Defining Abstract Semantics for Static Dependence Analysis of Relational Database Applications

Angshuman Jana[1] and Raju Halder[1,2]

[1] Indian Institute of Technology Patna, India,
{ajana.pcs13, halder}@iitp.ac.in
[2] HASLab, INESC TEC, Braga, Portugal,
raju.halder@inesctec.pt

Abstract. Dependence Graph provides the basis for powerful programming tools to address a large number of software engineering activities including security analysis. This paper proposes a semantics-based static dependence analysis framework for relational database applications based on the Abstract Interpretation theory. As database attributes differ from traditional imperative language variables, we define abstract semantics of database applications in relational abstract domain. This allows to identify statically various parts of database information (in abstract form) possibly used or defined by database statements, leading to a more precise dependence analysis. This way the semantics-based dependence computation improves *w.r.t.* its syntax-based counterpart. We prove the soundness of our proposed approach which guarantees that non-overlapping of the defined-part by one statement and the used-part by another statement in abstract domain always indicates a non-dependency in practice. Furthermore, the abstract semantics as a basis of the proposed framework makes it more powerful to solve undecidable scenario when initial database state is completely unknown.

Key words: Dependence Graph, Database Application, Static Analysis, Security

1 Introduction

Dependence Graph is an intermediate representation of program which explicits both the data- and control-dependences among program statements. This provides the basis for powerful programming tool to address a large number of software engineering activities, *e.g.* language-based information flow security analysis, safety verification, optimization, maintenance, code-understanding, code-reuse, etc. [10, 14–17, 20, 29, 30].

Different variants of Dependence Graph, *e.g.* Program Dependence Graph (PDG) [29], System Dependence Graph (SDG) [16], Class Dependence Graph (ClDG) [22], Database-Oriented Program Dependence Graph (DOPDG) [33], etc. are proposed in different contexts for different programming languages tuning them towards specific applications.

Syntax-based construction of Dependence Graph depends on the computation of (*i*) data-dependences based on the syntactic presence of one variable

in the definition of another variable and (*ii*) control-dependences based on the syntactic structure of the program [29].

Mastroeni and Zanardini [24] first observed that the syntax-based approach may fail to compute an optimal set of dependences where the syntactic presence of variables is not enough to represent relevancy. For instance, consider an expression "$e = x + 2 \times w \; mod \; 2$" where e is syntactically dependent on w but semantically there is no such dependence. Computation of such false dependences which focuses on values instead of variables, allows us to refine syntax-based dependence graph into more refined semantics-based dependence graph.

Willmor et al. [33] introduced the notion of Database-Oriented Program Dependence Graph (DOPDG), an extension of traditional PDG, to the context of database applications embedding query languages. DOPDG considers two additional dependences: Program-Database Dependences (PD-Dependences) and Database-Database Dependences (DD-dependences). A PD-Dependence represents the dependence between a SQL statement and an imperative statement, whereas a DD-dependence represents the dependence between two SQL statements.

In this paper, we extend the notion of semantics-based dependences to the case of database applications, leading to a refinement of DOPDGs. For example, consider the following SQL statements Q_1 and Q_2:

$$Q_1 : \texttt{UPDATE emp SET } age := age + 1 \texttt{ WHERE } age \geqslant 35 \texttt{ AND } sal \leqslant 2000$$
$$Q_2 : \texttt{SELECT MAX}(sal), \texttt{ AVG}(age) \texttt{ FROM emp WHERE } age \leqslant 30$$

The statement Q_2 is syntactically DD-Dependent on Q_1 for the attribute *age* as it is a defined-variable in Q_1 and a used-variable in Q_2. But observe that, if we focus on the values of *age* in the database, the part of *age*-values defined by Q_1 is not overlapping with the *age*-values subsequently used by Q_2. Therefore, there exist no semantics-based dependence between Q_1 and Q_2. Some of the worth-mentioning software engineering activities where semantics-based dependences in database applications play crucial roles are program slicing, language-based information-flow analyses, data-provenance, security analyses like SQL injection attack, etc. [1, 5, 22, 30, 32].

The semantics-based data-dependence computation in query languages needs a different treatment as the values of database attributes differ from that of imperative language variables. 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. Abstract Interpretation [7, 8] is a widely used formal method which provides a sound approximation of program's semantics to answer about program's runtime behavior including undecidable ones.

This paper proposes a novel approach to compute semantics-based dependences among statements in database applications based on the Abstract Interpretation framework [8]. In particular, our contributions in this paper are:

- We define an abstract semantics of database statements in the relational domain of polyhedra based on the Abstract Interpretation framework.

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 $op_b \in \{+, -, *, /, \ldots...\}$
$k : \mathbb{S}$ (String)	$b ::= e_1\ op_r\ e_2 \mid \neg b \mid b_1 \vee b_2 \mid b_1 \wedge b_2 \mid true \mid false$, where $op_r \in \{=, \geq, \leq, <, \ldots\}$
$v_a : \mathbb{V}_a$ (Application Variables)	$g(\vec{e})$ ::= `GROUP BY`$(\vec{e}) \mid id$
$v_d : \mathbb{V}_d$ (Database Attributes)	$r ::=$ `DISTINCT` $\mid$ `ALL`
$e : \mathbb{E}$ (Arithmetic Expressions)	$s ::=$ `AVG` $\mid$ `SUM` $\mid$ `MAX` $\mid$ `MIN` $\mid$ `COUNT`
$b : \mathbb{B}$ (Boolean Expressions)	$h(e) ::= s \circ r(e) \mid$ `DISTINCT`$(e) \mid id$
$A : \mathbb{A}$ (Action)	$h(*) ::=$ `COUNT`$(*)$
$\tau : \mathbb{T}$ (Terms)	$\vec{h}(\vec{x}) ::= \langle h_1(x_1), ..., h_n(x_n)\rangle$, where $\vec{h} = \langle h_1, ..., h_n\rangle$ and $\vec{x} = \langle x_1, ..., x_n\rangle$
$a_f : \mathbb{A}_f$ (Atomic Formulas)	$f(\vec{e}) ::=$ `ORDER BY ASC`$(\vec{e}) \mid$ `ORDER BY DESC`$(\vec{e}) \mid id$
$\phi : \mathbb{W}$ (Pre-condition)	$Q ::= \langle A, \phi\rangle$
$Q : \mathbb{Q}$ (SQL statements)	$A ::=$ `SELECT`$(v_a,\ f(\vec{e}),\ r(\vec{h}(\vec{x})),\ \phi,\ g(\vec{e})) \mid$ `UPDATE`$(\vec{v_d},\ \vec{e}) \mid$ `INSERT`$(\vec{v_d},\ \vec{e}) \mid$ `DELETE`$(\vec{v_d})$
$I : \mathbb{I}$ (Imperative statements)	$\tau ::= n \mid k \mid v_a \mid v_d \mid f_n(\tau_1, \tau_2, ..., \tau_n)$, where f_n is an n-ary function.
$c : \mathbb{C}$ (Statements)	$a_f ::= R_n(\tau_1, \tau_2, ..., \tau_n) \mid \tau_1 = \tau_2$, where $R_n(\tau_1, \tau_2, ..., \tau_n) \in \{true, false\}$
	$\phi ::= a_f \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \forall x_i\ \phi \mid \exists x_i\ \phi$
	$I ::= skip \mid v := e$
	$c ::= Q \mid I \mid$ if b then c_1 else $c_2 \mid$ while b do c

Table 1: Syntax of query languages

- We develop an algorithm based on the data-flow analysis to compute abstract database states (in the form of polyhedra) at each program point of the applications.
- We propose an algorithm to compute non-overlapping of *used*- and *defined*-part by various database statements and hence non-dependences among them based on the abstract semantics.
- Finally, we provide an in-depth comparative analysis of our approach *w.r.t.* some existing notable directions in the literature.

The structure of the rest of paper is as follow: Section 2 recalls the syntax and concrete semantics of query languages. Section 3 recalls the notion of DOPDG and provides the basis on its syntax and semantics-based constructions. In section 4, we define abstract semantics of database statements in the domain of polyhedra based on the Abstract Interpretation theory and we propose a refinement of DOPDGs into more precise form. Section 5 discusses a detail comparisons of our proposal *w.r.t.* the literature and an overall complexity analysis. In section 6, we mention several applicative scenarios of our approach. Section 7 concludes the work.

2 Concrete Semantics of Database Query Languages

In this section, we recall from [12] the formulation of the semantics of database query languages.

Syntax. Table 1 depicts the syntactic sets and the abstract syntax of database statements in Backus-Naur form. Database applications involve two types of variables: application variables (denoted $\mathbb{V}_a$) and database attributes (denoted $\mathbb{V}_d$). The SQL clauses `GROUP BY`, `ORDER BY`, `DISTINCT/ALL` and the aggregate functions (`SUM, COUNT, MAX, MIN, AVG`) are represented in the form of functions $g()$, $f()$, $r()$, $h()$ respectively parameterized with either none or one arithmetic expression e or an ordered sequence of arithmetaic expressions $\vec{e}$. The

abstract syntax of a database statement is denoted by $\langle A, \phi \rangle$ where A represents Action-part and ϕ represents Condition-part which follows first-order logic formula. The Action-part include `SELECT, UPDATE, DELETE and INSERT`. For example, consider the query `Q`="`UPDATE emp SET` $sal{:=}sal{+}100$ `WHERE` $age \geqslant 40$". According to abstract syntax, `Q` is denoted by $\langle A,\ \phi \rangle = \langle \texttt{UPDATE}(\vec{v_d}, \vec{e}),\ \phi \rangle$, where $\vec{v_d} = \langle sal \rangle$ and $\vec{e} = \langle sal + 100 \rangle$ and $\phi{=}age \geqslant 40$.

