Data-centric Refinement of Database-Database Dependency Analysis of Database Program

Angshuman Jana
Indian Institute of Information Technology Guwahati, India

Keywords: Database Program, Structured Query Language, Program Dependency Graph, Refinement.

Abstract: Since the pioneer work by Ottenstein and Ottenstein, the notion of Program Dependency Graph (PDG) has attracted a wide variety of compelling applications in software engineering, e.g. program slicing, information flow security analysis, debugging, code-optimization, code-reuse, code-understanding, and many more. In order to exploit the power of dependency graph in solving problems related to relational database applications, Willmor et al. first proposed Database Oriented Program Dependency Graph (DOPDG), an extension of PDG by taking database statements and their dependencies further into consideration. However, the dependency information generated by the DOPDG construction algorithm is prone to imprecision due to its syntax-based computation, and therefore the approach may increase the susceptibility of false alarms in the above-mentioned application scenarios. Addressing this challenge, in this paper, the following two main research objectives are highlighted: (1) How to obtain more precise dependency information (hence more precise DOPDG)? and (2) How to compute them efficiently? To this aim, a data-centric based approach is proposed to compute precise dependency information by removing false alarms. To refine the database-database dependency, the syntax-based DOPDG construction is augmented by adding three extra nodes and edges (as per the condition-action execution sequence) with each node that represents the database statement.

1 INTRODUCTION

The database technology is always at the heart of any information systems, facilitating one to store external data into persistent storage and to process them efficiently (Goldin et al., 2004). Even in the era of big data, a survey by TDWI in 2013 (Russom, 2013) says that, for a quarter of organizations, more than 20% of large volume of data are structured in nature and are stored in the form of relational database. Due to the structured form of stored data, relational database management systems gain immense popularity among the database community. A most common way to develop a database application is to embed relational database languages such as SQL, PL/SQL, HQL, etc., into other host languages like C, C++, Java, etc.(Goldin et al., 2000; Date, 2006). Over the decades, database applications are playing a pivotal role in every aspect of our daily lives by providing an easy interface to store, access and process crucial data with the help of Relational Database Management System (RDBMS). Some examples of software systems where database applications act as an integral part include online shopping store, banking system, railway reservation system, even critical systems such as air traffic control, health care and so on.

In the software systems, the dependency information among program statements and variables, solving a large number of software engineering tasks security analysis (Hammer, 2010; Mandal et al., 2014; Ahuja et al., 2016), taint analysis (Krinke, 2007), program slicing (Tip, 1994; Jana et al., 2015), optimization (Ferrante et al., 1987; Bondhugula et al., 2008), code-reuse (Jiang, 2009), code-understanding (Podgurski and Clarke, 1990; Jana et al., 2018a). One most suitable representation of these dependencies is in the form of *Dependency Graph* that consists of both data- and control-dependencies among program components. The control-dependencies among statements are computed based on the syntactic structure of the program: a statement s_2 is said to be control dependent on another statement s_1 iff there exists a path p from s_1 to s_2 such that every statement $s_i \neq s_1$ within p will be followed by s_2 in every possible path to the end of the program, and there is an execution path from s_1 to the end of the program that does not go through s_2. Similarly, a way to compute data-dependencies is to consider syntactic presence of one variable in the

Jana, A.
Data-centric Refinement of Database-Database Dependency Analysis of Database Program.
DOI: 10.5220/0009891202340241
In *Proceedings of the 15th International Conference on Software Technologies (ICSOFT 2020)*, pages 234-241
ISBN: 978-989-758-443-5

definition of another variable: a statement s_2 is said to be data-dependent on another statement s_1 if there exists a variable x such that x is defined by s_1 and subsequently used by s_2, and there is a x-definition free path from s_1 to s_2 (Ottenstein and Ottenstein, 1984).

