1}, which is metaphorically stronger than declaring it to a constant. Formally, it can be written as P = WP(S, Q) which indicates that P is the weakest precondition for function S and postcondition Q.
Using hoare logic we can refine the data dependency as in the following steps as mentioned. (i) Construction of the Action Tree, (ii) Calculating Traces, (iii) Backtracking with Pre-Conditions:, (iv) Product of observational-window and traces, (v) Recognizing dependences [49].
To illustrate the above proposed approach, let us take a small database code snippet (Fig. 9) and a database table (Table 2) to discuss the different phases in details.

Construction of the Action Tree: If we consider a database code snippet q, it can be represented abstractly as q = < E_A, E_ϕ >, where E_A represents the action component and E_ϕ represents the condition component of the database statement. The construction of the action tree is a stage that divides the database statements into condition-action components using the condition-action portion of the database statement (e.g., INSERT, DELETE, and UPDATE). This division of the database information generates a tree-like structure, known as the *Action Tree*. The edges in the tree depict the conditions of the database statement, and the nodes represent the actions. Let us illustrate the construction of *Action Tree* in details.

ID	PRODUCT	PRICE
3001	Vanilla Ice-Cream	130
3002	Sphagetti	90
3003	Ravioli	100
3004	Semolina	110
3005	Millet	150
3006	Cookies	120
3007	Wheat Flour	155
3008	Wafers Jumbo Pack	180
3009	Ground Coffee	100
3010	Doritos	105

Table 2: DB Table for Product

Example 3 *Given a Program P which is depicted in Fig. 9. Let us denote each statement as s1, s2, s3 and s4. Analyzing the program we can see that the order of definition of PRICE will be* $s1 \rightarrow s2 \rightarrow s3$.
At the program point s1, the statement functions as a declaration for all the attributes in the database. This declaration is referred to as the action component and is represented as "s1: DB" by a child node. Since there is no condition component present, it is simply labeled as "s1: T_r" (T_r denotes True).
In statement s2, the condition part $E_\phi = 100 \leq PRICE \leq 150$ divides the database in two parts: one that satisfies the condition (say, E_ϕ) and one which do not satisfy the given condition (say, $\neg E_\phi$). The action PRICE′=PRICE+20 is applied on E_ϕ, whereas for $\neg E_\phi$ remains same. According to condition-action rule, both $s1 \xrightarrow{E_\phi} s2$ and $s1 \xrightarrow{\neg E_\phi} s2$ exists. Hence, we create two child nodes: one denotes the action s2: PRICE′=PRICE+20 with edge labelled by s2: E_ϕ i.e, 100≤PRICE≤150 and the other denotes s2: PRICE′=PRICE with the edge labelled as s2: $\neg E_\phi$ i.e, PRICE<100 ∧ PRICE>150. Similar approach is applied for the statement at s3. Regarding the statement $s2 \xrightarrow{\neg E_\phi} s3$, the edge is labeled $\neg E_\phi$= PRICE<90∧PRICE>155, which violates the condition-action rule and thus, there is no child node assigned to the action PRICE″=PRICE′. The Fig. 10 representing the final Action Tree for program P is depicted below.

Calculating Traces: A trace in an Action Tree is a sequence of labels that starts at the root node and ends at a leaf node. The traces of an Action Tree can be obtained by following the path from the root node to its leaf nodes.
In formal terms, a trace is expressed as $<(s_i\!: E_\phi, s_i\!: E_A)>_{i\geq 1}$, where E_ϕ symbolizes the condition component and E_A symbolizes the action component of given DB statement.

```
1. Statement con = DriverManager.getConnection("jdbc sql: . . . ", "scot", "tig").createStatement();
2. con.executeQuery("UPDATE Product SET PRICE = PRICE + 20 WHERE PRICE BETWEEN 100 AND 150 ");
3. con.executeQuery("UPDATE Product SET PRICE = PRICE + 10 WHERE PRICE BETWEEN 90 AND 155 ");
4. ResultSet rs1=con.executeQuery("SELECT PRICE FROM Product WHERE PRICE BETWEEN 90 AND 160 ");
```