Concrete Semantics. An application environment $\rho_a \in (\mathfrak{E}_a = \mathbb{V}_a \mapsto \texttt{Val})$ maps application variables to the domain of values `Val`. A database is a set of tables $\{t_i \mid i \in I_x\}$ for a given set of indexes I_x. A database environment is defined as a function ρ_d whose domain is I_x, such that for $i \in I_x$, $\rho_d(i) = t_i$. A table environment ρ_t for a table t is defined as a function such that $\forall a_i \in attribute(t)$, $\rho_t(a_i){=}\langle \pi_i(l_j) \mid l_j \in t \rangle$ where π is the projection operator, i.e., $\pi_i(l_j)$ is the i^{th} element of the l_j-th row.

The set of states is defined as $\Sigma = \mathfrak{E}_d \times \mathfrak{E}_a$ where $\mathfrak{E}_d$, $\mathfrak{E}_a$ are the set of database environments and application environments respectively. Therefore, a state $\rho \in \Sigma$ is denoted by a tuple $\rho = (\rho_d, \rho_a)$ where $\rho_d \in \mathfrak{E}_d$ and $\rho_a \in \mathfrak{E}_a$.

The state transition semantics is defined as $\mathbf{S}$: $\mathbb{Q} \times \Sigma \to \Sigma$, which specifies how the execution of a database statement $Q \in \mathbb{Q}$ on a state $\rho \in \Sigma$ results into an another state $\rho' \in \Sigma$.

3 Syntax-based DOPDG Vs. Semantics-based DOPDG

As already mentioned before, the syntax-based dependence computation depends on the syntactic presence of one variable in the definition of another variable or on the syntactic structure of the program, whereas the semantics-based dependence computation focuses on values rather than variables. This way, semantics-based analyses remove a number of false dependences and result into an optimal set of dependences.

In this section, we first discuss the syntax-based DOPDG construction briefly and then we provide the basis of semantics-based DOPDG.

3.1 Syntax-based DOPDG

Database-Oriented Program Dependence Graph (DOPDG) [33] is an extension to the traditional Program Dependence Graph (PDG) to represent dependences in database query languages. It considers the following two additional dependences:

Definition 1 (Program-Database (PD) Dependence). *A database statement Q is PD-dependent on an imperative statement I for a variable x (denoted $I \xrightarrow{x} Q$) if the following three hold: (i) x is defined by I, (ii) x is used by Q, and (iii) there is no redefinition of x between I and Q.*

The PD-dependence of I on Q is defined similarly.

Definition 2 (Database-Database (DD) Dependences). *A database statement Q_2 is DD-dependent on another database statement Q_1 for an attribute x (denoted $Q_1 \xrightarrow{x} Q_2$) if the following conditions hold: (i) x is defined by Q_1, (ii) x is used by Q_2, and (iii) there is no rollback effect of Q_1 in between them.*

Observe that the above definitions are based on the syntactic presence of "*used*" and "*defined*" variables in the statements. Therefore, syntax-based construction of DOPDG can be formalized based on the following two functions: `USE`: $\mathbb{C} \rightarrow \wp(\mathbb{V}_d \cup \mathbb{V}_a)$ and `DEF`: $\mathbb{C} \rightarrow \wp(\mathbb{V}_d \cup \mathbb{V}_a)$ which extract the set of variables (application variables and database attributes) *used* and *defined* in a statement $c \in \mathbb{C}$ (either imperative or database statement) respectively. Once the *used* and *defined* variables are computed for the program statements, the syntax-based dependences are determined according to Definitions 1 and 2. This is illustrated in example 1.

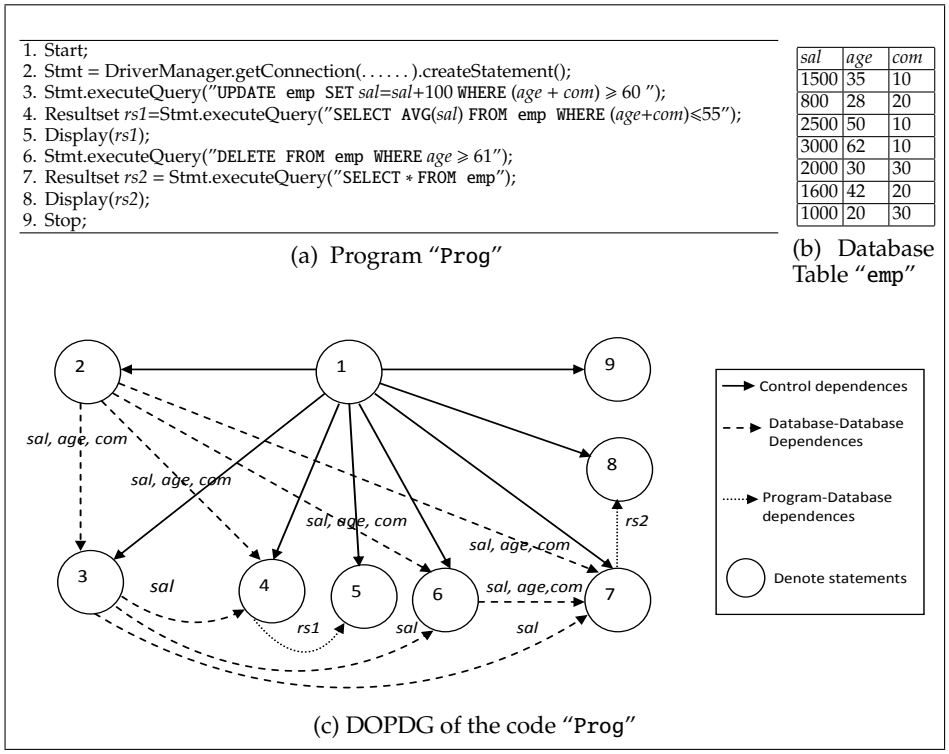

Fig. 1: A running example and its syntax-based DOPDG

Example 1. Consider the database application "`Prog`" and the associated database "`emp`" depicted in Figure 1. The syntax-based DOPDG of `Prog` is depicted in Figure 1(c). The control-dependences between program statements are computed by following similar approach as in the case of traditional Program Dependence Graphs. For instance, the edges $1 \rightarrow 2$, $1 \rightarrow 3$, $1 \rightarrow 4$, etc. represent

control dependencies. To obtain DD- and PD-dependencies, we extract *defined*- and *used*-variables at each program point using DEF and USE functions as follows:

DEF(2) = { *sal*, *age*, *com* }	DEF(3) = { *sal* }	USE(3) = { *sal*, *age*, *com* }
USE(4) = { *sal*, *age*, *com* }	USE(5) = { *rs1* }	DEF(6) = { *sal*, *age*, *com* }
USE(6) = { *sal*, *age*, *com* }	USE(7) = { *sal*, *age*, *com* }	USE(8) = { *rs2* }

Using above information one can easily compute DD- and PD-dependencies. For instance, edges $2 \rightarrow 3$, $2 \rightarrow 4$, $2 \rightarrow 6$, $2 \rightarrow 7$, $3 \rightarrow 4$, $3 \rightarrow 6$, $3 \rightarrow 7$ and $6 \rightarrow 7$ represent DD-dependencies (denoted by dashed-lines), whereas edges $4 \rightarrow 5$ and $7 \rightarrow 8$ represent PD-dependency (denoted by dotted-line). This is to note that, as an improvement, the following two cases are considered which may arise in the case of DD-dependence computation:

Case 1: Statement Q_1 defines the values of an attribute x partially which is subsequently used by Q_2. The **presence** of WHERE clause in Q_1 determines this. In this case, Q_2 is DD-dependent on Q_1 as well as on the statement connecting the database. For instance, in Figure 1(c), the node 4 is DD-dependent on both the nodes 3 and 2.

Case 2: Statement Q_1 defines the values of an attribute x fully. This is determined by the the **absence** of WHERE clause in Q_1. In this case, all the subsequent database statements which use x will be DD-dependent on Q_1 only.

Observe that syntax-based construction may generate false dependences, and hence it is not optimal. For instance, in the example, the dependence $3 \rightarrow 4$ is a false dependence.

3.2 Semantics-based DOPDG

The syntax-based DOPDG may not provide an optimal set of dependences. This motivates researchers towards semantics-based dependence computation which focuses on the values rather than the attributes.

Given a SQL statement $Q = \langle A, \phi \rangle$ and its target table t. Suppose $\vec{x} = \texttt{USE}(A)$, $\vec{y} = \texttt{USE}(\phi)$ and $\vec{z} = \texttt{DEF}(\text{Q})$. According to the concrete semantics, suppose $\mathbf{S}[\![Q]\!](\rho_t, \rho_a) = (\rho_{t'}, \rho_a)$.

