Abstract Program Slicing of Database Query Languages

Raju Halder
Dept. of Comp. Sc. and Engg.
Indian Institute of Technology Patna, India

Agostino Cortesi
DAIS, Università Ca' Foscari Venezia, Italy
cortesi@unive.it

ABSTRACT

In this paper, the notions of semantic relevancy of statements, semantic data dependences and conditional dependences are extended to the case of programs embedding SQL statements in both concrete and abstract domains. This allows us to refine traditional syntax-based Database-Oriented Program Dependence Graphs, yielding to a more accurate semantics-based abstract program slicing algorithm.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program Analysis; H.2.3 [**Languages**]: Data manipulation languages (DML), Query languages

General Terms

Static Analysis, Abstract Interpretation, Databases

Keywords

Program Slicing, Query Languages, Dependence Graphs, Abstract Interpretation

1. INTRODUCTION

The notion of program slicing emerged as a useful technique that extracts from programs the statements which are relevant to a given behavior. Various applications of program slicing include program understanding, debugging, maintenance, testing, parallelization, integration, software measurement, etc. Mark Weiser [18] defines a static program slice as an executable subset of program statements that preserves the original program's behavior at a particular program point for a subset of program variables for all program inputs. This particular point of interest in the program is referred to as slicing criterion.

Most of the slicing techniques in the literature refer to imperative languages and make use (implicitly or explicitly) of the notion of Program Dependence Graph (PDG) [5, 9, 12]. However, the presence of SQL operations such as SELECT, INSERT, UPDATE or DELETE in data-intensive applications that access or manipulate databases, requires the extension of traditional PDGs into Database-Oriented Program Dependence Graphs (DOPDGs), where two additional types of dependences, called Program-Database Dependences (PD-Dependences) and Database-Database Dependences (DD-Dependences), need to be considered [19].

Program slicing can be defined in concrete as well as in abstract domains, where in the former case we consider only the exact values of program variables and database variables (or attributes), while in the latter case we consider some properties instead of their exact values [5, 10, 11, 14, 20]. A pictorial view of abstract slicing of programs embedding SQL statements is depicted in Figure 1. Observe that the sliced database *DB'* on which the slices perform their computations, in both concrete and abstract slicing, is a part of the original database *DB*.

All the slicing techniques for programs embedding SQL statements proposed so far are syntax-based [2, 17, 19]. In [5], the notion of semantic relevancy of statements has been proposed for a slicing refinement of imperative programs: statement relevancy is combined with the notions of semantic data dependences [11] and conditional dependences [16].

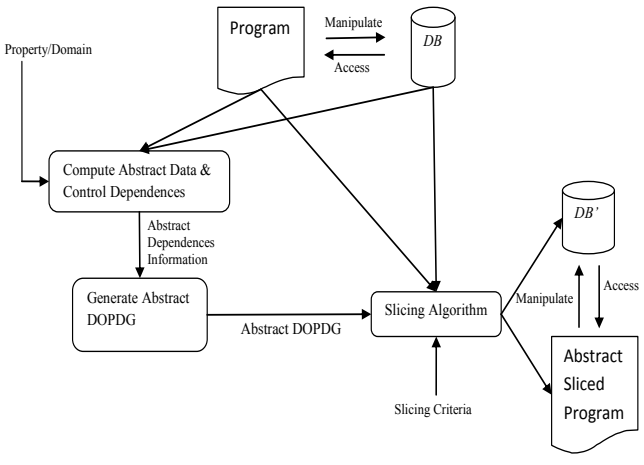

Figure 1: Abstract DOPDG-based slicing for programs embedding SQL statements

SAC'13 March 18-22, 2013, Coimbra, Portugal.

ID	name	age	locID	jobname
1	Alberto	28	2	Programmer
2	Matteo	30	1	HR
3	Francesco	35	3	Analyst

(a) Table "citizen"

locID	locname
1	Venice
2	Marghera
3	Mestre

(b) Table "loc"

jobname	category	rank	sal
Programmer	A	1	1500
Analyst	A	2	1800
Project Manager	A	3	2100
Receptionist	B	1	1000
Secretary	B	2	1100
HR	B	3	2000

(c) Table "job"

Table 1: Database dB

In this paper, we further extend this approach to the context of programs embedding SQL statements, and we propose a refinement of DOPDG-based slicing algorithms thereof. In particular, this paper provides the following contributions:

- We define syntax-based DOPDG by expressing the conditions of DD- and PD-Dependences in denotational form, as an alternative to the rules introduced by Baralis and Widom [1].
- We refine syntax-based DOPDGs into semantics-based abstract DOPDGs by combining the notions of (*i*) semantic relevancy of statements [5], (*ii*) semantic data dependences [11], and (*iii*) conditional dependences [16].

These contributions lead to an abstract program slicing algorithm for programs embedding SQL statements that strictly improves with respect to the literature.

The structure of the rest of the paper is as follows: Section 2 illustrates a motivating example. Section 3 recalls some basics on the syntax and semantics of programs embedding SQL statements. In section 4, we describe Database-Oriented Program Dependence Graphs (DOPDGs). In section 5, we lift DOPDGs from the concrete domain to an abstract domain. Section 6 provides an overall complexity analysis of the proposal. In section 7, we discuss the related works in the literature. Finally, section 8 concludes.

2. A MOTIVATING EXAMPLE

Consider the database dB in Table 1 and a portion of code P depicted in Figure 2. It is clear from the code that the employees are promoted from lower rank to higher rank belonging to same job category. Finally, the code computes the average salary of employees under each of the groups with same job category.

Suppose an auditing officer observes that the average salary of employees under job category "A" (displaying at program point 12) is out of the expected range. For instance, suppose, according to the company policy, that the average salary of job category "A" should belong to the interval [1500, 2100], while the computed result is 800. Abstract program slicing, in such case, can help to identify the program statements in P that are responsible for this error. To do so, we consider an abstract slicing criterion ⟨12, rs, Ival⟩, where "rs" is the variable of interest at program point 12 and "Ival" is the domain of intervals representing the property of "rs".