Since the pioneer work by Ottenstein and Ottenstein (Ottenstein and Ottenstein, 1984), the notion of Program Dependency Graph (PDG) has attracted a wide variety of compelling applications in software engineering, e.g. program slicing, information flow security analysis, debugging, code-optimization, code-reuse, code-understanding, and many more. Since its inception, a number of variants are also proposed for various programming languages and features, possibly tuning them towards their suitable application domains, like System Dependence Graph (SDG) (Horwitz et al., 1990), Class Dependence Graph (ClDG) (Larsen and Harrold, 1996) and etc. In order to exploit the power of dependency graph in solving problems related to relational database applications, Willmor et al. first proposed Database Oriented Program Dependency Graph (DOPDG), an extension of PDG by taking database statements and their dependencies further into consideration as (i) Program-Database dependency (PD-dependency) which represents dependency between an imperative statement and a database statement, and (ii) Database-Database dependency (DD-dependency) which represents a dependency between two database statements. However, the dependency information generated by the DOPDG construction algorithm is prone to imprecision due to its syntax-based computation, and therefore the approach may increase the susceptibility of false alarms.

To exemplify our motivation briefly, let us consider a small database code snippet below that consists three SQL statements Q_1, Q_2 and Q_3:

Q_1 : UPDATE emp SETsal:=sal+1000 WHERE sal $\leq$ 3000

Q_2 : UPDATE emp SETsal:=sal∗.2 WHERE sal $\geq$ 7000

Q_3 : SELECT MAX(sal) FROM emp WHERE sal $\geq$ 5000

The statement Q_3 is syntactically dependent on Q_1 and Q_2 for 'sal' because it is a used-variable in Q_3 and it is a defined-variable both in Q_1 and Q_2. Note that, the values of 'sal' in the database, the part of sal-values defined by Q_1 is not overlapping with the sal-values subsequently used by Q_3. Therefore, the dependency between Q_1 and Q_3 is false alarm.

Observe that syntax-based DOPDG construction approach may generate false dependencies. Generation of false dependency information and its use in any software-engineering activities, such as safety property verification of any critical systems, may enforce. Particularly, false dependency information reduces the system throughput, as a result, financial cost and resource utilization may be affected. Unfortunately, since then no significant contribution is found in this research direction. As the values of database attributes differ from that of imperative language variables, the computation of semantics (and hence semantics-based dependency) of database applications is, however, challenging and requires different treatment. The key point here is the static identification of various parts of the database information possibly accessed or manipulated by database statements at various program points.

Addressing this challenge, in this paper, I aim to answer the following two main research objectives: (1) How to obtain more precise dependency information (hence more precise DOPDG)? and (2) How to compute them efficiently? To this aim, I propose a data-centric based approach to compute precise dependency information by removing false alarms. To refine the database-database dependency, I augment the syntax-based DOPDG construction by adding three extra nodes along with edges (as per the condition-action execution sequence) with each node that represents the database statement. This propose approach serves an automatic tool to compute various dependencies information among variables and statements in database applications. This tool will also useful in future to solve many software-engineering problems, *e.g.* Database Code Slicing (Larsen and Harrold, 1996), Database Leakage Analysis (Halder et al., 2014), Data Provenance (Cheney et al., 2007), Materialization View Creation (Sen et al., 2012), Concurrent System modeling, etc.

Roadmap: In section 2, I discuss the current state-of-the-art in the literature. In section 3, I describe a running example. The propose approach is introduced in section 4. Section 5 provides an overall tool architecture. Finally section 6 concludes the work.

2 RELATED WORKS

In (Ottenstein and Ottenstein, 1984; Ferrante et al., 1987) authors introduced the notion of Program Dependency Graph (PDG) aiming program optimization. It is an intermediate representation of programs where nodes represent program statements and edges represent data- and control-dependencies between the statements. Over the past, PDG plays important roles in various software systems activities, e.g. program slicing (Tip, 1994), code-reuse (Jiang, 2009), language-based information flow security analysis

(Krinke, 2007; Hammer, 2010; Halder et al., 2014; Halder et al., 2016), code-understanding (Podgurski and Clarke, 1990). Since then, various extension and modification of PDG have been proposed towards many directions. Over the past several decades, various form of dependency graphs are evolved in different contexts for different programming languages, *e.g.* Program Dependence Graph (PDG) (Ottenstein and Ottenstein, 1984) in case of intra-procedural programs, System Dependence Graph (SDG) (Horwitz et al., 1990) in case of inter-procedural programs, Class Dependence Graph (ClDG) is introduced for Object Oriented Programming (OOP) languages in (Larsen and Harrold, 1996). Willmor et.al. (Willmor et al., 2004) introduced a variant of program dependency graph, known as Database-Oriented Program Dependency Graph (DOPDG), by considering the two additional data dependencies due to the presence of database statements: (*i*) Program-Database dependency (PD-dependency) which represents dependency between an imperative statement and a database statement, and (*ii*) Database-Database dependency (DD-dependency) which represents a dependency between two database statements.