Figure. 9: Program P

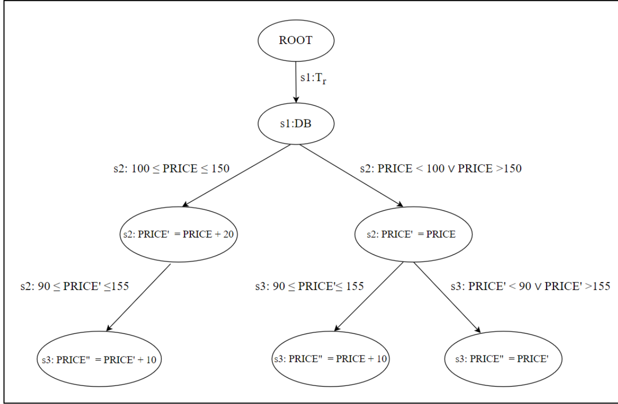

Figure. 10: Action Tree

Example 4 *Let us illustrate the given action tree from Fig. 10 in traces.*

μ_1= (s1:T_r, s1:DB) (s2:100≤PRICE≤150, s2:PRICE′=PRICE+20) (s3:90≤PRICE′155, s3:PRICE″=PRICE′+10)

μ_2= (s1:T_r, s1:DB) (s2:PRICE<100∨PRICE>150, s2:PRICE′= PRICE) (s3:90≤PRICE′≤155,s3:PRICE″=PRICE′+10)

μ_3= (s1:T_r, s1:DB) (s2:PRICE<100∨PRICE>150, s2:PRICE′= PRICE)(s3:PRICE′<∨ PRICE′ > 155, s3:PRICE″= PRICE′)

Backtracking with Pre-Conditions: As previously discussed, Hoare logic is a deductive system. Our goal is to determine the semantic-based dependencies between a used attribute and all previous declaration statements. To achieve this, we use the condition component of the statements as an observational window for the used statement. By considering the observational window of the used statement, we calculate the weakest pre-condition. The purpose of this is to trace the flow of definitions and to assess if it influences the observational window.

Example 5 *Let's consider the program P depicted in Fig. 9. For statement s4, the observational window is defined as (90≤PRICE″≤160) and can be represented as: w4= s4: 90≤ PRICE″ ≤ 160. By using Hoare logic, we obtain,*

{w1} s1: DB {w2} s2: PRICE′=PRICE+20 {w3} s3: PRICE″= PRICE′+10 {w4}

where, w3= s3: 80≤PRICE′ ≤150,
w2=s2: 60≤ PRICE ≤ 130,
w1=s1: 60≤ PRICE ≤ 130

The flow of sequence can also be illustrated pictorially as shown in Fig. 11.
As we can see, the sequence of the assertions above follows an observational trace which can be written as,
$\mu_\circ$= w1w2w3w4= (s1:60 ≤ PRICE ≤ 130) (s2: 60 ≤ PRICE≤130) (s3:80 ≤ PRICE′≤150) (s4:90≤ PRICE″≤160)
Formally, an observational trace can be defined as $<(s_j : E\phi)>_{j\geq 1}$, *where* E_ϕ represents a set of well-formed formulas.

Product of observational-window and traces: The product of an action tree trace μ and an observational trace $\mu_\circ$ involves taking the cross product of the two sequences, resulting in a new sequence of pairs. This can be expressed as,
$\mu \times \mu_\circ = <(s_i : E'_\phi, s_i : E'_A)>_{i\geq 1} \times <(s_j : E''_\phi)>_{j\geq 1} = <(s_x, : p, s_x : q)>_{x\geq 1}$ where,

$$(s_x : p, s_x : q) = \begin{cases} \text{if } \exists\, i, j : i = j \text{ and } E'_\phi \wedge E''_\phi \neq \emptyset \\ (s_i : E'_\phi, s_i : E'_A) \times (s_j : E''_\phi) = (s_i : T_r, s_i : E'_A) \\ \\ \text{if } \exists\, i, j : i = j \text{ and } E'_\phi \wedge E''_\phi = \emptyset \\ (s_i : E'_\phi, s_i : E'_A) \times (s_j : E''_\phi) = (s_i : F_a, s_i : E'_A) \\ \\ (s_j : E''_\phi, s_j : Obs) \text{ if } \nexists\, i : j = i \end{cases} \tag{1}$$

Here, T_r denotes true value, F_a denotes false value and Obs represents '*observe*'.