The *used* and *defined* part of t by Q are computed according to the following equations [13]:

$$\mathbf{A}_{use}(Q, t) = \rho_{t\downarrow\phi}(\vec{x}) \cup \rho_{t\downarrow\phi}(\vec{y}) \tag{1}$$

$$\mathbf{A}_{def}(Q, t) = \Delta(\rho_{t'}(\vec{z}), \rho_t(\vec{z})) \tag{2}$$

where

$t \downarrow \phi$: Set of tuples in table t which satisfies Condition-part ϕ.
$\rho_{t\downarrow\phi}(\vec{x})$: Values of $\vec{x}$ in $(t \downarrow \phi)$.
$\rho_{t\downarrow\phi}(\vec{y})$: Values of $\vec{y}$ in $(t \downarrow \phi)$.
Δ : Computes the difference between the original database state on which Q operates and the new database state obtained after performing the action-part A.

In other words, the function $\mathbf{A}_{use}$ maps to the part of the database information used by Q, whereas the function $\mathbf{A}_{def}$ defines the changes occurred in the database states when data is updated or deleted or inserted by Q. The following example illustrates this.

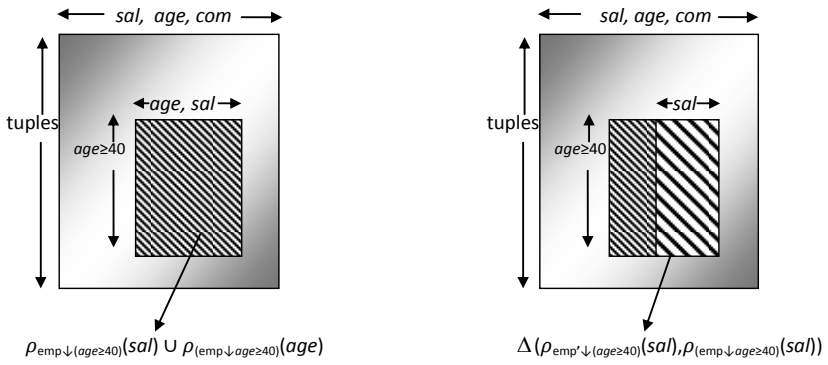

Fig. 2: $\mathbf{A}_{use}$ and $\mathbf{A}_{def}$ of table "`emp`" *w.r.t.* Q_{upd}

Example 2. Consider a database table "`emp`" in Figure 1(b) and the following update statement:

$$Q_{upd}: \texttt{UPDATE emp SET } sal := sal + 100 \texttt{ WHERE } age \geqslant 40$$

According to equations 1 and 2, the *used*-part and *defined*-part are as follows: $\mathbf{A}_{use}(Q_{upd}, \texttt{emp}) = \rho_{\texttt{emp}\downarrow(age\geqslant 40)}(sal) \cup \rho_{\texttt{emp}\downarrow(age\geqslant 40)}(age)$ and $\mathbf{A}_{def}(Q_{upd}, \texttt{emp}) = \Delta(\rho_{\texttt{emp}'}(sal), \rho_{\texttt{emp}}(sal))$. This is depicted pictorially in Figure 2.

Definition 3 (Semantics-based DD-Dependence [13]). *The SQL statement $Q_2 = \langle A_2, \phi_2 \rangle$ with $target(Q_2) = t'$ is DD-Dependent on another SQL statement Q_1 for Υ (denoted $Q_1 \xrightarrow{\Upsilon} Q_2$) if $Q_1 \in \{Q_{upd}, Q_{ins}, Q_{del}\}$ and the overlapping-part $\Upsilon = \mathbf{A}_{use}(Q_2, t') \cap \mathbf{A}_{def}(Q_1, t) \neq \emptyset$.*

We are now in position to propose a new approach to compute $\mathbf{A}_{use}$, $\mathbf{A}_{def}$, and the overlapping-part Υ. To do this, in the subsequent sections, we extend a well-know semantics-based formal analysis technique, the Abstract Interpretation Theory. To be specific, we define abstract semantics of database statements in the relational abstract domain of polyhedra [9]. As this can easily be extended to other relational or non-relational abstract domains, we also discuss the advantages and disadvantages of this analysis in various abstract domains in terms of preciseness, efficiency and scalability.

4 Extending Abstract Interpretation Theory

Abstract interpretation is a theory of abstraction and constructive sound approximation of the semantics of programming languages, aiming to infer or verify program's runtime properties including undecidable ones. This starts with the

formal definition of the semantics of a programming language (formally describing all possible program behaviors in all possible execution environments), continues with the formalization of program properties, and the expression of the strongest program property of interest in fixed point form [8].

Formally, the concrete domain $\mathtt{D}^c$ forms a complete lattice $\langle \wp(\mathtt{D}^c), \subseteq, \emptyset, \mathtt{D}^c, \cup, \cap \rangle$. On this domain, a semantics $\mathbf{S}$ is defined. In the same way, an abstract semantics $\bar{\mathbf{S}}$ is defined aiming to approximate the concrete one in a computable way. Formally, the abstract domain $\mathtt{D}^a$ has to form a complete lattice $\langle \mathtt{D}^a, \sqsubseteq, \emptyset, \mathtt{D}^a, \sqcup, \sqcap \rangle$. The concrete elements are related to the abstract domain by concretization function γ and abstraction function α. In order to obtain a sound analysis, we require that γ and α form a Galois connection [8]. An abstract semantics $\bar{\mathbf{S}}$ is defined as a sound approximation of the concrete one, *i.e.*, $\forall a \in \mathtt{D}^a.\ \alpha \circ \mathbf{S}[\![\gamma(a)]\!] \sqsubseteq \bar{\mathbf{S}}[\![a]\!]$.

4.1 Relational polyhedra domain as abstraction

Domain of Polyhedra. [3] The regions in n-dimensional space $\mathbb{R}^n$ bounded by finite sets of hyperplanes are called polyhedra. Let $x_1, x_2, \ldots\ x_n$ be the program variables. We represent by $\vec{l} = \langle l_1, l_2, \ldots l_n \rangle \in \mathbb{R}^n$, an n-tuple (vector) of real numbers. By $\beta = \vec{l}.\vec{x} \otimes m$ where $\vec{l} \neq \vec{0}$, $\vec{x} = \langle x_1, x_2, \ldots, x_n \rangle$, $m \in \mathbb{R}$, $\otimes \in \{\geq, >\}$, we represent a linear inequality over $\mathbb{R}^n$. A linear inequality defines an affine half-space of $\mathbb{R}^n$. If $\mathtt{P}$ is expressed as the intersection of a finite number of affine half-spaces of $\mathbb{R}^n$, then $\mathtt{P} \in \mathbb{R}^n$ is a convex polyhedron. Formally, a convex polyhedron $\mathtt{P} = (\Theta, n)$ is a set of linear restraints $\Theta = \{\beta_1, \beta_2 \ldots \beta_m\}$ on $\mathbb{R}^n$. Equivalently, $\mathtt{P}$ can be represented by frame representation which is a collection of generators *i.e.* *vertices* and *rays* [6]. On the other hand, given a set of restraints Θ on $\mathbb{R}^n$, a set of solutions or points $\{\sigma \mid \sigma \models \Theta\}$ defines a polyhedron $\mathtt{P} = (\Theta, n)$.

Forming Abstract Lattice on the domain of Polyhedra [9]. The set of polyhedra $\mathfrak{p}$ with partial order $\sqsubseteq$ forms a complete lattice $\mathtt{L}^a = \langle \mathfrak{p}, \sqsubseteq, \emptyset, \mathbb{R}^n, \sqcup, \sqcap \rangle$ where $\emptyset$ is the bottom element and $\mathbb{R}^n$ is top element. Given $\mathtt{P}_1, \mathtt{P}_2 \in \mathfrak{p}$, the partial order, meet and join operations are defined below:

- $\mathtt{P}_1 \sqsubseteq \mathtt{P}_2$ if and only if $\gamma(\mathtt{P}_1) \subseteq \gamma(\mathtt{P}_2)$, where $\gamma(\mathtt{P})$ represents the set of solutions or points in $\mathtt{P}$ as concrete values.
- $\mathtt{P}_1 \sqcap \mathtt{P}_2$ is the convex polyhedron containing exactly the set of points $\gamma(\mathtt{P}_1) \cap \gamma(\mathtt{P}_2)$.
- $\mathtt{P}_1 \sqcup \mathtt{P}_2$ is not necessarily a convex-polyhedron. Therefore, the least polyhedron enclosing this union is computed in terms of convex hull.

Galois Connection. Let $\mathtt{L}^c = \langle \wp(\mathtt{Val}), \subseteq, \emptyset, \mathtt{Val}, \cup, \cap \rangle$ be the concrete lattice defined over concrete domain of values $\mathtt{Val}$ and $\mathtt{L}^a = \langle \mathfrak{p}, \sqsubseteq, \emptyset, \mathbb{R}^n, \sqcup, \sqcap \rangle$ be an abstract lattice over the domain of polyhedra. The Galois Connection is defined as $\langle \mathtt{L}^c, \alpha, \gamma, \mathtt{L}^a \rangle$ such that $\alpha(S) \sqsubseteq \mathtt{P} \iff S \subseteq \gamma(\mathtt{P})$ where $S \in \wp(\mathtt{Val})$ is a set of concrete values and $\mathtt{P} \in \mathfrak{p}$ is a polyhedra. Some useful operations in the abstract domain include emptiness checking, projection, etc [4, 9].