In order to compute an abstract slice of P *w.r.t.* ⟨12, rs, Ival⟩ we need to consider abstract computation of the code over

```
1.  start;
2.  Statement myStmt = DriverManager.getConnection("jdbc:mysql://200.
    210.220.1:1114/demo", "scott", "tiger").createStatement();
3.  $empID=read();
4.  $oldJob=read();
5.  $newJob=read();
6.  ResultSet rs1=myStmt.executeQuery("SELECT category, rank FROM
    job WHERE jobname=$oldJob");
7.  ResultSet rs2=myStmt.executeQuery("SELECT category, rank FROM
    job WHERE jobname=$newJob");
8.  if(rs1.next() && rs2.next()){
9.      if(rs1.getString("category").equals(rs2.getString("category")) &&
        rs2.getInt("rank") > rs1.getInt("rank")){
10.         myStmt.executeQuery("UPDATE citizen SET jobname =
            $newjob WHERE ID = $empID");}}
11. ResultSet rs = myStmt.executeQuery("SELECT avg(J.sal) FROM cit-
    izen C, job J WHERE C.jobname=J.jobname GROUP BY J.category");
12. display(rs);
13. stop;
```

Figure 2: Program P

$\widetilde{ID}$	$\widetilde{name}$	$\widetilde{age}$	$\widetilde{locID}$	$\widetilde{jobname}$
1	Alberto	28	2	TechStaff
2	Matteo	30	1	AdminStaff
3	Francesco	35	3	TechStaff

(a) Table "$\widetilde{citizen}$"

$\widetilde{locID}$	$\widetilde{locname}$
1	Venice
2	Marghera
3	Mestre

(b) Table "$\widetilde{loc}$"

$\widetilde{jobname}$	$\widetilde{category}$	$\widetilde{rank}$	$\widetilde{sal}$
TechStaff	A	[1, 3]	[1500, 2100]
AdminStaff	B	[1, 3]	[1000, 2000]

(c) Table "$\widetilde{job}$"

Table 2: Abstract Database $\widetilde{dB}$

an abstract version of the database in an abstract domain of interest[1]. An abstract database $\widetilde{dB}$ corresponding to the concrete database dB is depicted in Table 2, where some of the numeric values are abstracted by the domain of intervals and jobs are represented by the abstract domain ABSJOB={⊥, TechStaff, AdminStaff, ⊤}.

The abstract computation of $\widetilde{P}$ (which is the corresponding abstract version of P) on $\widetilde{dB}$ says that statement 10 is semantically irrelevant *w.r.t.* ABSJOB, since the update operation on jobs by statement 10 is performed within the same job category from lower to higher rank and it does not affect the property of jobs (which is represented by ABSJOB) at all. Therefore, although the statement 11 syntactically depends on the statement 10 due to the presence of "jobname" attribute in it, semantically there is no such dependence. The removal of irrelevant statement 10 also allows to disregard the statements 3 to 9 on which statement 10 depends. Therefore, the semantics-based abstract slice of P *w.r.t.* ⟨12, rs, Ival⟩

[1]The details on the abstract semantics of programs embedding SQL statements can be found in [7].

```
1.  start;
2.  Statement myStmt = DriverManager.getConnection("jdbc:mysql://200.
    210.220.1:1114/demo", "scott", "tiger").createStatement();
11. ResultSet rs = myStmt.executeQuery("SELECT avg(J.sal) FROM citizen
    C, job J WHERE C.jobname=J.jobname GROUP BY J.category");
12. display(rs);
```

(a) abstract slice *w.r.t.* ⟨12, rs, Ival⟩ of P

jobname
Programmer
HR
Analyst

Table "citizen"

jobname	category	sal
Programmer	A	1500
Analyst	A	1800
Project Manager	A	2100
Receptionist	B	1000
Secretary	B	1100
HR	B	2000

Table "job"

(b) Part of *dB* relevant to the slice

Figure 3: Abstract slice and its relevant database part

and the relevant part of the database on which the slice performs its necessary computation are shown in Figure 3(a) and 3(b) respectively. This way the computation of statement relevancy removes some false dependences between program statements, resulting into more precise abstract slices.

Expert readers may have observed that the program in Figure 2 is possibly prone to SQL injection attacks [6]. For instance, in line 4 a "DROP TABLE" command may be inserted causing disasters in line 6. If slicing is performed statically (not dynamically), the analysis of possible injection attacks can be done on sliced programs. In our example, as dangerous statements 3, 4, 5, 6, 7, 10 are removed, yielding to an overall more efficient SQL injection attack analysis.

3. PROGRAMS EMBEDDING SQL STATEMENTS

In this section, we recall from [7] some basic notions that will be used in the rest of the paper.

Let e, t and f be an expression, a database table and a function respectively. We use the following functions in the subsequent sections:

- *const(e)* returns the constants appearing in e.
- *var(e)* returns the variables appearing in e.
- *attr(t)* returns the attributes associated with t.
- *dom(f)* returns the domain of f.
- *target(f)* returns a subset of dom(f) on which the application of f is restricted.

An application embedding SQL statements basically interacts with two worlds: user-world and database-world. Corresponding to these two worlds there exist two sets of variables: database variables $\mathbb{V}_d$ and application variables $\mathbb{V}_a$. Variables from $\mathbb{V}_d$ are involved only in the SQL statements, whereas variables in $\mathbb{V}_a$ may occur in all types of instructions of the application. Any SQL statement Q is denoted by a tuple $Q = \langle A, \phi \rangle$ where A and ϕ refer to *action part* and *pre-condition part* of Q respectively. A SQL statement Q first identifies an active data set from the database using the pre-condition ϕ, and then performs the appropriate operations on that data set using the SQL action A. The pre-condition ϕ appears in SQL statements as a well-formed formula in first-order logic. Table 3 depicts the syntactic sets, the abstract syntax of programs embedding SQL statements, the environments and states associated with programs, and the semantics of SQL statements. Observe that all the syntactic elements in SQL statements (for example, GROUP BY, ORDER BY, DISTINCT clauses, etc) are represented as functions and the semantics is described as a partial functions on the states which specify how expressions are evaluated and instructions are executed. A state in the program is represented by the tuple (I, ρ_d, ρ_a) where $I \in \mathbb{I}$ is a program statement, $\rho_d \in \mathfrak{E}_d$ is a database environment and $\rho_a \in \mathfrak{E}_a$ is an application environment. For more detail, please refer to [7].

We define the following functions that extract from statements the sets of database and application variables involved in those statements:

$$\mathtt{USE}^d_v : \mathbb{I} \to \mathbb{V}_d \qquad \mathtt{USE}^a_v : \mathbb{I} \to \mathbb{V}_a \qquad \mathtt{USE}_v : \mathbb{I} \to \mathbb{V}_d \cup \mathbb{V}_a$$

$$\mathtt{DEF}^d_v : \mathbb{I} \to \mathbb{V}_d \qquad \mathtt{DEF}^a_v : \mathbb{I} \to \mathbb{V}_a \qquad \mathtt{DEF}_v : \mathbb{I} \to \mathbb{V}_d \cup \mathbb{V}_a$$

Therefore, given a SQL statement $Q \in \mathbb{I}$, the sets of used and defined variables in it are as follows:

$$\mathtt{USE}_v(Q) = \mathtt{USE}^d_v(Q) \cup \mathtt{USE}^a_v(Q) \quad \mathtt{DEF}_v(Q) = \mathtt{DEF}^d_v(Q) \cup \mathtt{DEF}^a_v(Q)$$

Any imperative statement $I \in \mathbb{I}$ does not involve database variables, and therefore, $\mathtt{USE}^d_v(I) = \emptyset = \mathtt{DEF}^d_v(I)$. Table 4 depicts the sets of variables that are defined and used by various SQL statements based on the syntactic presence in the statements.