Example 6 *Let us take the example of Program P from Fig. 9. The result of the cross product between the action-tree trace (μ) and the observational trace ($\mu_\circ$) of Program P in Figure can be obtained using equation 1.*

$\mu_1 \times \mu_\circ = (s1 : T_r, s1 : DB)(s2 : T_r, s2 : PRICE' = PRICE + 20)$ $(s3 : T_r, s3 : PRICE'' = PRICE' + 10)(s4\!: 90 \leq PRICE'' \leq 160, s4 : Obs)$
$\mu_2 \times \mu_\circ = (s1 : T_r, s1 : DB)(s2 : T_r, s2 : PRICE'' = PRICE)$ $(s3 : T_r, s3 : PRICE'' = PRICE' + 10)(s4\!: 90 \leq PRICE'' \leq 160, s4 : Obs)$
$\mu_3 \times \mu_\circ = (s1 : T_r, s1 : DB)(s2 : T_r, s2 : PRICE' = PRICE)$ $(s3 : T_r, s3 : PRICE'' = PRICE')(s4\!:\!90 \leq PRICE'' \leq 160, s4 : Obs)$

Recognizing dependences: The process of identifying dependences involves converting traces into binary values of either *yes* or *no*. Traces in which the state is altered will be replaced with "Y", while traces that do not alter the state will be marked as "N" that denotes Yes and No respectively.

Example 7 *Applying the conversion of traces to Program P we get the following result.*

$\mu_1^m = (s1 : T_r, s1 : Y)(s2 : T_r, s2 : Y)(s3 : T_r, s3 : Y)$ $(s4\!:\!90 \leq PRICE'' \leq 160, s4 : Obs)$
$\mu_2^m = (s1 : T_r, s1 : Y)(s2 : T_r, s2 : N)(s3 : T_r, s3 : Y)$ $(s4 : 90 \leq PRICE'' \leq 160, s4 : Obs)$
$\mu_3^m = (s1 : T_r, s1 : Y)(s2 : T_r, s2 : N)(s3 : T_r, s3 : N)$ $(s4 : 90 \leq PRICE'' \leq 160, s4 : Obs)$

Given a set of masked traces, we will now apply a filter to filter out the actions which are irrelevant to our semantic dependency computation. The filter on masked traces can be denoted as:

$FIL(\mu)$={(s_k:u, s_k:v)| u= false ∨ v= no}

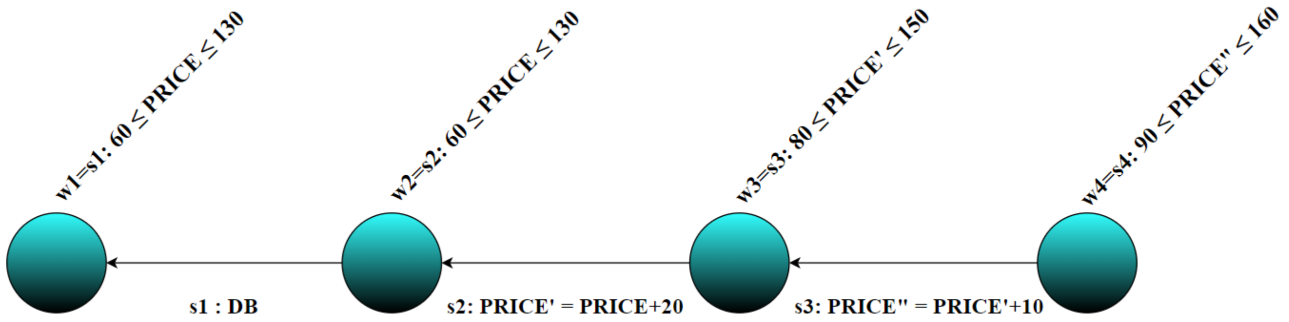

Figure. 11: Backwards with pre-condition

Example 8 *Let us apply the filter to our masked traces and find the refined actions.*

$FIL(\mu_1^m) = (s1 : T_r, s1 : Y)(s2 : T_r, s2 : Y)(s3 : T_r, s3 : Y)$
$(s4 : 90 \leq PRICE'' \leq 160, s4 : Obs)$

$FIL(\mu_2^m)= (s1 : T_r, s1 : Y)(s3 : T_r, s3 : Y)(s4 : 90 \leq PRICE'' \leq 160,$ $s4 : Obs)$

$FIL(\mu_3^m)= (s1 : T_r, s1 : Y)(s4 : 90 \leq PRICE'' \leq 160, s4 : Obs)$

The following function will identify the semantic-based dependencies from the filtered traces, using a trace element e=(s_k: u, s_k:v). The label of the trace element is obtained by the function $L_a(e) = k$.