All such proposed dependency graphs are constructed based on the syntactic presence of variable in the definitions of other variable. However, syntactic dependency computations may produce false alarms. As a notable achievement, (Mastroeni and Zanardini, 2008) introduced the notion of semantic-data dependency which focuses on the actual values of variable rather than their syntactic presence. For instance, although the expression "$e = x^2 + 4w \bmod 2 + z$" syntactically depends on w, semantically there is no dependency as the evaluation of "$4w \bmod 2$" is always zero. Therefore, syntax-based approach may fail to compute an optimal results. Another approach Condition-Action rule (Baralis and Widom, 1994) is also applicable for dependencies computation, in case of database applications, SQL statements define either a part of the values or all of the values corresponding to an attribute depending on the condition present in the WHERE clause. But this approach is unable to provided the optimal solution and suffer from high computational cost ($O(2^n)$ where n represent the number of variables in a program). In (Alam and Halder, 2016) authors proposed semantics-based DOPDG using weakest precondition and postcondition of Hoare Logic to address the information-flow analysis of database applications. But this approach lead to an exponential computational overhead and also unable to compute optimal result. (Halder and Cortesi, 2013; Jana et al., 2018b; Jana and Halder, 2016) formalized the semantics for dependency refinement in a simple setting following the Abstract Interpretation as an initial attempt. However this is also suffer form large number of false alarm. A semantic characterization of dependency provenance is proposed in (Cheney et al., 2007), where dependency provenance is intended to show how (part of) the out put of a query depended on (part of) its input. (Amtoft and Banerjee, 2007) defined a Hoare-style logic to analyze variable independency.

```
Start;
Q0: Connection c =DriverManager.getConnection(......);
Q1: UPDATE emp SET sal := sal + Sbonus WHERE age ⩾ 60;
Q2: SELECT AVG(sal) FROM emp WHERE  age ⩾ 60;
Q3: SELECT AVG(sal) FROM emp WHERE  age < 60
Q4: UPDATE emp SET sal := sal + Cbonus;
Q5: SELECT AVG(sal) FROM emp;
Stop;
```

Figure 1: A database code snippet Prog.

3 RUNNING EXAMPLE

let us consider a small database code snippet Prog, depicted in Figure 1, that enhance the salary of all employees in any organization by the common bonus amount *Cbonus* and by the additional special bonus amount *Sbonus* only for aged employees. Note that, the syntactic presence of attribute *'sal'* as a defined-attribute at statement Q_1 and as an used-attribute at statement Q_3. Therefore, the statement Q_3 is syntactically dependent on statement Q_1. However, a careful observation reveals that syntactic presence of database attribute as a way of database database dependency computation may often result in false alarm, and thus fails to generate precise set of dependencies. For example, if any one focus on the value of the attribute *'sal'* in the code that the values of *'sal'* referred in the "`WHERE`" clauses at statements Q_1 and Q_3 do not overlap with each other. Hence the statement Q_3 does not dependent on statement Q_1. I show, in the subsequent sections, how the propose approach effectively identifies false DD-dependencies in Prog.

4 PROPOSED APPROACH

In this section, I describe a novel approach, how to refine the syntactic DOPDG for gaining the precise Database-Database (DD) dependency information among the statements of a database program. At first, I recall from (Willmor et al., 2004) the syntax-based DOPDG construction. In the next step, the syntactic DOPDG is augmented by adding three extra

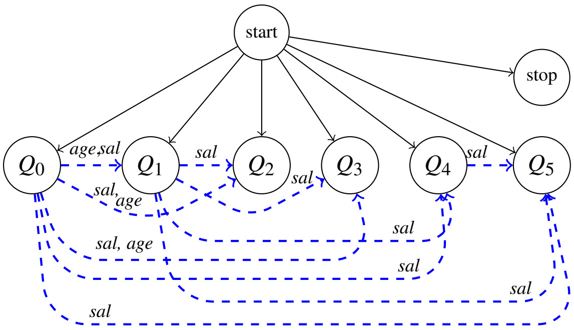

Figure 2: Syntax-based DOPDG of Prog.

nodes along with edges (as per the condition-action execution sequence) with each node that represents the database statement. Finally, based on the augmented DOPDG ψ the *used* and *defined*-parts of the database is calculated and their overlapping information refine the DD-dependency.