4. DATABASE-ORIENTED PROGRAM DEPENDENCE GRAPHS

Willmor et al. [19] introduced a variant of the program dependence graph, known as Database-Oriented Program Dependence Graph (DOPDG), by considering two additional types of data dependences: Program-Database Dependences (PD-Dependences) and Database-Database Dependences (DD-Dependences).

A PD-Dependence arises between a SQL statement and an imperative statement where either the database state defined by SQL statement is used by the imperative statement or the data defined by imperative statement is used by the SQL statement. A DD-Dependence arises between two SQL statements where the database state defined by one SQL statement is used by the other SQL statement. The data dependences between imperative statements are similar as in the case of traditional PDGs.

In order to obtain finer slices, [19] addressed the issue of false DD-Dependences that occur when a part of the database state defined by one SQL statement does not overlap with the part of the database state used by the other. It applies the Condition-Action rules introduced by Baralis and Widom [1] that determine when an action of one rule can affect the condition of another rule.

In the subsequent sections, we express DD-Dependences and PD-Dependences using the conditions in denotational form which are semantically equivalent to the rules introduced by Baralis and Widom [1]. Then, we lift these dependences to an abstract domain of interest, and this way we may apply the notion of semantic relevancy of statements [5], semantic data dependences [11] and conditional dependences [16], to the case of programs embedding SQL statements.

4.1 Identifying DD-Dependences

The data dependence between two imperative statements I_1 and I_2 for a data x is denoted by $I_1 \overset{x}{\to} I_2$. Similarly, we

Syntactic Sets
$n : \mathbb{Z}$ (Integer)
$k : \mathbb{S}$ (String)
$c : \mathbb{C}$ (Constants)
$v_a : \mathbb{V}_a$ (Application Variables)
$v_d : \mathbb{V}_d$ (Database Variables)
$v : \mathbb{V} \triangleq \mathbb{V}_d \cup \mathbb{V}_a$ (Variables)
$e : \mathbb{E}$ (Arithmetic Expressions)
$b : \mathbb{B}$ (Boolean Expressions)
$A : \mathbb{A}$ (Action)
$\tau : \mathbb{T}$ (Terms)
$a_f : \mathbb{A}_f$ (Atomic Formulas)
$\phi : \mathbb{W}$ (Pre-condition)
$Q : \mathbb{Q}$ (SQL statements)
$I : \mathbb{I}$ (Program statements)

Abstract Syntax
$c ::= n \mid k$
$e ::= c \mid v_d \mid v_a \mid op_u\ e \mid e_1\ op_b\ e_2$, where $op_u \in \{+, -\}$ and $op_b \in \{+, -, *, /, \ldots ...\}$
$b ::= e_1\ op_r\ e_2 \mid \neg b \mid b_1 \vee b_2 \mid b_1 \wedge b_2 \mid true \mid false$, where $op_r \in \{=, \geq, \leq, <, >, \neq, \ldots\}$
$\tau ::= c \mid v_a \mid v_d \mid f_n(\tau_1, \tau_2, ..., \tau_n)$, where f_n is an n-ary function.
$a_f ::= R_n(\tau_1, \tau_2, ..., \tau_n) \mid \tau_1 = \tau_2$, where $R_n(\tau_1, \tau_2, ..., \tau_n) \in \{true, false\}$
$\phi ::= a_f \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \forall x_i\ \phi \mid \exists x_i\ \phi$
$g(\vec{e}) ::= \texttt{GROUP BY}(\vec{e}) \mid id$
$r ::= \texttt{DISTINCT} \mid \texttt{ALL}$
$s ::= \texttt{AVG} \mid \texttt{SUM} \mid \texttt{MAX} \mid \texttt{MIN} \mid \texttt{COUNT}$
$h(e) ::= s \circ r(e) \mid \texttt{DISTINCT}(e) \mid id$
$h(*) ::= \texttt{COUNT}(*)$
$\vec{h}(\vec{x}) ::= \langle h_1(x_1), ..., h_n(x_n)\rangle$, where $\vec{h} = \langle h_1, ..., h_n\rangle$ and $\vec{x} = \langle x_1, ..., x_n\rangle$
$f(\vec{e}) ::= \texttt{ORDER BY ASC}(\vec{e}) \mid \texttt{ORDER BY DESC}(\vec{e}) \mid id$
$A ::= select(v_a,\ f(\vec{e}),\ r(\vec{h}(\vec{x})),\ \phi,\ g(\vec{e})) \mid update(\vec{v_d},\ \vec{e}) \mid insert(\vec{v_d},\ \vec{e}) \mid delete(\vec{v_d})$
$Q ::= \langle A, \phi\rangle \mid Q'\ \texttt{UNION}\ Q'' \mid Q'\ \texttt{INTERSECT}\ Q'' \mid Q'\ \texttt{MINUS}\ Q''$
$I ::= skip \mid v_a := e \mid v_a :=? \mid Q \mid if\ b\ then\ I_1\ else\ I_2 \mid while\ b\ do\ I \mid I_1; I_2$

Application Environment $\rho_a : \mathbb{V}_a \nrightarrow \mathfrak{D}_\mho$ where $\mathfrak{D}_\mho$ is domain of $\mathbb{V}_a$
Database Environment A database is a set of tables $\{t_i \mid i \in I_x\}$ for a given set of indexes I_x.
We define database environment as a function ρ_d whose domain is I_x, such that for $i \in I_x$, $\rho_d(i) = t_i$.
Table Environment A table environment ρ_t for a table t is defined as a function such that $\forall a_i \in attr(t)$, $\rho_t(a_i) = \langle \pi_i(l_j) \mid l_j \in t\rangle$
where π is the projection operator, i.e., $\pi_i(l_j)$ is the i^{th} element of the l_j-th row.
State The set of states is defined as $\mathfrak{S} \triangleq \mathbb{I} \times \mathfrak{E}_d \times \mathfrak{E}_a$
where $\mathfrak{E}_d$ and $\mathfrak{E}_a$ denote the set of all database environments and the set of all application environments respectively.
Semantics The semantics of SQL statement Q is defined as $S[\![Q]\!](\rho_d, \rho_a) = S[\![Q]\!](\rho_t, \rho_a) = (\rho_{t'}, \rho_{a'})$
where $target(Q) = t$ and $S[\![.]\!]$ is a semantic function.