[3] The Abstract Domain of Polyhedra Library is available in [2, 19].

4.2 Defining Abstract Semantics in relational polyhedra domain

The abstract transition semantics is defined as $\bar{\mathbf{S}}$: $\mathbb{C}\times \mathfrak{p} \rightarrow \wp(\mathfrak{p})$ where $\mathbb{C}$ is the set of statements and $\mathfrak{p}$ is the set of all polyhedra. It defines an abstract semantics of a statement in the domain of polyhedra by specifying how the execution of a statement on a polyhedron results into a set of new polyhedra.

Let us define the abstract transition semantics for imperative as well as database statements, and the abstract semantics of database application using data-flow analysis.

Assignment [9]: $\bar{\mathbf{S}}[\![x_j := e]\!](\mathtt{P}) = \{\mathtt{P}'\}$ where $\mathtt{P}'$ is obtained as follows: (*i*) **Case**-1: If e is a non-linear expression or the assignment is non-invertible, then we simply project-out the corresponding variables from the equations which results into a new polyhedron $\mathtt{P}'$; (*ii*) **Case**-2: Otherwise, we introduce a fresh variable x_j' to hold the value of e, then we project out x_j, and finally we reuse x_j' in place of x_j which results into $\mathtt{P}'$.

Example 3. Given $\mathtt{P}$=(β, n)=($\{x\geqslant 3, y\geqslant 2\}$, 2). The equivalent frame representation (*vertices* and *rays*) of $\mathtt{P}$ is $\mathtt{V}$={(3, 2)} and $\mathtt{R}$={(1, 0), (0, 1)}. The transition semantics of assignment $x := x + y$ is define as

$$\bar{\mathbf{S}}[\![x := x + y]\!](\{x \geqslant 3, y \geqslant 2\}, 2) = \{\mathtt{P}'\}$$

where $\mathtt{P}' = (\{x - y \geqslant 3, y \geqslant 2\}, 2)$ whose equivalent frame representation is $\mathtt{V}$={(5, 2)} and $\mathtt{R}$={(1, 0), (-1, -1)}.

Test [9]: Given a boolean expression in the form of linear inequalities $\beta = \vec{l}.\vec{x}\otimes m$ and a polyhedron $\mathtt{P}$: $\bar{\mathbf{S}}[\![\beta]\!]\mathtt{P} = \{\mathtt{P}_T,\ \mathtt{P}_F\}$ where $\mathtt{P}_T = (\mathtt{P} \sqcap \beta)$ and $P_F = (\mathtt{P} \sqcap \neg\beta)$.

Example 4. Given $\mathtt{P}$=(β, n)=($\{x \geqslant 8, y \geqslant 6\}$, 2). The equivalent frame representation (*vertices* and *rays*) of $\mathtt{P}$ is $\mathtt{V}$={(8, 6)} and $\mathtt{R}$={(1, 0), (0, 1)}. The transition semantics of boolean expression $x \geqslant 20$ is define as: $\bar{\mathbf{S}}[\![x \geqslant 20]\!]\mathtt{P} = \{\mathtt{P}_T, \mathtt{P}_F\}$ where $\mathtt{P}_T = (\{x \geqslant 20, y \geqslant 6\}, 2)$ whose equivalent frame representation is $\mathtt{V}_T$={(20, 6)} and $\mathtt{R}_T$={(1, 0), (0, 1)} and $\mathtt{P}_F = (\{x \geqslant 8, -x \geqslant -19, y \geqslant 6\}, 2)$ whose equivalent frame representation is $\mathtt{V}_F$={(8, 6),(19, 6)} and $\mathtt{R}_F$={(0, 1)}.

UPDATE: $\bar{\mathbf{S}}[\![\mathtt{UPDATE}(\vec{v_d}, \vec{e}),\ \phi\rangle]\!]\mathtt{P} = \{\mathtt{P}'_T\ ,\ \mathtt{P}_F\}$ where

$\mathtt{P}_T = (\mathtt{P} \sqcap \phi)$.
$\mathtt{P}'_T = \bar{\mathbf{S}}[\![\mathtt{UPDATE}(\vec{v_d}, \vec{e})]\!]\mathtt{P}_T = \bar{\mathbf{S}}[\![\vec{v_d} := \vec{e}]\!]\mathtt{P}_T$.
$\mathtt{P}_F = (\mathtt{P} \sqcap \neg\phi)$.

We denote by the notation $\vec{v_d}$:= $\vec{e}$ a series of assignments $\langle v_1 := e_1, v_2 := e_2, \ldots, v_n := e_n\rangle$ where $\vec{v_d} = \langle v_1, v_2, \ldots, v_n\rangle$ and $\vec{e}$=$\langle e_1, e_2, \ldots, e_n\rangle$, which follow the transition semantic definition for the assignment statement.

DELETE: $\bar{\mathbf{S}}[\![\langle\mathtt{DELETE}(\vec{v_d}),\ \phi\rangle]\!]\mathtt{P} = \{(\mathtt{P} \sqcap \neg\phi)\}$

INSERT: $\bar{\mathbf{S}}[\![\langle \texttt{INSERT}(\vec{v_a}, \vec{e}),\ \phi\rangle]\!]\mathtt{P} = \{\mathtt{P} \sqcup \mathtt{P}_{new}\} = \{\mathtt{P}'\}$
where $\mathtt{P}_{new}$ represents a polyhedron corresponding to the new inserted tuple values.

SELECT: The select operation does not modify any information in a polyhedron. Therefore, the transition semantics is defined as:

$$\begin{aligned}&\bar{\mathbf{S}}[\![\langle \texttt{SELECT}\big(v_a,\ f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi_2,\ g(\vec{e})\big),\ \phi_1\rangle]\!]\mathtt{P}_{std}\\ =&\bar{\mathbf{S}}[\![\langle \texttt{SELECT}\big(v_a,\ f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi_2,\ g(\vec{e})\big),\ true\rangle]\!]\mathtt{P}_T \quad \sqcup\\ &\bar{\mathbf{S}}[\![\langle \texttt{SELECT}\big(v_a,\ f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi_2,\ g(\vec{e})\big),\ false\rangle]\!]\mathtt{P}_F \quad = \quad \{\mathtt{P}_{std}\}\end{aligned}$$

The following example illustrates the semantics of `UPDATE, DELETE, and INSERT` statements.

Example 5. Consider the table "`std`" in Figure 3(a). The abstract representation of "`std`" in the form of polyhedron, depicted in Figure 3(b), is

$$\mathtt{P}_{std} = \langle\{roll \geqslant 1, -roll \geqslant -4, mark \geqslant 400, -mark \geqslant -1000, rank \geqslant 18, -rank \geqslant -62\}, 3\rangle$$

Consider the following statements:

$$\begin{aligned}Q_{upd} =&\texttt{UPDATE std SET } mark = mark + \$vmark \texttt{ WHERE } rank \geqslant 30\\ Q_{del} =&\texttt{DELETE FROM std WHERE } rank \geqslant 30\\ Q_{ins} =&\texttt{INSERT INTO std}(roll, mark, rank) \texttt{ VALUES } (5, 300, 20)\end{aligned}$$

where "$vmark" is an application variable which accepts run-time input (positive values). The equivalent abstract syntax are:

$$\begin{aligned}Q_{upd} =&\langle \texttt{UPDATE}(\langle mark\rangle, \langle mark + \$vmark\rangle),\ rank \geqslant 30\rangle\\ Q_{del} =&\langle \texttt{DELETE}(\langle roll, mark, rank\rangle),\ rank \geqslant 30\rangle\\ Q_{ins} =&\langle \texttt{INSERT}(\langle roll, mark, rank\rangle, \langle 5, 300, 20\rangle), false\rangle\end{aligned}$$

The transition semantics of Q_{upd} is defined as:

$$\bar{\mathbf{S}}[\![Q_{upd}]\!]\mathtt{P}_{std} = \bar{\mathbf{S}}[\![\langle \texttt{UPDATE}(\langle mark\rangle, \langle mark + \$vmark\rangle), rank \geqslant 30\rangle]\!]\mathtt{P}_{std} = \{\mathtt{P}'_T,\ \mathtt{P}_F\}$$

where $\mathtt{P}'_T = \langle\{roll \geqslant 1, -roll \geqslant -4, mark \geqslant 400, rank \geqslant 30, -rank \geqslant -62\}, 3\rangle$ and $\mathtt{P}_F = \langle\{roll \geqslant 1, -roll \geqslant -4, mark \geqslant 400, -mark \geqslant -1000, rank \geqslant 18, -rank \geqslant -29\}, 3\rangle$.

The pictorial representation of $\mathtt{P}_F$, $\mathtt{P'}_T$ are in Figures 3(c) and 3(d) respectively.