4.1 Syntax-based DOPDG

Database-Oriented Program Dependency Graph (DOPDG) (Willmor et al., 2004) is an extension of PDG to the case of database programs. DOPDG considers two additional dependencies: (*i*) Program-Database dependency and (*ii*) Database-Database dependency. A PD-dependency represents the dependency between a database statement and an imperative statement, whereas a DD-dependency represents the dependency between two database statements. Let us recall them below:

Definition 1 (Program-Database (PD) Dependency (Willmor et al., 2004))**.** *A database statement Q is PD dependent on an imperative statement S for a variable k (denoted* $S \xrightarrow{K} Q$*) if the below three hold: (i) k is defined by S, (ii) k is used by Q, and (iii) there is no redefinition of k between S and Q.*

The PD-dependency of *S* on *Q* is defined similarly.

Definition 2 (Database-Database (DD) Dependency (Willmor et al., 2004))**.** *Let Q.SE, Q.IN, Q.UP and Q.DE represent the operations on database which are select, insert, update, and delete respectively by statement Q. A database statement* Q_1 *is DD-dependent on another database statement* Q_2 *for an attribute a (denoted* $Q_1 \xrightarrow{a} Q_2$*) if the following hold: (i)* $Q_1.SEL \cap (Q_2.INS \cup Q_2.UPD \cup Q_2.DEL) \neq \emptyset$*, and (ii) there is no roll-back operation in the execution path p between* Q_2 *and* Q_1 *(exclusive) which reverses back the effect of* Q_2*.*

Example 1. *Consider the running example Prog depicted in Figures 1 (section 3). The control dependencies Start* $\rightarrow Q_1$*, Start* $\rightarrow Q_2$*, etc. are computed in similar way as in the case of traditional PDG. The used and defined attributes at each program point of Prog are computed as follows:*

$DEF(Q_0)=\{sal, Sbonus, Cbonus, age\}$
$DEF(Q_1)=\{sal\}$ $USE(Q_1) = \{Sbonus, age\}$
$DEF(Q_2)=\{\emptyset\}$ $USE(Q_2) = \{sal, age\}$
$DEF(Q_3)=\{\emptyset\}$ $USE(Q_3) =\{sal, age\}$
$DEF(Q_4)=\{sal\}$ $USE(Q_4) = \{Cbonus\}$
$DEF(Q_5)=\{\emptyset\}$ $USE(Q_5) = \{sal\}$

Observe that statement Q_0 *defines all database attributes as it connects to the database, resulting* $DEF(Q_0)$ *to contain all attributes. From the above information, the following data dependencies are identified:*

- *DD-dependencies for attributes sal and age:* $\{Q_0 \rightarrow Q_1, Q_2, Q_3, Q_4, Q_5\}$, $\{Q_1 \rightarrow Q_2, Q_3, Q_4, Q_5\}$ *and* $\{Q_4 \rightarrow Q_5\}$*,*

The syntax-based DOPDG construction of Prog is depicted in Figure 2.

4.2 Augmentation of DOPDG

In this section, the syntax-based DOPDG is augmented by adding extra nodes and edges (according to condition and action present in a database statement) with the node that represent the database statement. At first step, I identify the set of database statements (Select, Insert, Update and Delete) in a database program and mark (may used any color) the corresponding nodes in the DOPDG. In particular, the presence of Data Manipulation Language (DML) statements in a database program is identified based on the presence of keywords such as `SELECT`, `UPDATE`, `DELETE` and `INSERT` in the database statements.

As per the execution sequence, I divide each database statement with two part: one is condition-part and another one is action-part. Formally, a SQL statement Q is denoted by $\langle A, \phi \rangle$ where A represents an action-part and ϕ represents a conditional-part. The action-part A includes `SELECT`, `UPDATE`, `DELETE` and `INSERT` operations which are denoted by A_{sel}, A_{upd}, A_{del} and A_{ins} respectively. The conditional-part ϕ represents the condition under the `WHERE` clause of the statement, which follows first-order logic formula. For instance, the query Q = “`UPDATE emp SET` *sal*:=*sal*+`100` `WHERE` *age* ⩾`40`” is denoted by $Q = \langle A_{upd}, \phi \rangle$ where A_{upd} represents “ *sal*:=*sal*+`100`” and ϕ represents “*age* ⩾`40`”.