Table 3: Syntax and semantics of programs embedding SQL statements

SQL statements	Defined/Used variables
Q_{sel}=$\langle A_{sel}, \phi\rangle$	$\texttt{DEF}_v(Q_{sel}) = v_a$
$= \langle select(v_a,\ f(\vec{e}),\ r(\vec{h}(\vec{x})),\ \phi_1,\ g(\vec{e})), \phi\rangle$	$\texttt{USE}_v(Q_{sel}) = var(\phi) \cup var(\vec{e}) \cup var(\phi_1) \cup var(\vec{x}) \cup var(\vec{e'})$
Q_{ins}=$\langle A_{ins}, \phi\rangle$	$\texttt{DEF}_v(Q_{ins}) = var(\vec{v_d})$
$= \langle insert(\vec{v_d},\ \vec{e}), true\rangle$	$\texttt{USE}_v(Q_{ins}) = var(\vec{e})$
Q_{upd}=$\langle A_{update}, \phi\rangle$	$\texttt{DEF}_v(C_{update}) = var(\vec{v_d})$
$= \langle update(\vec{v_d},\ \vec{e}), \phi\rangle$	$\texttt{USE}_v(C_{update}) = var(\vec{e}) \cup var(\phi)$
Q_{del}=$\langle A_{del}, \phi\rangle$	$\texttt{DEF}_v(Q_{del}) = var(\vec{v_d})$
$= \langle delete(\vec{v_d}), \phi\rangle$	$\texttt{USE}_v(Q_{del}) = var(\phi)$

Table 4: Sets of variables defined and used by SQL statements

denote a DD-Dependence between two SQL statements Q_1 and Q_2 by $Q_1 \xrightarrow{\Upsilon} Q_2$, where Υ is the part of the database information that is defined by Q_1 and subsequently used by Q_2. Below we describe how to determine Υ.

Consider a SQL statement $Q = \langle A, \phi\rangle$ where $target(Q) = t$, $\texttt{USE}^d_v(Q) = \vec{a}_{use} \subseteq attr(t)$ and $\texttt{DEF}^d_v(Q) = \vec{a}_{def} \subseteq attr(t)$. Suppose, according to the denotational semantics of Q, that $S[\![Q]\!](\rho_t, \rho_a) = (\rho_{t'}, \rho_{a'})$ where S is a semantic function [7]. We define two functions $\mathfrak{A}_{use}$ and $\mathfrak{A}_{def}$ as follows:

$$\mathfrak{A}_{use}(Q, t) = \rho_{t\downarrow\phi}(\vec{a}_{use}) \quad (1)$$

$$\mathfrak{A}_{def}(Q, t) = \Delta(\rho_{t'}(\vec{a}_{def}), \rho_t(\vec{a}_{def})) \quad (2)$$

Given a SQL statement Q and its target table t, the function $\mathfrak{A}_{use}$ maps to the part of the database information accessed or used by Q depicted in Figure 4. We denote by the notation $(t \downarrow \phi)$ the set of tuples in t for which ϕ holds "*true*". The function $\mathfrak{A}_{def}$ defines the changes occurred in the database states when data is updated or deleted or inserted by SQL statements. Δ computes the differences between the original database states on which SQL statements operate and the new database states obtained after performing the SQL actions.

By following equations 1 and 2, we can compute the part of the database information Υ that is updated or deleted or inserted by $Q_1 = \langle A_1, \phi_1\rangle$ and subsequently used by $Q_2 = \langle A_2, \phi_2\rangle$ as follows:

$$\Upsilon = \mathfrak{A}_{use}(Q_2, t') \cap \mathfrak{A}_{def}(Q_1, t) \quad (3)$$

where $target(Q_1) = t$ and $S[\![Q_1]\!](\rho_t, \rho_a) = (\rho_{t'}, \rho_a)$ and $target(Q_2)$

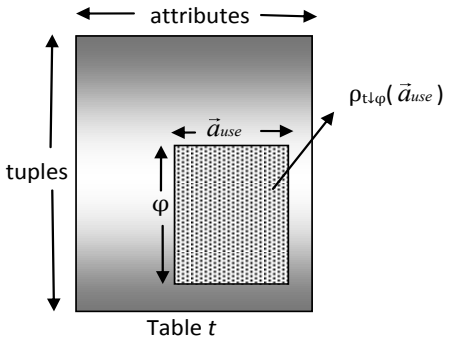

Figure 4: Part of database state used by $Q = \langle A, \phi\rangle$

$= t'$. Observe that Q_1 can only be UPDATE or INSERT or DELETE statements and it does not change the application environment at all. In case of SELECT statement, $\Upsilon = \emptyset$ because $\mathfrak{A}_{def}(Q_{sel}, t) = \emptyset$ where Q_{sel} is a SELECT statement with $target(Q_{sel}) = t$. Therefore, there exist no DD-Dependence on SELECT statements.

Definition 1 (DD-Dependency). *Consider a SQL statement $Q_1 = \langle A_1, \phi_1\rangle$ with $target(Q_1) = t$ such that $S[\![Q_1]\!](\rho_t, \rho_a) = (\rho_{t'}, \rho_a)$. The SQL statement $Q_2 = \langle A_2, \phi_2\rangle$ with $target(Q_2) = t'$ is DD-Dependent on Q_1 for Υ (denoted $Q_1 \xrightarrow{\Upsilon} Q_2$) if $Q_1 \in \{Q_{upd}, Q_{ins}, Q_{del}\}$ and $\Upsilon = \mathfrak{A}_{use}(Q_2, t') \cap \mathfrak{A}_{def}(Q_1, t) \neq \emptyset$.*

4.2 Identifying PD-Dependences

An imperative statement I is PD-Dependent on a SQL statement Q for some variable x (denoted $Q \xrightarrow{x} I$) if I uses x whose value is obtained by Q from the database and there is an x-definition-free path from Q to I. For instance, consider the following java code interacting with database *dB* (Table 1) where the statements 6 and 7 are PD-Dependent on the SELECT statement in 5 for the variable x:

```
4.  ....
5.  ResultSet x = myStmt.executeQuery("SELECT name, age FROM citizen
    WHERE sal≥2000");
6.  while ( x.next() ) {
7.        System.out.println(x.getString("name")+x.getString("age"));}
8.  ....
```

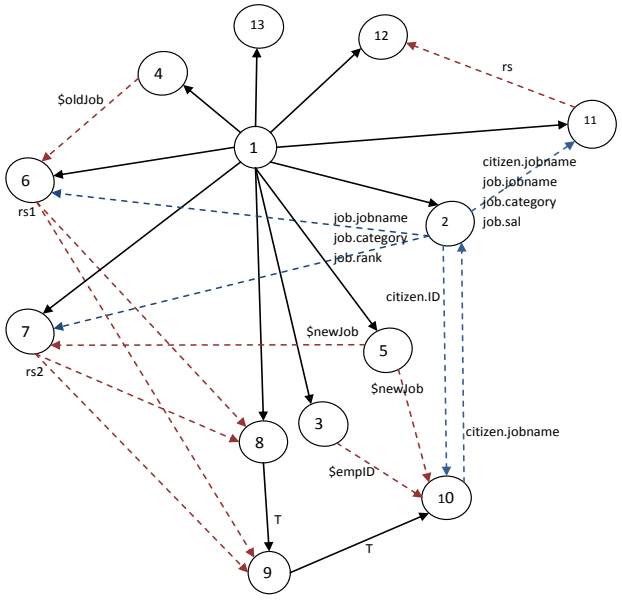

Figure 5: DOPDG of the code P

Observe that the only SQL statements that are involved in PD-Dependences are SELECT statements which are able to define the value of x by retrieving information from the databases.