The transition semantics of Q_{del} is:

$$\begin{aligned}\bar{\mathbf{S}}[\![Q_{del}]\!]\mathtt{P}_{std} =&\bar{\mathbf{S}}[\![\langle \texttt{DELETE}(\langle roll, mark, rank\rangle),\ rank \geqslant 30\rangle]\!]\mathtt{P}_{std}\\ &= \{\mathtt{P}_{std} \sqcap \neg(rank \geqslant 30)\} = \{\mathtt{P}'\}\end{aligned}$$

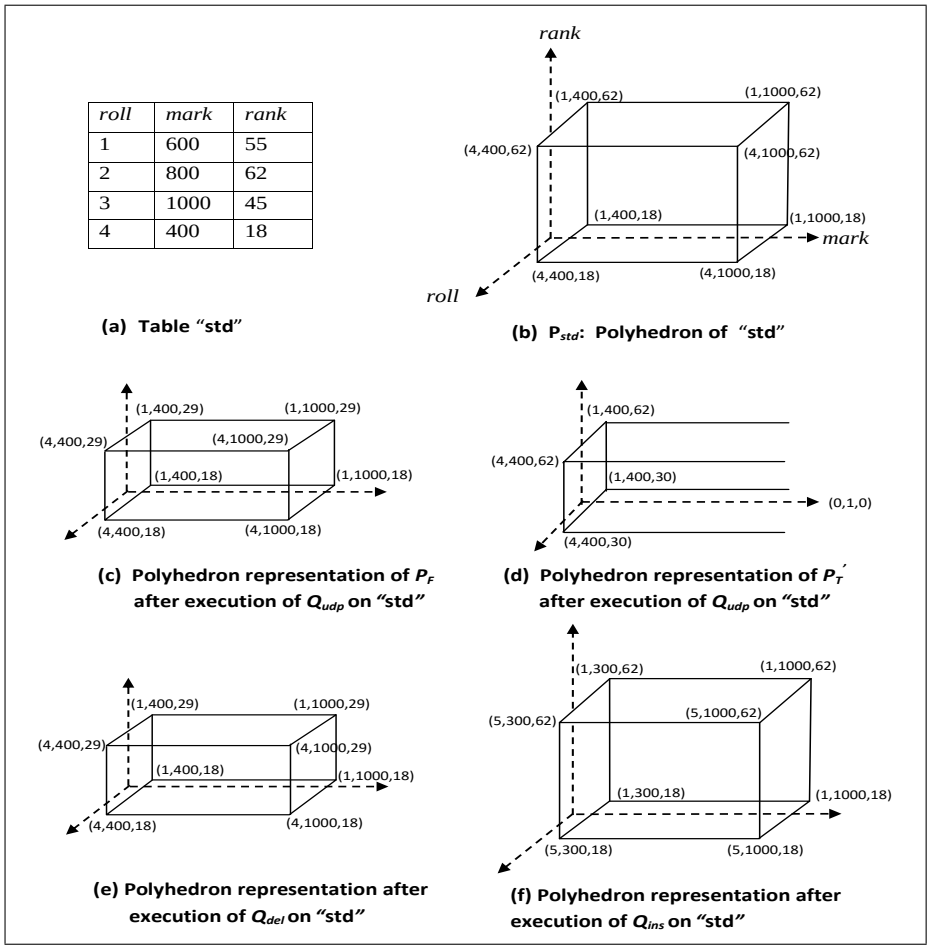

Fig. 3: Polyhedra representation of "*std*" and after database operations on "*std*"

where $\mathrm{P}' = \langle\{roll \geqslant 1, -roll \geqslant -4, mark \geqslant 400, -mark \geqslant -1000, rank \geqslant 18, -rank \geqslant -29\}, 3\rangle$ which is depicted in Figure 3(e).

The transition semantics of Q_{ins} is:

$$\bar{\mathbf{S}}[\![Q_{ins}]\!]\mathrm{P}_{std} = \{\mathrm{P}_{std} \sqcup \langle\{roll = 5, mark = 300, rank = 20\}\rangle\} = \{\mathrm{P}'\}$$

The resulting polyhedron P' is shown in Figure 3(f).

Theorem 1 (Correctness). *Given a table* $\mathtt{t}$*, suppose the application of database statement* Q *results* $\mathtt{t}'$*, i.e.* $Q(\mathtt{t}) = \mathtt{t}'$*. Let* P *be the polyhedron representation of* $\mathtt{t}$ *such that* $\bar{\mathbf{S}}[\![Q]\!]P = \{P'\}$*. The transition relation* $\bar{\mathbf{S}}$ *is correct w.r.t.* γ *if* $Q(t) \subseteq \gamma(\bar{\mathbf{S}}[\![Q]\!]P)$*.*

Proof. This is proved by using Galois Connection [8]. We skip the proof for brevity.

Algorithm to compute abstract semantics of database applications. The algorithm **Abstract-Semantics** computes polyhedron abstraction of database-values at each program point based on the data-flow equations [28] using semantic transition relation $\bar{\mathbf{S}}$. The data-flow equations are defined on the control-flow graph of the database application which consists of various nodes: *start*, *end*, *assignment*, *test*, *join*, *DB-connect*, *update*, *delete*, *insert*, *select*. The algorithm starts with the polyhedron representation of the initial database and applies data-flow equations until least fixed point solution is reached. The final output represents a set of polyhedral representation of the database at each program point obtained after sequential execution of the program. This result is used to compute $\mathbf{A}_{use}$, $\mathbf{A}_{def}$, Υ and the semantics-based dependences (see section 4.3 and 4.4). Observe that if the initial database is unknown then the domain range of each attribute data type and other integrity properties and constraints are used to represent the initial polyhedron as an overapproximation of all possible initial database states.

Algorithm 1: Abstract-Semantics

Input: Database Application of size n and database $\mathtt{dB}$
Output: Set of polyhedra occurs at each of the program points

Let $\mathtt{P}_{\mathtt{dB}}$ represents the polyhedron representation of the initial database $\mathtt{dB}$. Let $\mathtt{P}(c_i)$ where $i = 1, \ldots, n$, represents set of polyhedra occurs at i^{th} statement c_i of the application. Let $pred(c_i)$ represents the set of predecessor statements of c_i according to the control-flow graph of the application,

1. Compute $\mathtt{P}_{\mathtt{dB}}$ which is the polyhedron representation of the initial database.
2. $\forall i \in [0, \ldots, n]$, $\mathtt{P}(c_i) := \emptyset$.
3. Repeat step 4 until least fix-point is reached.
4. $\forall i = 1, \ldots, n$: apply the data flow equations $\mathtt{DF}(c_i)$ defined below:

Switch($\mathtt{DF}(c_i)$){

Case $\mathtt{DF}(start)$: $\emptyset$.

Case $\mathtt{DF}(end)$: $\bigsqcup_{c_j \in pred(c_i)} \mathtt{P}(c_j)$.

Case $\mathtt{DF}(assignment)$: $\bigsqcup_{c_j \in pred(c_i)} \bar{\mathbf{S}}[\![x_j := e]\!]\,(\mathtt{P}(c_j))$

Case $\mathtt{DF}(test)$: $\bigsqcup_{c_j \in pred(c_i)} \bar{\mathbf{S}}[\![\vec{l}.\vec{x} \otimes m]\!]\,(\mathtt{P}(c_j))$

Case $\mathtt{DF}(join)$: $\bigsqcup_{c_j \in pred(c_i)} \mathtt{P}(c_j)$.

Case $\mathtt{DF}(DB\text{-}connect)$: $\mathtt{P}_{\mathtt{dB}}$

Case $\mathtt{DF}(update)$: $\bigsqcup_{c_j \in pred(c_i)} \bar{\mathbf{S}}[\![\mathtt{UPDATE}(\vec{v}_d, \vec{e}),\ \phi\rangle]\!]\,(\mathtt{P}(c_j))$

Case $\mathtt{DF}(delete)$: $\bigsqcup_{c_j \in pred(c_i)} \bar{\mathbf{S}}[\![\mathtt{DELETE}(\vec{v}_d),\ \phi\rangle]\!]\,(\mathtt{P}(c_j))$

Case $\mathtt{DF}(insert)$: $\bigsqcup_{c_j \in pred(c_i)} \bar{\mathbf{S}}[\![\mathtt{INSERT}(\vec{v}_d, \vec{e})\rangle]\!]\,(\mathtt{P}(c_j))$

Case $\mathtt{DF}(select)$: $\bigsqcup_{c_j \in pred(c_i)} \bar{\mathbf{S}}[\![\langle \mathtt{SELECT}\Big(f(\vec{e'}), r(\vec{h}(\vec{x})), \phi_2, g(\vec{e})\Big), \phi_1\rangle]\!](\mathtt{P}(c_j))$

}

End

4.3 Computation of $\mathbf{A}_{use}$ and $\mathbf{A}_{def}$

Recall the equations 1 and 2 in section 3.2 to compute *used*- and *defined*-part of the target table t by $Q = \langle A, \phi \rangle$.

The computation of $\mathbf{A}_{use}$, $\mathbf{A}_{def}$ *w.r.t.* abstract semantics are defined below. Observe that $\mathbf{A}_{def}(Q, t)$ is represented in the form of two-tuple $\langle \mathtt{P}_T, \mathtt{P}'_T \rangle$ where $\mathtt{P}_T$ and $\mathtt{P}'_T$ are the components representing the true-part of the polyhedra of t before and after the execution of Q.