Now, each marked node of the syntax-based DOPDG is augmented by three extra nodes and edges where each node and edge are labeled by $\langle \phi, \Delta_{\phi} \rangle$, $\langle \neg\phi, \Delta_{\neg\phi} \rangle$ and $\langle A, \Delta_A \rangle$ respectively. The Δ_{ϕ} represents

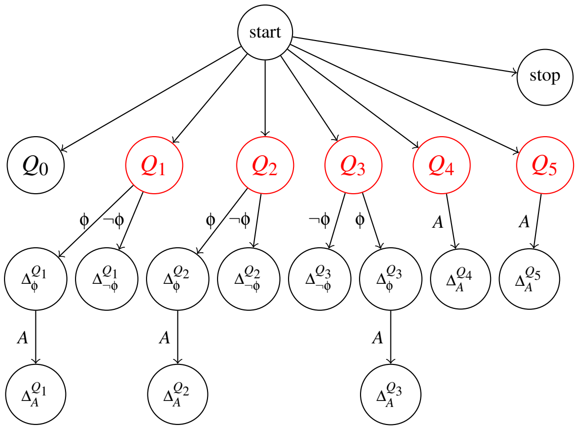

Figure 3: Augmentation of the DOPDG of Prog.

a part of the database which satisfies ϕ, whereas the $\Delta_{\neg\phi}$ represents a part of database which does not satisfy ϕ. The Δ_A is obtained after performing an action A on Δ_{ϕ}. Observe that, the ϕ, $\neg\phi$ and A are labeled with the edges of the corresponding nodes.

Example 2. *Let us consider the database program Prog in Figure 1. The augmented DOPDG of Prog is depicted in Figure 3. Observe that, in Prog the set database statements are* Q_1, Q_2, Q_3, Q_4 *and* Q_5 *and their corresponding nodes are marked by red color in the augmented DOPDG. Now,* Q_1 *is represented by* $\langle A_{upd}, \phi\rangle$ *where* A_{upd} *represents "* `sal:=sal+Sbonus`*" and* ϕ *represents "age* $\geqslant$`60`*". Therefore, in the augmented DOPDG, node* $\Delta_{\phi}^{Q_1}$ *represent the part of the database which satisfies age* $\geqslant$`60`*, the node* $\Delta_{\neg\phi}^{Q_1}$ *represent the part of the database which satisfies* $\neq$`(`*age* $\geqslant$`60)` *and the node* $\Delta_{A}^{Q_1}$ *which obtained after performing* `sal:=sal+Sbonus` *and their associated edges are added with node* Q_1 *of the DOPDG. Similarly nodes and edges are added with the nodes* Q_2, Q_3, Q_4 *and* Q_5 *of the DOPDG. Note that, in the case of* Q_4 *and* Q_5*, the* ϕ *is empty. Therefore, only nodes* $\Delta_{A}^{Q_4}$ *and* $\Delta_{A}^{Q_5}$ *along with the connected edges are added with node* Q_4 *and* Q_5 *respectively.*

4.3 Dependency Computations

Now I compute the DD-dependencies among database statements. From the augmented DOPDG ψ, I compute the set of *used*- and *defined*-parts of the database w.r.t. database statements.

Given two database statements Q_1 and Q_2. The *defined*-part by Q_1 and the *used*-part by Q_2 are :

$$\mathsf{E}^{Q_1} = \mathbf{D}_{def}(Q_1, \psi) = \langle \Delta_{\phi}^{Q_1}, \Delta_{A}^{Q_1} \rangle$$

$$\mathsf{U}^{Q_2} = \mathbf{D}_{use}(Q_2, \psi) = \langle \Delta_{\phi}^{Q_2} \rangle$$