$$D_p\ (\mu) = D_p(<e_1, e_2, e_3,...., e_n>) = L_a(e_{n-1}) \rightarrow L_a(e_n)$$

Example 9 *We will apply the above function on our refined traces which we have obtained from the Program P (Fig. 9), to extract the semantic-based dependency present in the program:*

$$D_p(FIL(\mu_1^m))= s_3 \rightarrow s_4,$$
$$D_p(FIL(\mu_2^m))= s_3 \rightarrow s_4,$$
$$D_p(FIL(\mu_3^m))= s_1 \rightarrow s_4$$

Dependency computation is an important step to omit the false dependencies present in a given database to avoid errors and its inconsistencies as we have discussed on the previous sections.
Now, let us consider our code snippet "`Prog`" from Fig. 3. We denote each statement as s_1, s_2, s_3,..., s_{10}. We will consider computation from s_4 statement since we need to identify false dependencies in the database part of the program. For dependency computation we will compare each statement to its subsequent statements. For Eg., it needs to satisfy the given condition.

$$\{ s_{kpre_m} \cap s_{kpost_n} \} \vee \{ s_{kpre_m} \cap s_{kpost_{n+1}} \} \vee \ldots \vee \{ s_{kpre_m} \cap s_{kpost_n} \} = \phi$$

The above condition suggests that for a given precondition say s_{kpre_m}, if it does not satisfy its given postcondition say s_{kpost_n} and there is no absolute relation amongst the two statements then it indicates a false dependency between the two considered statement.

In our database code snippet "`Prog`", if we take statement s_5 and find what kind of relation/dependency does exist between s_5's precondition and the postconditions of its subsequent statements. First we consider s_5 with s_6. If we perform intersection between the precondition of s_5 and postcondition of s_6 we find that the range of window remains the same and there is a relation between the two, hence no false dependency. Now, we consider s_5 with s_7. Here we can see that there is no common window range between the two conditions even though it is using the variable *sal* as defined in statement s_5. Since, there is no actual dependency among the two it can be identified as a false dependency. In statement s_9 we can see that our original value for the variable *sal* is overwritten by statement s_8. Hence, the range of window considered in s_9 will be in reference to the updated variable *sal* done in s_8. Hence, there exists a false dependency between the two statements since our original value defined would be overwritten and will change the structure of the database table.
We can clearly see from the Fig. 12 that the multi-level false dependency of three subsequent statements which condition-action rule based approach failed to identify has also been removed using the hoare logic approach. Therefore, we get more precise result.

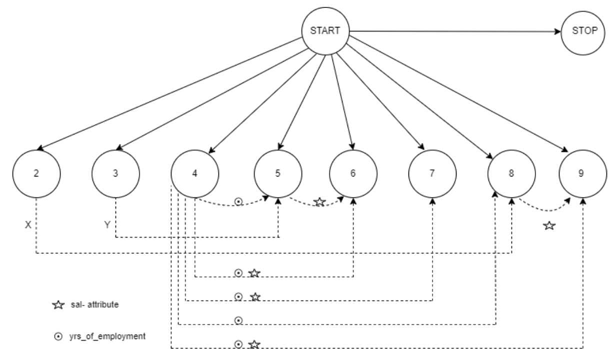

Figure. 12: Refined DDG of Program "`Prog`" using Hoare logic

D. Slicing Computation

Now, we compute the slice of "`Prog`" using the refined DDG of Hoare logic. Slicing performs in two directions (i) Forward Slicing- extracting those parts of the program which will be affected by the slicing criterion (ii) Backward Slicing- extracting those parts of the program which are affected by the slicing criterion. Let us consider a policy ψ as follows:

ψ = *The average salary of employees should be more than 20% of the minimum salary of the category*

From ψ, we extract the slicing criteria $S = < 9, sal >$, where sal is the database attribute and 9 is the last program point of "`Prog`". The goal of the backward slicing is to only extract the sentences that have an impact on the values of *sal* at various program points.

We compute slicing of "Prog" w.r.t S using backward slicing algorithm [11]. We consider refined database dependency graph of "Prog" (Fig. 12) as an input and the algorithm produces the sub-graph of the refined database dependency graph. The sub-graph is depicted in Fig. 14 and the node of the sub-graph represents the slicing of "Prog" w.r.t ψ which is shown in Fig. 13.