UPDATE: $\bar{\mathbf{S}}[\![\langle \mathtt{UPDATE}(\vec{v}_d, \vec{e}),\ \phi \rangle]\!]\mathtt{P}$
$= \bar{\mathbf{S}}[\![\langle \mathtt{UPDATE}(\vec{v}_d, \vec{e}),\ true \rangle]\!]\mathtt{P}_T \ \cup\ \bar{\mathbf{S}}[\![\langle \mathtt{UPDATE}(\vec{v}_d, \vec{e}),\ false \rangle]\!]\mathtt{P}_F = \{\mathtt{P}'_T\ ,\ \mathtt{P}_F\}$

$$\mathbf{A}_{use}(Q_{upd},\ t) = \langle \mathtt{P}_T \rangle \qquad \text{and} \qquad \mathbf{A}_{def}(Q_{upd},\ t) = \langle \mathtt{P}_T,\ \mathtt{P}'_T \rangle$$

DELETE: $\bar{\mathbf{S}}[\![\langle \mathtt{DELETE}(\vec{v}_d),\ \phi \rangle]\!]\mathtt{P} = \bar{\mathbf{S}}[\![\langle \mathtt{DELETE}(\vec{v}_d),\ true \rangle]\!]\mathtt{P}_T = \{\mathtt{P}'\}$

$$\mathbf{A}_{use}(Q_{del},\ t) = \langle \mathtt{P}_T \rangle \qquad \text{and} \qquad \mathbf{A}_{def}(Q_{del},\ t) = \langle \mathtt{P}_T, \emptyset \rangle$$

INSERT: $\bar{\mathbf{S}}[\![\langle \mathtt{INSERT}(\vec{v}_d, \vec{e}), \phi \rangle]\!]\mathtt{P} = \bar{\mathbf{S}}[\![\langle \mathtt{INSERT}(\vec{v}_d, \vec{e}), true \rangle]\!]\mathtt{P} = \{\mathtt{P} \sqcup \mathtt{P}_{new}\}$
where $\mathtt{P}_{new}$ is the polyhedron represented by the inserted tuple values.

$$\mathbf{A}_{use}(Q_{ins},\ t) = \langle \emptyset \rangle \qquad \text{and} \qquad \mathbf{A}_{def}(Q_{ins},\ t) = \langle \emptyset, \mathtt{P}_{new} \rangle$$

SELECT: $\bar{\mathbf{S}}[\![\langle \mathtt{SELECT}\big(v_a,\ f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi_2,\ g(\vec{e})\big),\ \phi_1 \rangle]\!]\mathtt{P}$
$= \bar{\mathbf{S}}[\![\langle \mathtt{SELECT}\big(v_a,\ f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi_2,\ g(\vec{e})\big),\ true \rangle]\!]\mathtt{P}_T \ \bigcup$
$\bar{\mathbf{S}}[\![\langle \mathtt{SELECT}\big(v_a,\ f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi_2,\ g(\vec{e})\big),\ false \rangle]\!]\mathtt{P}_F = \{\mathtt{P}\}$

$$\mathbf{A}_{use}(Q_{sel},\ t) = \langle \mathtt{P}_T \rangle \qquad \text{and} \qquad \mathbf{A}_{def}(Q_{sel},\ t) = \langle \emptyset, \emptyset \rangle$$

Observe that, in case of update operation, $\mathbf{A}_{def}$ consists of two elements: the first one represents the polyhedron before updation and the second one represents the polyhedron after updation. In $\mathbf{A}_{use}$ and $\mathbf{A}_{def}$, we keep both the elements separated by comma, instead of performing union or minus operation, as it reduces false positive and the computational complexity significantly. Example 6 illustrates this on the running example in Figure 1.

Example 6. Consider the example in Figure 1. The abstract representation of the table "`emp`" (in Figure 1(b)) in the form of polyhedron is:

$$\mathtt{P}_{emp} = \langle \{com \geqslant 10, -com \geqslant -30, sal \geqslant 800, -sal \geqslant -3000, age \geqslant 20, -age \geqslant -62\}, 3 \rangle$$

This is shown in Figure 4(a). The abstract syntax at program points 3 and 4 are:

$$Q_3 : \langle \mathtt{UPDATE}(\langle sal \rangle\ \langle sal\ +\ 100 \rangle),\ (com\ +\ age) \geqslant\ 60 \rangle$$
$$Q_4 : \langle \mathtt{SELECT}(rs1,\ id,\ \mathtt{ALL}(\mathtt{AVG}(sal)),\ true,\ id),\ (age + com) \leqslant\ 55 \rangle$$

where *id* represents identity functions for f and g. The transition semantics of Q_3 is:

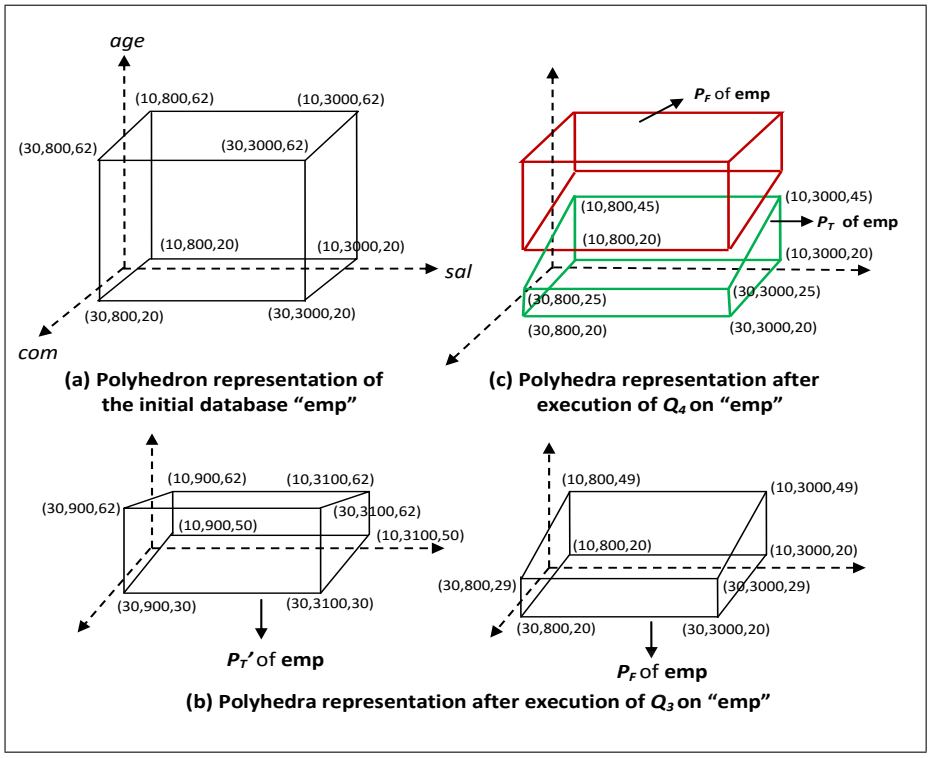

Fig. 4: Polyhedra representation of the database "emp" w.r.t Q_3 and Q_4

$\bar{\mathbf{S}}[\![Q_3]\!]\mathrm{P}_{emp} = \bar{\mathbf{S}}[\![\langle \mathtt{UPDATE}(\langle sal\rangle, \langle sal{+}100\rangle),\ (age{+}com) \geqslant 60\rangle]\!]\mathrm{P}_{emp} = \{\mathrm{P}'_T,\ \mathrm{P}_F\}$, where

$$\mathrm{P}_T = \langle\{com \geqslant 10, -com \geqslant -30,\ sal \geqslant 800, -sal \geqslant -3000,\ age \geqslant 30, -age \geqslant -62,\ (age + com) \geqslant 60\}, 3\rangle.$$

$$\mathrm{P}'_T = \langle\{com \geqslant 10, -com \geqslant -30,\ sal \geqslant 900, -sal \geqslant -3100,\ age \geqslant 30, -age \geqslant -62,\ (age + com) \geqslant 60\}, 3\rangle.$$

$$\mathrm{P}_F = \langle\{com \geqslant 10, -com \geqslant -30,\ sal \geqslant 800, -sal \geqslant -3000,\ age \geqslant 20, -age \geqslant -49,\ -(age + com) \geqslant -59\}, 3\rangle.$$

This is depicted in Figure 4(b). Similarly, the transition semantics of Q_4 is:

$$\begin{aligned}\bar{\mathbf{S}}[\![Q_4]\!]\mathrm{P}_{emp} =& \bar{\mathbf{S}}[\![\langle \mathtt{SELECT}(rs1,\ id,\ \mathtt{ALL}(\mathtt{AVG}(sal)),\ true,\ id),\ (age + com) \leqslant 55\rangle]\!]\mathrm{P}_{emp}\\ =& \bar{\mathbf{S}}[\![\langle \mathtt{SELECT}(rs1,\ id,\ \mathtt{ALL}(\mathtt{AVG}(sal)),\ true,\ id),\ true\rangle]\!]\mathrm{P}_T \bigcup\\ & \bar{\mathbf{S}}[\![\langle \mathtt{SELECT}(rs1,\ id,\ \mathtt{ALL}(\mathtt{AVG}(sal)),\ true,\ id),\ false\rangle]\!]\mathrm{P}_F\\ =& \{\mathrm{P}_{emp}\},\ \text{where}\end{aligned}$$

$$\mathtt{P}_T = \langle\{com \geqslant 10, -com \geqslant -30,\ sal \geqslant 800, -sal \geqslant -3000, age \geqslant 20,\ -age \geqslant -45,\ -(age + com) \geqslant -55\}, 3\rangle.$$

$$\mathtt{P}_F = \langle\{com \geqslant 10,\ -com \geqslant -30, sal \geqslant 800,\ -sal \geqslant -3000, age \geqslant 26, -age \geqslant -62,\ (age + com) \geqslant 56\}, 3\rangle.$$

This is depicted in Figure 4(c). According to the abstract semantics, the *defined*-part by Q_3 and the *used*-part by Q_4 are:

$$\mathbf{A}_{def}(Q_3, \mathtt{emp}) = \langle \mathtt{P}_T,\ \mathtt{P}'_T\rangle \qquad \text{and} \qquad \mathbf{A}_{use}(Q_4, \mathtt{emp}) = \langle \mathtt{P}_T\rangle$$

4.4 Computation of Υ

According to the Definition 3, the dependence of Q_2 on Q_1 is denoted as $Q_1 \xrightarrow{\Upsilon} Q_2$ where $\bar{\mathbf{S}}[\![Q_1]\!](\rho_t, \rho_a) = \{(\rho_{t'}, \rho_a)\}$ and $\Upsilon = \mathbf{A}_{def}(Q_1, t) \cap \mathbf{A}_{use}(Q_2, t') \neq \emptyset$.