The semantic dependency and independency of Q_2 on Q_1 are determined based on the following four cases:

$$\mathbf{Case-1}.\ \Delta_{\phi}^{Q_1} \cap \Delta_{\phi}^{Q_2} \neq \emptyset \wedge \Delta_{A}^{Q_1} \cap \Delta_{\phi}^{Q_2} = \emptyset$$

$$\mathbf{Case-2}.\ \Delta_{\phi}^{Q_1} \cap \Delta_{\phi}^{Q_2} = \emptyset \wedge \Delta_{A}^{Q_1} \cap \Delta_{\phi}^{Q_2} \neq \emptyset$$

$$\mathbf{Case-3}.\ \Delta_{\phi}^{Q_1} \cap \Delta_{\phi}^{Q_2} \neq \emptyset \wedge \Delta_{A}^{Q_1} \cap \Delta_{\phi}^{Q_2} \neq \emptyset$$

$$\mathbf{Case-4}.\ \Delta_{\phi}^{Q_1} \cap \Delta_{\phi}^{Q_2} = \emptyset \wedge \Delta_{A}^{Q_1} \cap \Delta_{\phi}^{Q_2} = \emptyset$$

Therefore, Q_2 is DD-Independent on Q_1 if and only if $\mathsf{E}^{Q_1} \cap \mathsf{U}^{Q_2} = \emptyset$; that is $\Delta_{\phi}^{Q_1} \cap \Delta_{\phi}^{Q_2} = \emptyset \wedge \Delta_{A}^{Q_1} \cap \Delta_{\phi}^{Q_2} = \emptyset$. This pictorial representation of the above cases are depicted in Figure 4.

Algorithm to Compute DD-dependency based on *used* and *defined* Information The algorithm **DDA** takes a list of *used*- and *defined*-parts ($\mathbf{D}_{use}$ and $\mathbf{D}_{def}$) at each program point c_i of the database program of size n, and generates refine DOPDG. The algorithm remove edges (false alarm) between DOPDG-nodes c_i and c_j based on the emptiness checking of the intersection of the *defined*-part by c_i and the *used*-part by c_j. To remove false dependency where more than one database statements (in sequence) redefine an attribute values which is finally used by another statement, the condition $\mathbf{D}_{def}(i) \subseteq \mathbf{D}_{def}(j)$ verifies whether *defined*-part at program point c_i is fully covered by the *defined*-part at program point c_j. In this case, integer variable f represents the '1' value which indicate the dependency between c_i and c_j.

Algorithm 1: DDA.

```
Input: used- and defined-part ($\mathbf{D}_{use}$, $\mathbf{D}_{def}$) by all
       database statements in the program.
Output: Refine DOPDG
Set flag=TRUE
for $i$ =1 to n-1 do
    for $j$=i+1 to n do
        if $D_{def}(i) \cap D_{use}(j)= \emptyset$ then
            int f = 0;
            Remove the edge from $i^{th}$ node to $j^{th}$
              node ($i \rightarrow j$);
            Report this as a false alarm;
        else
            ...Do nothing...
        if flag=True then
            if $D_{def}$ (i) $\subseteq$ $D_{def}$ (j) then
                f = 1;
                Break;
End
```

Illustration on Running Example: Now I illustrate the DD-dependency refinement on the running example Prog in Figure 1 (section 3). The DD-

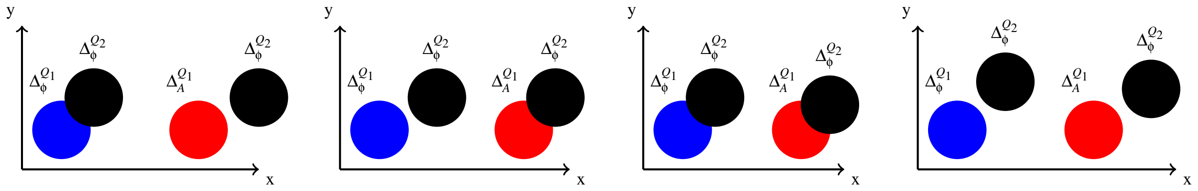

Figure 4: Representations of 4 Cases: from **Case-1** to **Case-4** (from left to right).

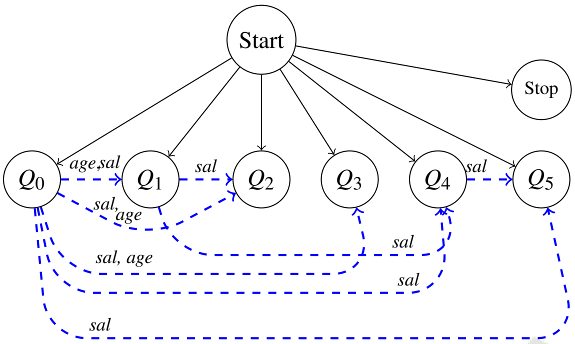

Figure 5: Refine DOPDG of Prog.

dependency refinements are computed applying the following steps:

- Compute *defined*- and *used*-parts based on the ψ.
- Refinement of syntactic dependencies in "`Prog`" using Algorithm 1.