VI. Experimental Results

Using the Satisfiability Modulo Theorem (SMT), we put the Condition-Action rule-based method into practise [50]. We specifically used Z3 tool, which is a high performance SMT solver developed by Microsoft Research [51][52]. The experiment is carried out on a machine equipped with i5 processor, a clock speed of 1.60GHz, Windows 10 64-bit OS having 4GB of RAM.

We evaluate experimental results on a set of benchmarks codes (.jsp files) [50] using Z3 tool. The used benchmark programs are usually database applications which are open source implemented in jsp program. To compute results, we perform various steps which are: (i) Database statements are chosen in pairs based on the sequence in which they appear in the code. (ii) These database statements has to be converted to Static Single Assignment (SSA) (iii) Pairwise Verification Condition (VC) generation from the SSA form by extraction of the condition and action components from the provided statements, and finally (iv) using the Z3 tool, dependency verification based on VC satisfiability is performed.

Example 10 *Let us consider the following pair of database statements.*

Q1: UPDATE events SET no_of_days=no_of_days+1 WHERE no_of_presenters≥20
Q2: UPDATE events SET no_of_days=no_of_days-1 WHERE no_of_presenters≤14

The aforementioned statements' SSA equivalents are:

Q1: UPDATE events SET no_of_days2 = no_of_days1+1 WHERE no_of_presenters1≥20
Q2: UPDATE events SET no_of_days3=no_of_days1-1 WHERE no_of_presenters1≤14

The VCs for the above statements are:

(no_of_days2==no_of_days1+1)∧(no_of_presenters1>=20)∧ (no_of_days3==no_of_days1-1)∧(no_of_presenters1<=14)

The Z3 encoded code can be written as:

```
1. (declare-const no_of_days1 Int)
2. (declare-const no_of_days2 Int)
3. (declare-const no_of_days3 Int)
4. (declare-const no_of_presenters1 Int)
5. (push)
6. (assert(=(+ no_of_days1 1)no_of_days2))
7. (assert(>=(+ no_of_presenters1 0)20))
8. (assert(=(- no_of_days1 1)no_of_days3))
9. (assert(<=(+ no_of_presenters1 0)14))
10. (check-sat)
```

Z3 indicates whether these two statements are dependent or independent of each other. The result is depicted in Table 3. Observe that, we obtain precise dependencies using our proposed approach. Note that, NCLOC represents Non-comment lines of code and Attr represents the attributes. The corresponding graphical result is represented in Fig. 15.

VII. Discussion

Program slicing pulls a subset of statements from an original source code that are relevant to a specific behaviour. This enables programmers to solve numerous software-engineering issues like debugging, maintenance, testing, etc. In the literature, we observe that syntactic-based computation of slicing may generate imprecise results as it may contain false dependencies. The reason behind this is that syntax-based computation focuses on the variables rather than the values. Therefore, as the syntactic presence of variables is inadequate to represent importance, the syntax-based method may fail to calculate a precise list of dependents. For example, consider the statement "a = y + 2 × z % 2" where a is syntactically reliant on z. However, there is no semantic reliance of z on a. On the other hand, in the limitation of the syntax-based approach (Section V-A) the syntax-based dependency between two SQL statements generates false dependency because we focus on attributes instead of values. Therefore, we enhance the syntax-based database dependency graph into a more refined semantics-based database dependency graph.

Although, in the case of the Condition-Action rules-based approach we achieved a more precise slicing result as compared to the syntax-based approach. Moreover, it fails to compute optimal results because it is not a flow-sensitive approach. Therefore, we consider the semantic approach Hoare Logic where we removed more false alarms and obtained more precise results compared to the condition action rule-based approach. In this case, we follow the following steps for the analysis.

- Constructing an action tree representation of the program's actions which helps in visualizing the program's flow of control.
- Computing the traces which are the sequences of program statements that are executed during the program's execution.
- Performing backtracking with Pre-Conditions where the precondition is applied backward through the action tree to determine the values of database attributes before the execution of the code.
- Computing the product of traces and the observational window, which is the time window in which the dependencies are being analyzed.
- Finally, identifying the dependencies between the program's variables by analyzing the product of the traces and the observational window.

This way this semantic-based approach preserves the data flow and captures more false dependencies which were not removed using the Condition-Action rules-based approach. Observe that, in this case, we consider the triplet of three statements for the dependency analysis (instead of the pair-wise) that make this approach flow-sensitive. These two semantics-based approaches suffer from high computational complexity (i.e. exponential), however, we obtain more precise slicing results in the case of Hoare Logic as compared to Condition-Action rules-based approach.