Consider a SELECT statement $Q_{sel} = \langle select(v_a, f(\vec{e'}), r(\vec{h}(\vec{x})), \phi_1, g(\vec{e})), \phi\rangle$ with $target(Q_{sel}) = t$. From table 4, we get $\texttt{DEF}_v(Q_{sel}) = v_a$ where v_a is a resultset type of application variable. According to the semantics of Q_{sel}, we get $S[\![Q_{sel}]\!](\rho_t, \rho_a) = (\rho_t, \rho_{a'})$. Therefore, the data for which an imperative statement is PD-Dependent on Q_{sel} is $\rho_{a'}(v_a)$.

On the other hand, a SQL statement Q is PD-Dependent on an imperative statement I for some variable x (denoted $I \xrightarrow{x} Q$) if Q uses x whose value is defined by I and there is an x-definition-free path from I to Q. In this case, Q can be either SELECT or UPDATE or INSERT or DELETE.

4.3 Constructing Concrete DOPDGs

Figure 5 depicts the syntax-based DOPDG for the program P (Figure 2), where by the dotted lines we denote PD-/DD-Dependences and by solid lines we denote control dependences. The computation of data dependences between imperative statements and the computation of control dependences are similar as in the case of traditional PDGs [12].

Observe that we insert DD-Dependence edges from the nodes that define database states partially to the node that acts as the database source. For instance, in Figure 5 we insert a DD-Dependence edge from node 10 to node 2, where the node 10 is an UPDATE statement (that defines the database partially) and the node 2 is treated as the source of the database as it makes a connection to the database. In contrast, when a SQL statement defines the whole database information, the corresponding node in the DOPDG will then be treated as a new source of database information for the subsequent nodes.

5. CONSTRUCTING ABSTRACT DOPDGS

In this section, we lift DOPDGs from the concrete domain to an abstract domain by defining DD- and PD-Dependences in the abstract domain, and we apply the notions of semantic relevancy, semantic data dependences and conditional dependences.

Definition 2 (Abstract Databases). *Let dB be a database. The database $\widetilde{dB} = \alpha(dB)$ where α is the abstraction function, is said to be an abstract version of dB if there exist a representation function γ, called concretization function, such that for all tuple $\langle x_1, x_2, \ldots, x_n\rangle \in dB$ there exist a tuple $\langle y_1, y_2, \ldots, y_n\rangle \in \widetilde{dB}$ such that $\forall i \in [1 \ldots n]$ $(x_i \in id(y_i) \vee x_i \in \gamma(y_i))$.*

5.1 Abstract DD-Dependences.

Consider an abstract domain σ. Let $\widetilde{Q} = \langle \widetilde{A}, \widetilde{\phi}\rangle$ be an abstract SQL statement with $target(\widetilde{Q}) = \widetilde{t}$ where $\widetilde{t}$ is an abstract table. Suppose, according to the abstract semantics, that $\widetilde{S}[\![\widetilde{Q}]\!](\rho_{\tilde{t}}, \rho_{\tilde{a}}) = (\rho_{\tilde{t}'}, \rho_{\tilde{a}'})$ where $\widetilde{S}$ is an abstract semantic function. We define the abstract functions $\widetilde{\mathfrak{A}}_{use}$ and $\widetilde{\mathfrak{A}}_{def}$ corresponding to $\mathfrak{A}_{def}$ and $\mathfrak{A}_{use}$ respectively as follows:

$$\widetilde{\mathfrak{A}}_{use}(\widetilde{Q}, \widetilde{t}) = \rho_{\tilde{t}} \downarrow_T \tilde{\phi}(\vec{\tilde{a}}_{use}) \cup \rho_{\tilde{t}} \downarrow_U \tilde{\phi}(\vec{\tilde{a}}_{use}) \tag{4}$$

$$\widetilde{\mathfrak{A}}_{def}(\widetilde{Q}, \widetilde{t}) = \widetilde{\Delta}(\rho_{\tilde{t}'}(\vec{\tilde{a}}_{def}), \rho_{\tilde{t}}(\vec{\tilde{a}}_{def})) \tag{5}$$

Observe that the notations $\tilde{t} \downarrow_T \tilde{\phi}$ and $\tilde{t} \downarrow_U \tilde{\phi}$ denote the set of abstract tuples in $\widetilde{t}$ for which $\widetilde{\phi}$ yields to "*true*" and "*unknown*" respectively, in order to provide sound approximations in the abstract domain σ. The abstract evaluation of $\widetilde{\phi}$ over the abstract tuples in $\widetilde{t}$ results into the logic value "*unknown*" in three-valued logic due to the imprecision introduced in the abstract domain.

Definition 3. *Let ϱ be an operation on a database table t. Let (α, γ) be a Galois Connection. Let $\widetilde{t}$ be an abstract table corresponding to t. An abstract operation $\widetilde{\varrho}$ on $\widetilde{t}$ is sound w.r.t. ϱ if $\varrho(t) \in \gamma(\widetilde{\varrho}(\widetilde{t}))$.*

From the abstract definition of $\widetilde{\mathfrak{A}}_{use}$ and $\widetilde{\mathfrak{A}}_{def}$, we can express the result into two distinct parts: *yes*-part and *may*-part, as follows:

$$\widetilde{\mathfrak{A}}_{use}(\widetilde{Q}, \widetilde{t}) = \xi^{use} = \langle \xi^{use}_{yes}, \xi^{use}_{may}\rangle \text{ and } \widetilde{\mathfrak{A}}_{def}(\widetilde{Q}, \widetilde{t}) = \xi^{def} = \langle \xi^{def}_{yes}, \xi^{def}_{may}\rangle$$

where $\xi^{use}_{yes} \cap \xi^{use}_{may} = \emptyset$ and $\xi^{def}_{yes} \cap \xi^{def}_{may} = \emptyset$. The *yes*-part defines the part of the result for which $\widetilde{\phi}$ evaluates to "*true*", whereas the *may*-part defines the remaining part of the result for which $\widetilde{\phi}$ evaluates to "*unknown*". Observe that in case of abstract INSERT statement, the *may*-part $\xi^{def}_{may} = \emptyset$.