By removing two false dependencies $Q_1 \rightarrow Q_3$ and $Q_1 \rightarrow Q_5$, the refine DOPDG of Prog is depicted in Figure 5. Observe that, as $\Delta_\phi^{Q_1} \cap \Delta_\phi^{Q_3} = \emptyset \wedge \Delta_A^{Q_1} \cap \Delta_\phi^{Q_3} = \emptyset$ the dependency $Q_1 \rightarrow Q_3$ is removed (false alarm). Similarly, the dependency $Q_1 \rightarrow Q_5$ is removed (false alarm) as the part of *sal*-values defined by Q_1 is fully redefined by Q_4 and never reaches Q_5.

5 TOWARDS IMPLEMENTATION

I design a tool Database-Database Dependency Analyzer (D3A) based on the proposed framework. In general, the D3A accepts as inputs a database program and computes more precise set of Database-Database (DD) dependency information among the statements as output. The tool D3A consists of two major components: (i) Syntax-based module, and (ii) Refinement module. Figure 6 depicts the overall architecture of the tool. This two components also consist the following key modules which play important roles in implementing the proposed framework:

- **Proformat:** The module "Proformat" annotates input database programs and adds line numbers (starting from zero) to all statements.

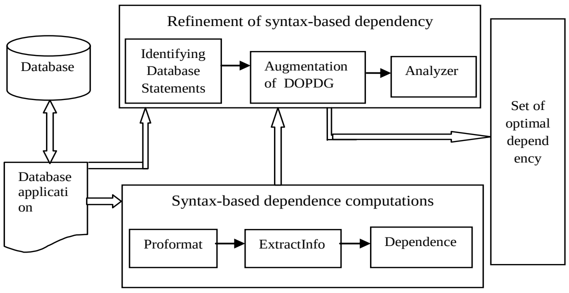

Figure 6: Overall Architecture of Database-Database Dependency Analyzer (D3A).

- **ExtractInfo:** This module extracts detail information about input programs, i.e. control statements, defined variables, used variables, etc. for all statements in the program.
- **Dependency:** The "Dependency" module computes syntax-based dependencies among program statements using the information computed by "ExtractInfo" module.
- **Identifying Database Statements:** This module computes the number of SQL statements present in the database program. In particular, the presence of Data Manipulation Language (DML) statements is identified based on the presence of keywords such as `SELECT`, `UPDATE`, `DELETE` and `INSERT` in the statements.
- **Augmentation of DOPDG:** The module augments the syntax-based DOPDG construction, by adding three extra nodes and edges (based on the condition-action execution sequence) with each node that represents the database statement.
- **Analyzer:** Finally this module identifies false dependency (if any) based on the *used* (as per condition of a statement) and *defined* (as per action of a statement) nodes of augmented DOPDG and their overlapping.

6 CONCLUSIONS AND FUTURE WORKS

In this paper, I proposed data-centric based approach to compute precise dependency information (by removing false alarms) among the database statement of a database application. To refine the syntax-based DD-dependency information (may exist false alarm), I design a Database-Database Dependency Analyzer (D3A) based on the following key modules: (i) Identifying database statements, (ii) Augmentation of syntax-based DOPDG and (iii) Analyzer. Currently, I am implementing the proposed tool D3A, as per the description provided in the tool architecture, in a modular way to support scalability. In future, this tool will be used to address efficiently several software engineering problems like Database Code Slicing (Larsen and Harrold, 1996), Database Leakage Analysis (Halder et al., 2014), Data Provenance (Cheney et al., 2007), Materialization View Creation (Sen et al., 2012), Concurrent System modeling, etc.

REFERENCES

Ahuja, B. K., Jana, A., Swarnkar, A., and Halder, R. (2016). On preventing sql injection attacks. In *Advanced Computing and Systems for Security*, pages 49–64. Springer.

Alam, M. I. and Halder, R. (2016). Refining Dependencies for Information Flow Analysis of Database Applications. In *International Journal of Trust Management in Computing and Communications*. Inderscience.

Amtoft, T. and Banerjee, A. (2007). A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Sci. Comput. Program.*, 64(1):3–28.