VIII. Conclusion

Program slicing is the most prominent tool for performing many software engineering activities (like code understand-

```
1. START
2. float x = 0.05;
3. float y = 0.15;
4. Statement con = DriverManager.getConnection("jdbc sql: . . . ", "scot", "tig").createStatement();
5. con.executeQuery("UPDATE Emp SET sal = sal + y * sal WHERE yrs of employment ⩾ 20 ");
8. con.executeQuery("UPDATE Emp SET sal = sal + x * sal WHERE yrs of employment BETWEEN 3 AND 45 ");
9. ResultSet rs1=con.executeQuery("SELECT AVG(sal) FROM Emp WHERE yrs_of_employment BETWEEN 3 AND 45 ");
```

Figure. 13: Sliced program "Prog"

Applications (File Names)	*NCLOC*	*SQL Stmnts*	*Attributes*	*Syntax based*	*Condition action*
Events(EventNew.jsp)	277	6	5	6	4
Ledger(LedgerRecord.jsp)	365	7	5	14	10
EmplDir(DepsRecord.jsp)	233	5	4	4	3
EmplDir(EmpsRecord.jsp)	367	6	4	6	4
Bookstore(EditorialsRecord.jsp)	240	4	3	5	2
Portal(EditOfficer.jsp)	340	8	4	10	9
Bookstore(BookMaint.jsp)	406	6	5	7	3
Bugtrack(ProjectMaint.jsp)	343	7	4	10	8
Bugtrack(EmployeeMaint.jsp)	368	6	4	12	11

Table 3: Results of DD-Dependency using the semantics rule-based method.

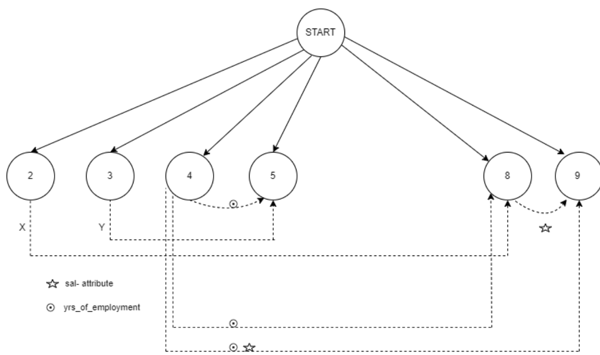

Figure. 14: Sub-DDG of program "Prog"

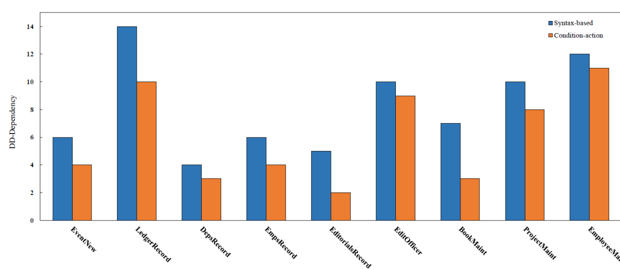

Figure. 15: Comparative analysis among the syntax-based and the condition-based approach.

ing, debugging, maintenance, testing, etc.) of a large and complex information system. In this paper, we proposed a novel program-slicing framework for data-intensive programs in information system scenarios where it considers the external database states as well. We design the framework using a data dependency graph and refined the dependency graph (by removing false dependencies) using semantics-based approaches Condition-Action rules and Hoare Logic. Then we compute the slicing w.r.t the policy on this refined dependency graph that leads to precise slicing result. Finally, we evaluate the experimental results on benchmark codes. This framework serves as a powerful tool to solve the above-mentioned software-engineering problems relating to query languages and underlying databases. However, the used semantics-based approaches suffer from high computational costs (exponential). Therefore, further investigation and implementation of other suitable semantics-based techniques will be our future aim.

References

[1] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 2015.

[2] B. Cherry, P. Benats, M. Gobert, L. Meurice, C. Nagy, and A. Cleve, "Static analysis of database accesses in mongodb applications," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 930–934.

[3] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 4, pp. 323–337, 1992.

[4] F. Tip, "A survey of program slicing techniques," Amsterdam, The Netherlands, Tech. Rep., 1994.