Given two abstract SQL statements $\widetilde{Q}_1$ and $\widetilde{Q}_2$ where $target(\widetilde{Q}_1) = \widetilde{t}$ and $S[\![\widetilde{Q}_1]\!](\rho_{\tilde{t}}, \rho_{\tilde{a}}) = (\rho_{\tilde{t}'}, \rho_{\tilde{a}})$ such that $target(\widetilde{Q}_2) = \widetilde{t'}$. The part of the abstract database information $\widetilde{\Upsilon}$ defined by $\widetilde{Q}_1$ and subsequently used by $\widetilde{Q}_2$ is computed as follows:

$$\begin{aligned}\widetilde{\Upsilon} =& \widetilde{\mathfrak{A}}_{use}(\widetilde{Q}_2, \widetilde{t'}) \,\widetilde{\cap}\, \widetilde{\mathfrak{A}}_{def}(\widetilde{Q}_1, \widetilde{t})\\ =& \xi^{use} \,\widetilde{\cap}\, \xi^{def} = \langle \xi^{use}_{yes}, \xi^{use}_{may}\rangle \,\widetilde{\cap}\, \langle \xi^{def}_{yes}, \xi^{def}_{may}\rangle\\ =& \langle (\xi^{use}_{yes} \cap \xi^{def}_{yes}), ((\xi^{use}_{may} \cap \xi^{def}) \cup (\xi^{def}_{may} \cap \xi^{use}))\rangle\end{aligned}$$

The soundness of $\widetilde{Q}_2(\widetilde{Q}_1(\widetilde{t}))$ guarantees the soundness of abstract DD-Dependences in an abstract domain of interest. The three conditions for full, partial and non DD-dependency for $\widetilde{Q}_1 \in \{\widetilde{Q}_{upd}, \widetilde{Q}_{ins}, \widetilde{Q}_{del}\}$ and $\widetilde{Q}_2 \in \{\widetilde{Q}_{sel}, \widetilde{Q}_{upd}, \widetilde{Q}_{ins}, \widetilde{Q}_{del}\}$ can be expressed similarly as defined in the concrete domain.

5.2 Abstract PD-Dependences

An abstract PD-Dependence is denoted by either $\widetilde{I} \xrightarrow{\widetilde{x}} \widetilde{Q}$ or $\widetilde{Q}_{sel} \xrightarrow{\widetilde{x}} \widetilde{I}$, where $\widetilde{I}$, $\widetilde{Q}$ and $\widetilde{Q}_{sel}$ represent abstract imperative statement, abstract SQL statement and abstract SELECT statement respectively. Given an abstract domain σ, the abstract SELECT statement $\widetilde{Q}_{sel}$ is always sound *w.r.t.* its concrete counterpart Q_{sel} [7]. The soundness of $\widetilde{Q}_{sel}$ guarantees the soundness of the abstract PD-Dependences.

5.3 Semantics-based Dependence Computation

In this subsection, we show how to integrate the three notions of semantic computations to the context of programs embedding SQL statements.

Semantic Relevancy of SQL statements.

When the execution of a statement does not change a property of the states possibly occurring at that statement, the statement is referred to as semantically irrelevant *w.r.t.* that property.

Definition 4 (Concrete Semantic Relevancy of SQL). *Consider a concrete property $\omega = \langle \omega_{db}, \omega_{ap} \rangle$, where ω_{db} and ω_{ap} are concrete properties on the database variables and on the application variables respectively. A SQL statement Q at program point p is called semantically irrelevant w.r.t. ω, if for all states $\nu = (\rho_t, \rho_a)$ possibly appearing at p the execution of Q on ν results into a state $\nu' = (\rho_{t'}, \rho_{a'})$ such that ν and ν' are equivalent w.r.t. ω, i.e. $\omega(\nu) \equiv \omega(\nu')$ or $\omega(\rho_t, \rho_a) \equiv \omega(\rho_{t'}, \rho_{a'})$ or $\omega_{db}(\rho_t) \equiv \omega_{db}(\rho_{t'}) \wedge \omega_{ap}(\rho_a) \equiv \omega_{ap}(\rho_{a'})$.*

Example 1. *Given a database containing salary information of the employees. Consider a concrete database property ω_{db} that expresses a constraint of gross salary on basic salary, defined as: $gross\ salary = \frac{(165 \times basic\ \ salary)}{100} + 200$. Let the initial database state satisfies ω_{db}. Let Q_1 and Q_2 be two SQL statements in a program. Suppose Q_1 updates basic salary, whereas Q_2 follows the above equation and updates the gross salary of employees. We say that Q_1 is semantically relevant w.r.t. ω_{db} as it changes the initial database state into a new state that does not satisfy ω_{db}. On the other hand, the entire block $\{Q_1; Q_2\}$ is semantically irrelevant w.r.t. ω_{db}, because its execution results into a state that preserves ω_{db}.*

Definition 5 (Abstract Semantic Relevancy of SQL). *Given an abstract domain σ [2]. A SQL statement $\widetilde{Q}$ with target($\widetilde{Q}$) = $\widetilde{t}$ at program point p is called semantically irrelevant w.r.t. σ, if for all abstract states $\epsilon = (\rho_{\widetilde{t}}, \rho_{\widetilde{a}})$ possibly appearing at p the execution of $\widetilde{Q}$ on ϵ results into an abstract state $\epsilon' = (\rho_{\widetilde{t}'}, \rho_{\widetilde{a}'})$ such that they are equivalent, i.e., $\epsilon \equiv \epsilon'$ w.r.t. σ.*

Example 2. *Table 2 depicts an abstract database where some of the numeric values are represented by the domain of intervals and jobs are represented by the abstract domain ABSJOB={$\bot$, TechStaff, AdminStaff, $\top$}. Since the execution of the statement 10 in P (Figure 2) does not change the property of job information, we say that statement 10 is semantically irrelevant w.r.t. ABSJOB.*

The notion of semantic relevancy of statements allows us to refine syntax-based DOPDGs into more precise semantics-based DOPDGs by removing the nodes corresponding to the irrelevant statements. An example is the removal of node 10 from the DOPDG of Figure 5.

[2]For the sake of simplicity, we assume here that the attributes are of the same type, and that the property σ is the same for all attributes. The definition can easily be generalized to the case of lists of properties related to corresponding attributes.

Semantic Data Dependences.