Baralis, E. and Widom, J. (1994). An Algebraic Approach to Rule Analysis in Expert Database Systems. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 475–486. Morgan Kaufmann Publishers Inc.

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. (2008). PLUTO: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*.

Cheney, J., Ahmed, A., and Acar, U. A. (2007). Provenance As Dependency Analysis. In *Proceedings of the 11th ICDPL*, DBPL'07, pages 138–152.

Date, C. J. (2006). *An introduction to database systems*. Pearson Education India.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Trans. on Programming Lang. and Sys.*, 9(3):319–349.

Goldin, D., Srinivasa, S., and Srikanti, V. (2004). Active databases as information systems. In *Database Engineering and Applications Symposium, 2004. IDEAS'04. Proceedings. International*, pages 123–130. IEEE.

Goldin, D., Srinivasa, S., and Thalheim, B. (2000). Is=dbs + interaction: towards principles of information system design. In *International Conference on Conceptual Modeling*, pages 140–153. Springer.

Halder, R. and Cortesi, A. (2013). Abstract Program Slicing of Database Query Languages. In *Proceedings of the the 28th Symposium On Applied Computing - Special Track on Database Theory, Technology, and Applications*, pages 838–845, Coimbra, Portugal. ACM Press.

Halder, R., Jana, A., and Cortesi, A. (2016). Data leakage analysis of the hibernate query language on a propositional formulae domain. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXIII*, pages 23–44. Springer.

Halder, R., Zanioli, M., and Cortesi, A. (2014). Information leakage analysis of database query languages. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC'14)*, pages 813–820, Gyeongju, Korea. ACM Press.

Hammer, C. (2010). Experiences with PDG-Based IFC. In *Proc. of the Engineering Secure Software and Systems*, pages 44–60, Pisa, Italy. Springer-Verlag.

Horwitz, S., Reps, T., and Binkley, D. (1990). Interprocedural slicing using dependence graphs. *ACM Transactions on PLS*, 12(1):26–60.

Jana, A., Alam, M. I., and Halder, R. (2018a). A symbolic model checker for database programs. In *ICSOFT*, pages 381–388.

Jana, A. and Halder, R. (2016). Defining abstract semantics for static dependence analysis of relational database applications. In *International Conference on Information Systems Security*, pages 151–171. Springer.

Jana, A., Halder, R., Chaki, N., and Cortesi, A. (2015). Policy-based slicing of hibernate query language. In *IFIP International Conference on Computer Information Systems and Industrial Management*, pages 267–281. Springer.

Jana, A., Halder, R., Kalahasti, A., Ganni, S., and Cortesi, A. (2018b). Extending abstract interpretation to dependency analysis of database applications. *IEEE Transactions on Software Engineering*.

Jiang, L. (2009). *Scalable Detection of Similar Code: Techniques and Applications*. PhD thesis, Davis, CA, USA.

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

Larsen, L. and Harrold, M. J. (1996). Slicing object-oriented software. In *Proceedings of the 18th ICSE*, pages 495–505, Berlin, Germany. IEEE CS.

Mandal, K. K., Jana, A., and Agarwal, V. (2014). A new approach of text steganography based on mathematical model of number system. In *2014 International*

Conference on Circuits, Power and Computing Technologies [ICCPCT-2014], pages 1737–1741. IEEE.

Mastroeni, I. and Zanardini, D. (2008). Data dependencies and program slicing: from syntax to abstract semantics. In *Proc. of the Partial evaluation and semantics-based program manipulation*.

Ottenstein, K. J. and Ottenstein, L. M. (1984). The program dependence graph in a software development environment. *ACM SIGPLAN Notices*, 19(5):177–184.

Podgurski, A. and Clarke, L. A. (1990). A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. on SE*, 16(9):965–979.

Russom, P. (2013). Managing big data. *TDWI Best Practices Report, TDWI Research*, pages 1–40.

Sen, S., Dutta, A., Cortesi, A., and Chaki, N. (2012). A New Scale for Attribute Dependency in Large Database Systems. In *CISIM*, volume 7564 of *LNCS*, pages 266–277.

Tip, F. (1994). A Survey of Program Slicing Techniques. Technical report.

Willmor, D., Embury, S. M., and Shao, J. (2004). Program slicing in the presence of a database state. In *Proc. of the IEEE ICSM*.