[5] C. Hammer, "Experiences with pdg-based ifc," in *Engineering Secure Software and Systems: Second International Symposium, ESSoS 2010, Pisa, Italy. Proceedings 2*. Springer, 2010, pp. 44–60.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[7] A. Podgurski and L. A. Clarke, "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Transactions on software Engineering*, vol. 16, no. 9, pp. 965–979, 1990.

[8] L. Jiang, *Scalable detection of similar code: Techniques and applications*. University of California, Davis, 2009.

[9] J. Krinke, "Information flow control and taint analysis with dependence graphs," in *3rd International Workshop on Code Based Security Assessments (CoBaSSA)*, 2007, pp. 6–9.

[10] A. Jana, R. Halder, N. Chaki, and A. Cortesi, "Policy-based slicing of hibernate query language," in *IFIP International Conference on Computer Information Systems and Industrial Management*. Springer, 2015, pp. 267–281.

[11] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.

[12] D. Willmor, S. M. Embury, and J. Shao, "Program slicing in the presence of database state," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 448–452.

[13] N. AlAbwaini, A. Aldaaje, T. Jaber, M. Abdallah, and A. Tamimi, "Using program slicing to detect the dead code," in *2018 8th International Conference on Computer Science and Information Technology (CSIT)*. IEEE, 2018, pp. 230–233.

[14] A. Cleve, "Program analysis and transformation for data-intensive system evolution," in *Proc. of the 26th Int. Conf. on Software Maintenance*. IEEE CS, 2010, pp. 1–6.

[15] Y. Sivagurunathan, M. Harman, and S. Danicic, "Slicing, i/o and the implicit state," in *Proc. of the 3rd Int. Workshop on Automatic Debugging*, 1997, pp. 59–68.

[16] H. B. K. Tan and T. W. Ling, "Correct program slicing of database operations," *IEEE Software*, vol. 15, pp. 105–112, 1998.

[17] H. W. Alomari and M. Stephan, "Clone detection through srcclone: A program slicing based approach," *Journal of Systems and Software*, vol. 184, p. 111115, 2022.

[18] I. Mastroeni and D. Zanardini, "Data dependencies and program slicing: from syntax to abstract semantics," in *Proc. of the ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM Press, 2008, pp. 125–134.

[19] E. Soremekun, L. Kirschner, M. Böhme, and A. Zeller, "Locating faults with program slicing: an empirical analysis," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–45, 2021.

[20] A. Kashyap and A. Jana, "Policy-based code slicing of database application using semantic rule-based approach," in *Innovations in Bio-Inspired Computing and Applications: Proceedings of the 13th International Conference on Innovations in Bio-Inspired Computing and Applications (IBICA)*. Springer, 2023, pp. 392–403.

[21] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1–36, 2005.

[22] H. Zhang, H. B. K. Tan, L. Zhang, X. Lin, X. Wang, C. Zhang, and H. Mei, "Checking enforcement of integrity constraints in database applications based on code patterns," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2253–2264, 2011.

[23] A. Formica and M. Missikoff, "Integrity constraints representation in object-oriented databases," in *Information and Knowledge Management Expanding the Definition of "Database" First International Conference, CIKM'92 Baltimore, Maryland, USA*. Springer, 1993, pp. 69–85.

[24] K. Ahmed, M. Lis, and J. Rubin, "Mandoline: Dynamic slicing of android applications with trace-based alias analysis," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 105–115.

[25] N. Sasirekha, A. E. Robert, and D. M. Hemalatha, "Program slicing techniques and its applications," *arXiv preprint arXiv:1108.1352*, 2011.

[26] W. Muylaert and C. De Roover, "Untangling composite commits using program slicing," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2018, pp. 193–202.

[27] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," *ACM Sigplan Notices*, vol. 19, no. 5, pp. 177–184, 1984.

[28] J. Arora, "Static program slicing-an efficient approach for prioritization of test cases for regression testing," *Int. Journal of Computer Applications*, vol. 135, no. 13, pp. 18–23, 2016.

[29] D. Ghosh and J. Singh, "A systematic review on program debugging techniques," *Smart Computing Paradigms: New Progresses and Challenges: Proceedings of ICACNI 2018, Volume 2*, pp. 193–199, 2019.

[30] M. Chalupa and J. Strejček, "Evaluation of program slicing in software verification," in *International Conference on Integrated Formal Methods*. Springer, 2019, pp. 101–119.

[31] J.-F. Bergeretti and B. A. Carré, "Information-flow and data-flow analysis of while-programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 1, pp. 37–61, 1985.