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

Case1. $\mathtt{P}_T^{Q_1} \sqcap \mathtt{P}_T^{Q_2} \neq \emptyset \wedge \mathtt{P}_T'^{Q_1} \sqcap \mathtt{P}_T^{Q_2} = \emptyset$ **Case2**. $\mathtt{P}_T^{Q_1} \sqcap \mathtt{P}_T^{Q_2} = \emptyset \wedge \mathtt{P}_T'^{Q_1} \sqcap \mathtt{P}_T^{Q_2} \neq \emptyset$

Case3. $\mathtt{P}_T^{Q_1} \sqcap \mathtt{P}_T^{Q_2} = \emptyset \wedge \mathtt{P}_T'^{Q_1} \sqcap \mathtt{P}_T^{Q_2} = \emptyset$ **Case4**. $\mathtt{P}_T^{Q_1} \sqcap \mathtt{P}_T^{Q_2} \neq \emptyset \wedge \mathtt{P}_T'^{Q_1} \sqcap \mathtt{P}_T^{Q_2} \neq \emptyset$

where $\sqcap$ is the greatest lower bound representing union operation, $\mathbf{A}_{def}(Q_1, t) = \langle \mathtt{P}_T^{Q_1}, \mathtt{P}_T'^{Q_1}\rangle$, and $\mathbf{A}_{use}(Q_2, t') = \mathtt{P}_T^{Q_2}$.

The SQL statements $Q1$ and $Q2$ are semantically independent when case 3 holds. That is,

$$\Upsilon = \mathbf{A}_{def}(Q_1, t) \cap \mathbf{A}_{use}(Q_2, t') = \emptyset \quad \text{iff} \quad \mathtt{P}_T^{Q_1} \sqcap \mathtt{P}_T^{Q_2} = \emptyset \wedge \mathtt{P}_T'^{Q_1} \sqcap \mathtt{P}_T^{Q_2} = \emptyset \qquad (3)$$

Example 7. In example 6, we have computed $\mathbf{A}_{def}(Q_3, \mathtt{emp}) = \langle \mathtt{P}_T,\ \mathtt{P}'_T\rangle$ and $\mathbf{A}_{use}(Q_4, \mathtt{emp}) = \langle \mathtt{P}_T\rangle$ at program points 3 and 4 of the program "`Prog`" in Figure 1. The dependence $Q_3 \xrightarrow{\Upsilon} Q_4$ does not exist semantically as

$$\mathtt{P}_T^{Q3} \sqcap \mathtt{P}_T^{Q4} = \emptyset \wedge \mathtt{P}_T'^{Q3} \sqcap \mathtt{P}_T^{Q4} = \emptyset$$

In words, the statement at program point 4 does not semantically dependent on the statement at 3 for the attribute *sal* in Figure 1.

Algorithm to compute semantics-based dependences based on abstract semantics. The algorithm **semDOPDG** takes a list of *used*- and *defined*-parts ($\mathbf{A}_{use}$ and $\mathbf{A}_{def}$) at each program statement c_i of the database application of size n, and computes its semantic-based DOPDG. The algorithm creates edges 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 following the equation 3. 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 $\mathbf{A}_{def}(i) \sqsubseteq \mathbf{A}_{def}(j)$ verifies whether *defined*-part at program point c_i is fully covered by the *defined*-part at program point c_j. In this case, the true value in flag variable represents the dependency between c_i and c_j.

Algorithm 2: semDOPDG

Input: *used*- and *defined*-part ($\mathbf{A}_{use}$, $\mathbf{A}_{def}$) by all database statements in the program.
Output: Semantic-based DOPDG

```
Set flag=TRUE
for i =1 to n-1 do
    for j=i+1 to n do
        if A_def(i) ⊓ A_use(j)= ∅ then
            Set flag = FALSE
        else
            Add the edge from i^th node to j^th node (i → i)
        if flag=True then
            if A_def (i) ⊑ A_def (j) then
                flag = TRUE;
                BREAK;
End
```

Soundness. The semantics-based dependence computation is sound if a dependency does not exist in the abstract domain then it must not exist in the concrete domain. In other words, semantics independences in the abstract domain implies semantics independences in the concrete domain.

Theorem 2 (Soundness of Semantic Independences). *Given two statements Q_1 and Q_2, let $\boldsymbol{A}_{def}(Q_1)$ and $\boldsymbol{A}_{use}(Q_2)$ be the database defined- and used-parts respectively represented in abstract polyhedra domain. The computation of semantic independence is sound if $\forall X \subseteq \gamma(\boldsymbol{A}_{def}(Q_1)), \forall Y \subseteq \gamma(\boldsymbol{A}_{use}(Q_2)) : \; X \cap Y \subseteq \gamma(\boldsymbol{A}_{def}(Q_1) \cap \boldsymbol{A}_{use}(Q_2))$ which implies that $\boldsymbol{A}_{def}(Q_1) \cap \boldsymbol{A}_{use}(Q_2) = \emptyset \Rightarrow X \cap Y = \emptyset$.*

Proof. We skip the proof for brevity.

5 Discussions of the proposal *w.r.t.* the literature

This section discusses some existing notable directions towards semantics-based dependence computations of database applications, and provides a comparative analysis of our approach *w.r.t.* the literature.

Query-containment as Dependency Computation. The query containment is the problem of checking whether for every database, the result of one query is a subset of the result of another query [26]. Formally, a query Q_1 is said to be contained in a query Q_2, denoted $Q_1 \sqsubseteq Q_2 \iff \forall D \; Q_1(D) \subseteq Q_2(D)$ and $Q_1 \equiv Q_2 \iff Q_1 \sqsubseteq Q_2 \wedge Q_2 \sqsubseteq Q_1$, where $Q(D)$ represents the result of query Q on database D. The complexity of conjunctive query containment is NP-complete [26]. Query containment is useful for the various purposes of query optimization, detecting independence of queries from database updates, rewriting queries using views, etc. [23, 26].

Dependency computation problem of database applications considers not only `SELECT` query, but also DML commands `INSERT`, `UPDATE`, `DELETE`. Therefore, solutions to the query containment problem are unable to provide a complete solution for the case of semantics-based dependency computation of database applications involving both *write-write* and *write-read* operations.

Propagation Analysis of Condition-Action rules. As a solution to compute overlapping part, Willmor et al. [33] refer to the analysis of Condition-Action rules of expert database system proposed in [3]. These rules are expressed in an extended relational algebra in the form $\mathtt{E}_{cond} \longrightarrow \mathtt{E}_{act}$ where $\mathtt{E}_{cond}$ and $\mathtt{E}_{act}$ represent the rule's condition and the rule's action as a data modification operation respectively. The propagation algorithm performs a syntactic analysis to predict how the action of one rule can affect the condition of another. In other words, the analysis checks whether the condition sees any data inserted or deleted or modified due to the action. Therefore, such kind of conditions verifications makes the computational complexity of dependence computation exponential *w.r.t.* the number of defining statements. Moreover, the algorithm fails to capture the semantic independencies when an attribute x is partially defined by more than one database statements (in sequence) and finally is used by another statement. In a nutshell, the propagation analysis is flow-insensitive.

Our Proposed Approach. Semantics in polyhedral abstract domain captures the relations among program variables and attributes and results into a more precise analysis with the cost of high computation complexity. Nevertheless, several other relational and non-relational abstract domains exist which provides a tradeoff between preciseness, efficiency and scalability. Intuitively, preciseness of the analysis in relational abstract domain are benefitted significantly when more number of relations among variables or attributes are present in the program itself, *e.g.* in the `WHERE` clause or in the conditional or iterative statements. For instance, consider the following statements:

$$Q_1 : \texttt{UPDATE}\ \mathtt{t}\ \texttt{SET}\ a := a + 1\ \texttt{WHERE}\ a \leqslant 3$$
$$Q_2 : \texttt{SELECT}\ a\ \texttt{FROM}\ \mathtt{t}\ \texttt{WHERE}\ b \geqslant 12$$

Due to the absence of any relation among attributes in the `WHERE` clauses of both statements, the polyhedral analysis yields a conservative results $Q_1 \rightarrow Q_2$ which may not be true in some case. This is worthwhile to mention that we have avoided convex-hull (union) operation partially in the proposal which reduces the computational complexity significantly.