Mastroeni and Zanardini [11] introduced the notion of semantic data dependences. 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. This can also be lifted to an abstract setting where dependences are computed with respect to some specific properties of interest rather than concrete values. For instance, if we consider the abstract domain SIGN={$\top$, pos, neg, $\bot$}, the expression e does not semantically depend on x *w.r.t.* SIGN, as the abstract evaluation of x^2 always yields to "pos" for all atomic values of $x \in$ {pos, neg}. We can apply this notion to the case of programs embedding SQL statements to determine whether the presence of variables in an expression semantically affect the evaluation of the expression. Observe that expressions in SQL statements contain application variables as well as database attributes. The semantic data dependence computation for application variables appearing in the expressions has already been discussed in [11]. We can extend this to the case of database attributes appearing in the expressions, resulting into the removal of corresponding PD- or DD-Dependences as well as the database information corresponding to these irrelevant attributes from the sliced database.

Example 3. *Consider the following code fragment, where statement 5 adds a new column to a table t and sets its default value to 0.*

```
4.  ....
5.  ALTER TABLE t ADD COLUMN newcol int(10) NOT NULL DEFAULT 0;
6.  ResultSet rs = myStmt.executeQuery("SELECT oldcol+newcol FROM t");
7.  while ( rs.next() ) {
8.        System.out.println(rs.getString(1));}
9.  ....
```

Observe that the expression "oldcol+newcol" in the select statement at program point 6 does not semantically depend on the database variable "newcol", as the variable "newcol" does not affect the result of the expression for any states possibly reaching program point 6. Therefore, we can remove the corresponding DD-Dependence edge $5 \xrightarrow{newcol} 6$ from the syntactic DOPDG. Also the database information corresponding to the attribute "newcol" does not appear in the resultant sliced database.

A DOPDG refinement algorithm **"REFINE-DOPDG"** that applies the notions of statement relevancy and semantic data dependences, can easily be formalized in a straightforward way.

Conditional Dependences.

The notion of Dependence Condition Graphs (DCGs) is introduced by Sukumaran et al. in [16]. A DCG is built from a PDG by annotating each edge $e = e.src \rightarrow e.tgt$ in the PDG with the information $e^b = \langle e^R, e^A \rangle$. The first component e^R refers to *Reach Sequences* that represents the conditions required to reach the target $e.tgt$ from the source $e.src$, whereas the second component e^A refers to *Avoid Sequences* that captures the possible conditions under which the assignment at $e.src$ can get over-written before it reaches $e.tgt$. An execution ψ is said to satisfy e^b for a data dependence edge e if it satisfies all the conditions in e^R and at the same time it avoids the conditions represented by e^A; this means that the execution ψ

Algorithm 1: REFINE-DODCG

Input: Syntax-based DODCG G_{dodcg} and an abstract domain σ
Output: Semantics-based abstract DODCG G^{s}_{dodcg} *w.r.t.* σ

1. FOR each node $q \in G_{dodcg}$ DO
2. If $\forall\psi$: $\psi \not\kappa^{\sigma} (p \xrightarrow{lab} q)$ where $lab \in \{true, false\}$ THEN
3. Remove from G_{dodcg} the node q and all its associated dependences. If q is a control node, the removal of q also removes all the nodes transitively control-dependent on it. Data dependences have to be re-adjusted accordingly;
4. END IF
5. FOR each data dependence edge $e = (q \xrightarrow{x} p_i)$ DO
6. IF $\forall\psi$: $\psi \not\kappa^{\sigma} e$ THEN
7. Remove e from G_{dodcg} and re-adjust the data dependence of p_i for the data x;
8. END IF
9. END FOR
10. FLAG:=true;
11. FOR each ϕ-sequences $\eta_{\phi} = (q \xrightarrow{x} \phi_1 \xrightarrow{x} \ldots \xrightarrow{x} \phi_j \xrightarrow{x} p_i)$ starting from q DO
12. IF $\exists\psi$: $\psi \vdash^{\sigma} \eta_{\phi}$ THEN
13. FLAG:=false;
14. BREAK;
15. END IF
16. END FOR
17. IF FLAG==true THEN
18. Remove the edge $q \xrightarrow{x} \phi_1$;
19. END IF
20. END FOR

Figure 6: Algorithm to generate Semantics-based Abstract DODCG

ensures that the value computed at *e.src* successfully reaches *e.tgt*. It is worthwhile to note that e^{A} is relevant only for data dependence edges and is $\emptyset$ for control edges.

Given an abstract Database-Oriented Program Dependence Graph (DOPDG), we can convert it into an abstract Database-Oriented Dependence Condition Graph (DODCG) by computing annotations e^{b} over all dependence edges e, by following similar steps as in the case of PDG-to-DCG conversion [16].

The computation of semantically unrealizable dependence paths (see Definition 6) in an abstract DODCG under abstract trace semantics of the program removes possible false dependences from the abstract DODCG [5].

Definition 6 (Unrealizable Dependence Path). *Given a DODCG G_{dodcg} and an abstract domain σ. A dependence path $\eta \in G_{dodcg}$ is called semantically unrealizable in the abstract domain σ if $\forall\psi$: $\psi \not\kappa^{\sigma} \eta$, where ψ is an abstract execution trace.*

The algorithm **"REFINE-DODCG"** refines a DODCG into more precise one by removing false dependence paths that are unrealizable under its trace semantics. The algorithm is depicted in Figure 6. The reader now is in position to fully understand the complete refined slicing algorithm **"GEN-SLICE"** for programs embedding SQL statements, depicted in Figure 7.

6. COMPLEXITY ANALYSIS

Given an abstract domain σ and a slicing criteria $C_r = \langle p, v, \sigma\rangle$, the overall computational complexity depends on the following:

1. Complexity in computing statement relevancy *w.r.t.* σ which is $O(n^2 v\gamma h)$, where n = program size, v = number of tuples present in the database, γ = maximum number of abstract operations present in a statement, and h = height of the abstract lattice.
2. Complexity in computing semantic data dependences *w.r.t.* σ which is $O(a^2\beta mn)$, where a = number of atomic values in the abstract domain, β = maximum number of abstract operations present in an expression, m = total number of variables in the program.
3. Complexity to generate semantics-based abstract DODCG *w.r.t.* σ and its slicing *w.r.t.* C_r which is $O(n^3 h)$.

We may assume that the number of atomic values (a) present in the abstract domain is constant, and that $O(\gamma) = O(\beta) = O(m) = O(n)$. Therefore, the overall complexity of our abstract slicing, in case of finite height abstract lattices, is $O(n^3 vh)$). This complexity is comparable with the complexity of usual dataflow analysis (like, liveness etc.) and also in this case, in the practice, we can get a quadratic complexity, which is acceptable for its practical adoption.