[32] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[33] S. Danicic, M. Harman, and Y. Sivagurunathan, "A parallel algorithm for static program slicing," *Information Processing Letters*, vol. 56, no. 6, pp. 307–313, 1995.

[34] I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon, "Program slicing based on specification," in *Proceedings of the 2001 ACM symposium on Applied computing*, 2001, pp. 605–609.

[35] J. Cheney, "Program slicing and data provenance." *IEEE Data Eng. Bull.*, vol. 30, no. 4, pp. 22–28, 2007.

[36] L. Bent, D. Atkinson, and W. Griswold, "A qualitative study of two whole-program slicers for c," *Technical Report*, 2000.

[37] D. Binkley and M. Harman, "A large-scale empirical study of forward and backward static slice size and context sensitivity," in *International Conference on Software Maintenance. ICSM 2003. Proceedings.* IEEE, 2003, pp. 44–53.

[38] D. W. Binkley and M. Harman, "A survey of empirical results on program slicing." *Adv. Comput.*, vol. 62, no. 105178, pp. 105–178, 2004.

[39] T. Hoffner, *Evaluation and comparison of program slicing tools*. Citeseer, 1995.

[40] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue, "Experimental evaluation of program slicing for fault localization," *Empirical Software Engineering*, vol. 7, no. 1, pp. 49–76, 2002.

[41] J. R. Lyle, "Evaluating variations on program slicing for debugging." *Dissertation Abstracts International Part B: Science and Engineering[DISS. ABST. INT. PT. B- SCI. & ENG.],*, vol. 46, no. 5, 1985.

[42] Y.-Z. Zhang, “Sympas: symbolic program slicing,” *Journal of Computer Science and Technology*, vol. 36, pp. 397–418, 2021.

[43] A. Jana, “Data-centric refinement of database-database dependency analysis of database program.” in *ICSOFT*, 2020, pp. 234–241.

[44] R. Halder, M. Zanioli, and A. Cortesi, “Information leakage analysis of database query languages,” in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014, pp. 813–820.

[45] C. Hammer and G. Snelting, “Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs,” *International Journal of Information Security*, vol. 8, no. 6, pp. 399–422, 2009.

[46] E. Baralis and J. Widom, “An algebraic approach to rule analysis in expert database systems,” Stanford, Tech. Rep., 1994.

[47] A. Jana, R. Halder, K. V. Abhishekh, S. D. Ganni, and A. Cortesi, “Extending abstract interpretation to dependency analysis of database applications,” *IEEE Transactions on Software Engineering*, vol. 46, no. 5, pp. 463–494, 2018.

[48] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[49] M. I. Alam and R. Halder, “Data-centric refinement of information flow analysis of database applications,” in *International Symposium on Security in Computing and Communication*. Springer, 2015, pp. 506–518.

[50] “Gotocode,” http://www.gotocode.com, [Online; accessed 20-Dec-2020], (now archived at: https://github.com/angshumanjana/GotoCode).

[51] E. Denney and B. Fischer, “Explaining verification conditions,” in *Int. Conference on Algebraic Methodology and Software Technology*. Springer, 2008, pp. 145–159.

[52] M. I. Alam, R. Halder, and J. S. Pinto, “A deductive reasoning approach for database applications using verification conditions,” *Journal of Systems and Software*, vol. 175, p. 110903, 2021.

Author Biographies

Anwesha Kashyap She received an M.Tech degree from the Indian Institute of Information Technology Guwahati, India, in 2021 and a B.Tech degree from Dibrugarh University Institute of Engineering and Technology, Dibrugarh, India, in 2019, both in the discipline of Computer Science and Engineering. She is currently a PhD Scholar in the Department of Computer Science and Engineering at the Indian Institute of Information Technology, Guwahati, India. Her current research interests include formal methods, program analysis and verification, information security.

Angshuman Jana He received a doctoral degree from the Indian Institute of Technology Patna, India, in 2019. He received a B.Tech degree from MAKAUT, India, in 2011 and an MTech degree from the National Institute of Technology Durgapur, India, in 2013, both in the discipline of computer science and engineering. He is currently working as an Assistant Professor in the Department of Computer Science and Engineering at the Indian Institute of Information Technology, Guwahati, India. His areas of research interest include program analysis and verification, formal methods, algorithm development, database languages, data security. He has several publications in reputed journals and conferences on program analysis and verification and information flow security analysis.