Let us discuss the scenario in a weakly relational abstract domain "octagon" of the form $c_i x_i + c_j x_j \leqslant c$ where x_i and x_j are program variables, c_i, $c_j \in$ [-1, 0, 1] and $c \in \mathbb{R} \cup \{\infty\}$ [27]. It can be seen as a restriction of the polyhedra domain where each inequality constraint only involves at most two variable and unit coefficients. In the case of dependency computation, the result produced by octagon abstract domain is less precise than polyhedra abstract domain, but is less costlier as compared to polyhedra. In some cases, octagon abstract domain

is not applicable where more than two attributes in a SQL statement formed a relation or the attributes in a SQL statement has non unite coefficient. For example, consider the following two SQL statements:

$$Q_1 : \texttt{UPDATE t SET } a := a + 1 \texttt{ WHERE } 3 * a + 2 * b \geqslant 35$$
$$Q_2 : \texttt{SELECT } a \texttt{ FROM t WHERE } 3 * a + 2 * b \leqslant 30$$

where statements have non unite coefficients in the constraints $3 * a + 2 * b \geqslant 35$ and $3 * a + 2 * b \leqslant 30$ which can not be represented in the form of octagonal constraints.

Most importantly, our proposed approach, irrespective of the abstract domains, serves as a pwoerful tool to give a solution in the case of undecidable scenario when no initial database state is provided. In such situation, the analysis starts with an overapproximation of all possible initial database states which is obtained by considering domain ranges of attributes' data types, integrity constraints, etc.

Computational Complexity. The computation of abstract semantics over the abstract domain of polyhedra is based on the polyhedra operations in terms of vertices and rays. The suitable libraries [2], [19] are available for polyhedra computation. Although, in general, the computational complexity is exponential ($O(2^n)$) [21], however for fixed dimension the complexity can be reduced to linear [25]. Alternatively, weakly relational abstract domains, *e.g.* domain of octagon, difference bound matrix [27], etc. exist which require polynomial time and space complexity, but they are less precise than polyhedra abstract domain and they supports more limited number of relations between program variables.

6 Applications

Semantic-based approach computes an optimal set of dependences in programs, yielding to more precise dependence graphs. This refinement plays crucial role in different fields of the software engineering. Some of the applications are ***(i) Language-based Information Flow Security Analysis*** [20], [30]: As dependence graph-based approaches are flow-sensitive, they are widely accepted approaches to perform language-based information flow security analysis. Several formal approaches also implicitly or explicitly use the notion of dependences. ***(ii) Slicing*** [16], [22]: Program slicing is one of the most suitable static analysis techniques used in many software engineering scenarios, *e.g.* debugging, testing, code-understanding, code-optimization etc. ***(iii) Data provenance*** [5]: Data provenance is a source of information or contextual information about an object. Its intention is to show how (part of) the output of a query depended on (part of) its input. Semantics-based dependence analysis techniques are familiar for semantic characterization of data provenance. ***(iv) Concurrent System modeling*** [11], [18]: In case of software transaction, semantic-based dependency computations play an important role to schedule various transactions for concurrent execution without lose of database consistency. ***(v) Materialization View Creation***

[31]: Attribute dependences are significantly useful in creation of materialized view of databases. Semantics-based approach can be applied in this context as well.

7 Conclusions

In this paper, we proposed a novel approach to compute semantics-based dependences in database applications based on the Abstract Interpretation framework. The semantic independences among database statements are computed based on the abstract semantics of database statements in the affine domain of polyhedra. Although more precise, however, as an alternative we may also use other weakly relational abstract domain as well (*e.g.* domain of octagons, difference bound matrix, etc.) to reduce the computational complexity with the cost of preciseness. The proposed approach serves as a powerful tool to give a solution in the case of undecidable scenario when no initial database state is provided. We are now implementing a prototype based on our proposal aiming to apply on real benchmark codes and to check the strength of the proposal in terms of precision, scalability and efficiency *w.r.t.* existing techniques.

Acknowledgment

This work is partially supported by the ERDF - European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project "POCI-01-0145-FEDER-006961", and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia as part of project "UID/EEA/50014/2013".

References

1. Ahuja, B.K., Jana, A., Swarnkar, A., Halder, R.: On preventing SQL injection attacks. In: 2nd International Doctoral Symposium on Applied Computation and Security Systems (ACSS), LNCS, May, 23-25, 2015. pp. 49–64. Springer (2015)
2. Bagnara, R., Hill, P.M., Zaffanella, E.: The ppl: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Tech. rep., Dipartimento di Matematica, Università di Parma, Italy (2006)
3. Baralis, E., Widom, J.: An algebraic approach to rule analysis in expert database systems (1994)
4. Chen, L., Minè, A., Cousot, P.: A sound floating-point polyhedra abstract domain. In: Proc. of the PLS. pp. 3–18 (2008)
5. Cheney, J., Ahmed, A., Acar, U.A.: Provenance as dependency analysis. In: Proc. of the ICDPL. pp. 138–152 (2007)
6. Chernikova, N.V.: Algorithm for discovering the set of all the solutions of a linear programming problem. vol. 8, pp. 282–293 (1968)
7. Cousot, P.: Web page on abstract interpretation. In: http://www.di.ens.fr/ cousot/AI/
8. Cousot, P., Cousot, R.: A gentle introduction to formal verification of computer systems by abstract interpretation. NATO Science Series III: Comp. and Syst. Sciences
9. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. of the POPL '78. pp. 84–96 (1978)

10. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. on Programming Lang. and Sys. 9(3), 319–349 (1987)
11. Guerraoui, R., Henzinger, T.A., Singh, V.: Software transactional memory on relaxed memory models. In: Computer Aided Verification. vol. 5643, pp. 321–336 (2009)
12. Halder, R., Cortesi, A.: Abstract interpretation of database query languages. Computer Languages, Systems & Structures 38, 123–157 (2012)
13. Halder, R., Cortesi, A.: Abstract program slicing of database query languages. In: Proc. of the Applied Computing. pp. 838–845. ACM (2013)
14. Halder, R., Jana, A., Cortesi, A.: Data leakage analysis of the hibernate query language on a propositional formulae domain. Trans. Large-Scale Data- and Knowledge-Centered Systems 23, 23–44 (2016)
15. Hammer, C.: Experiences with pdg-based ifc. In: Proc. of the Engineering Secure Software and Systems. pp. 44–60. Springer-Verlag, Pisa, Italy (2010)
16. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. on PLS 12(1), 26–60 (1990)
17. Jana, A., Halder, R., Chaki, N., Cortesi, A.: Policy-based slicing of hibernate query language. In: Computer Information Systems and Industrial Management, CISIM, LNCS, September 24-26, 2015. pp. 267–281. Springer (2015)
18. Jana, A., Halder, R., Cortesi, A.: Verification of hibernate query language by abstract interpretation. In: 5th International Conference on Intelligence Science and Big Data Engineering, IScIDE 2015, LNCS, June 14-16, 2015. pp. 116–128. Springer (2015)
19. Jeannet., B., Minè, A.: Apron: A library of numerical abstract domains for static analysis. In: Proc. of the Int. Conf. on CAV. pp. 661–667 (2009)
20. Johnson, A., Waye, L., Moore, S., Chong, S.: Exploring and enforcing security guarantees via program dependence graphs. In: Proc. of the 36th ACM SIGPLAN Conference on PLDI. pp. 291–302. PLDI '15, ACM (2015)
21. Kelner, J.A., Spielman, D.A.: A randomized polynomial-time simplex algorithm for linear programming. In: Proc. of the Theory of Computing. pp. 51–60. ACM (2006)
22. Larsen, L., Harrold, M.J.: Slicing object-oriented software. In: Proc. of the ICSE. pp. 495–505. IEEE CS (1996)
23. Levy, A.Y., Sagiv, Y.: Queries independent of updates. In: Proc. of the VLDB. pp. 171–181 (1993)
24. Mastroeni, I., Zanardini, D.: Data dependencies and program slicing: from syntax to abstract semantics. In: Proc. of the Partial evaluation and semantics-based program manipulation. pp. 125–134 (2008)
25. Megiddo, N.: Linear programming in linear time when the dimension is fixed. Journal of the ACM 31(1), 114–127 (1984)
26. Millstein, T., Levy, A., Friedman, M.: Query containment for data integration systems. In: Proc. of the PDS. pp. 67–75. ACM (2000)
27. Minè, A.: The octagon abstract domain. Higher Order Symbol. Comput. 19(1), 31–100 (2006 http://wwwastreeensfr/)
28. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999)
29. Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. ACM SIGPLAN Notices 19(5), 177–184 (1984)
30. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on SAC 21, 2003 (2003)
31. Sen, S., Dutta, A., Cortesi, A., Chaki, N.: A new scale for attribute dependency in large database systems. In: CISIM, LNCS, vol. 7564, pp. 266–277 (2012)
32. Tsahhirov, I., Laud, P.: Application of dependency graphs to security protocol analysis. In: Proc. of the 3rd Symposium on Trustworthy Global Computing. pp. 294–311. Sophia-Antipolis, France (Nov 5-6 2007)
33. Willmor, D., Embury, S.M., Shao, J.: Program slicing in the presence of a database state. In: Proc. of the IEEE ICSM. pp. 448–452 (2004)