Algorithm 2: GEN-SLICE

Input: Program P embedding SQL statements and an abstract slicing criterion $\langle p, v, \sigma\rangle$
Output: Abstract Slice *w.r.t.* $\langle p, v, \sigma\rangle$

1. Generate a semantics-based abstract DOPDG $G^{r,d}_{dopdg}$ from the program P by following the algorithm REFINE-DOPDG.
2. Convert $G^{r,d}_{dopdg}$ into the corresponding DODCG G_{dodcg} by computing annotations over all the data/control edges of it.
3. Generate a semantics-based abstract DODCG G^{s}_{dodcg} from G_{dodcg} by following the algorithm REFINE-DODCG.
4. Apply the criterion $\langle p, v\rangle$ on G^{s}_{dodcg} by following PDG-based slicing techniques [12] and generate a sub-DODCG G_{sdodcg} that includes the node corresponding to the program point p as well.
5. Refine G_{sdodcg} into more precise one G'_{sdodcg} by performing the following operation for all nodes $q \in G_{sdodcg}$:

   $\forall\eta_{\phi} = (q \xrightarrow{x} \phi_1 \xrightarrow{x} \ldots \xrightarrow{x} \phi_j \xrightarrow{x} p_i)$ and $\forall\psi$: if $\psi \not\kappa^{\sigma} \eta_{\phi}$, then remove the edge $q \xrightarrow{x} \phi_1$ from G_{sdodcg}.
6. Apply again the criterion $\langle p, v\rangle$ on G'_{sdodcg} that results into the desired slice.

Figure 7: Slicing Algorithm

7. RELATED WORK

Mastroeni and Zanardini [11] first introduced the notion of semantic data independences at expression-level that helps to generate more precise semantics-based PDGs by removing some false data dependences from the traditional syntactic PDGs. This is the basis to design an abstract semantics-based slicing algorithms aimed at identifying the part of the programs which is relevant with respect to a property (not necessarily the exact values) of the variables at a given program point. As an extension, recently in [5] we applied the notion of semantic relevancy of statements, and proposed a slicing refinement algorithm by combining with it the notions of semantics data dependences [11] and conditional dependences [16]. This allows us to transform the semantics-based (abstract) PDG into a semantics-based (abstract) Dependence Condition Graph that enables to identify the conditions for dependence between program points. Zanardini [20] also introduced a similar notion of invariance of statements based

on the notion "agreement", i.e., the set of conditions that are preserved at a given program point. He defined an abstract slicing framework that includes the possibility to restrict the input states of the program, in the style of abstract conditioned slicing, thus lying between static and dynamic slicing.

All the slicing techniques mentioned above refer to imperative languages. They do not take into account the presence of the additional forms of states associated with programs, such as reading from the standard input stream or accessing/manipulating data from external databases.

Sivagurunathan et.al. [15] first addressed this and introduced pseudo variables into the program to make the hidden I/O state accessible to the slicer. Tan and Ling [17] extended the notion of slicing to the context where various database operations are present in the programs. They followed a similar solution and introduced a set of implicit variables to capture the influence among I/O statements operating on database records. Willmor et.al. [19] introduced a variant of the program dependence graph, known as Database-Oriented Program Dependence Graph (DOPDG), by considering two additional types of data dependences: Program-Database and Database-Database Dependences. The proposed DOPDG-based slicing uses Condition-Action rules introduced by Baralis and Widom [1] that determine when an action of one rule can affect the condition of another rule. In the presence of embedded DML statements, Cleve [3] proposed to construct System Dependence Graph (SDG) to represent the control and the dataflow of both the host language and the embedded language. Once the full SDG has been built, program slices can be computed using the usual algorithm. Recently, Saha et al. [13] proposed a new key-based dynamic slicing algorithm and two differencing techniques that use the underlying program semantics to localize faults in the data-centric programs that use embedded database specific statements to perform operations on in-memory and persistent data.

8. CONCLUSIONS

In this paper, we extended the notion of abstract program slicing to the case of database applications embedding data manipulation operations of SQL. We applied semantics-based dependence computations to DOPDGs, resulting into more precise semantics-based (abstract) DOPDGs. We are now investigating its possible extensions to the case of concurrent transactions accessing or manipulating same databases by computing inter-transaction dependences.

Acknowledgement

Work partially supported by PRIN 2010-11 project "Security Horizons".

9. REFERENCES

[1] E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. In *Proc. of the 20th Int. Conf. on Very Large Data Bases*, pages 475–486, 1994.

[2] J. Cheney. Program slicing and data provenance. *IEEE Data Engineering Bulletin*, 30:22–28, 2007.

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

[4] A. Cortesi and R. Halder. Abstract interpretation of recursive queries. In *Proc. of the 9th Int. Conf. on Distributed Computing and Internet Technologies*. Springer LNCS, 2013.

[5] R. Halder and A. Cortesi. Abstract program slicing on dependence condition graph. *Science of Computer Programming, Elsevier Ed. (http://dx.doi.org/10.1016/j.scico.2012.05.007, in press)*.

[6] R. Halder and A. Cortesi. Obfuscation-based analysis of sql injection attacks. In *Proc. of the 15th IEEE Symposium on Computers and Communications*, pages 931–938. IEEE Press, 2010.

[7] R. Halder and A. Cortesi. Abstract interpretation of database query languages. *Computer Languages, Systems & Structures*, 38:123–157, 2012.

[8] R. Halder, S. Pal, and A. Cortesi. Watermarking techniques for relational databases: Survey, classification and comparison. *Journal of Universal Computer Science*, 16(21):3164–3190, 2010.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Programming Languages and Systems*, 12(1):26–60, 1990.

[10] I. Mastroeni and D. Nikolic. Abstract program slicing: From theory towards an implementation. In *Proc. of the 12th Int. Conf. on Formal Engineering Methods*, pages 452–467. Springer LNCS 6447, 2010.

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

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

[13] D. Saha, M. G. Nanda, P. Dhoolia, V. K. Nandivada, V. Sinha, and S. Chandra. Fault localization for data-centric programs. In *Proc. of the 19th ACM SIGSOFT symposium and the 13th European Conf. on Foundations of software engineering*, pages 157–167. ACM Press, 2011.

[14] H. Seok Hong, I. Lee, and O. Sokolsky. Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In *Proc. of the 5th Int. Workshop on Source Code Analysis and Manipulation*, pages 25–34. IEEE CS, 2005.

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

[16] S. Sukumarana, A. Sreenivasb, and R. Metta. The dependence condition graph: Precise conditions for dependence between program points. *Computer Languages, Systems & Structures*, 36:96–121, 2010.

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

[18] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[19] D. Willmor, S. M. Embury, and J. Shao. Program slicing in the presence of a database state. In *Proc. of the 20th Int. Conf. on Software Maintenance*, pages 448–452. IEEE CS, 2004.

[20] D. Zanardini. The semantics of abstract program slicing. In *Proc. of the Int. Working Conf. on Source Code Analysis and Manipulation*, pages 89–100. IEEE Press, 2008.