The Journal of Systems & Software 175 (2021) 110903

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A deductive reasoning approach for database applications using verification conditions☆

Md. Imran Alam [a,*], Raju Halder [a], Jorge Sousa Pinto [b]

[a] Indian Institute of Technology Patna, Patna, India
[b] HASLab/INESC TEC, Universidade do Minho, Braga, Portugal

ARTICLE INFO

Article history:
Received 28 May 2020
Received in revised form 19 September 2020
Accepted 24 December 2020
Available online 5 January 2021

Keywords:
Database languages
Formal verification
Deductive reasoning
Verification conditions

ABSTRACT

Deductive verification has gained paramount attention from both academia and industry. Although intensive research in this direction covers almost all mainstream languages, the research community has paid little attention to the verification of database applications. This paper proposes a comprehensive set of Verification Conditions (VCs) generation techniques from database programs, adapting Symbolic Execution, Conditional Normal Form, and Weakest Precondition. The validity checking of the generated VCs for a database program determines its correctness w.r.t. the annotated database properties. The developed prototype `DBverify` based on our theoretical foundation allows us to instantiate VC generation from PL/SQL codes, yielding to detailed performance analysis of the three approaches under different circumstances. With respect to the literature, the proposed approach shows its competence to support crucial SQL features (aggregate functions, nested queries, NULL values, and set operations) and the embedding of SQL codes within a host imperative language. For the chosen set of benchmark PL/SQL codes annotated with relevant properties of interest, our experiment shows that only 38% of procedures are correct, while 62% violate either all or part of the annotated properties. The primary cause for the latter case is mostly due to the acceptance of runtime inputs in SQL statements without proper checking.

1. Introduction

Static program verification is an integral part of the software engineering process to formally prove or disprove the correctness of programs without executing them. Acknowledging its immense importance in critical systems, intensive research in this direction has been taking place since its inception 45 years ago (Bjørner and Havelund, 2014) and a rich class of verification methods, such as Theorem Proving (Ahrendt et al., 2016; Filliâtre, 2011; Hähnle and Huisman, 2019), Model Checking (Clarke et al., 1994; Clarke Jr et al., 2018; Jhala and Majumdar, 2009), Process Algebra (Fokkink, 2013), have been successfully introduced to formally verify the correctness of both finite-state (e.g., hardware designs) and infinite-state (e.g., software programs) systems, guaranteeing that an implementation or design satisfies its specification.

Deductive reasoning, based on general-purpose theorem proving, has emerged as a promising technique for software verification. Intensive research in this direction has been taking place with a coverage of almost all mainstream languages (Ahrendt et al., 2016; Hähnle and Huisman, 2019; Chalin et al., 2005; Cuoq et al., 2012; Weiß, 2011). Presently, the field of deductive verification has reached a stage of maturity for its use in an industrial setting (Hähnle and Huisman, 2017). In general, the deductive approach internally employs a Verification Condition Generator (VCG) which takes, as input, a program with specification and returns as output a number of proof obligations, called Verification Conditions (VCs). The VCs are then sent to a backend proof tool for validity checking (da Cruz et al., 2012). The use of VCs provides a relatively complete proof system with the use of a richer first-order specification language.

Database applications play a pivotal role in every aspect of our daily lives. Their existence is realized everywhere, ranging from simple web applications to even critical systems like banking, e-commerce, e-government, health-care, etc. In many situations, organizations prefer to adopt third party software modules and integrate them into their existing database-driven systems, keeping the underlying databases intact. Therefore, verification of such untrusted modules is essential, as erroneous codes may lead to inconsistency in the existing database data by violating their properties of interest. Fig. 1 exemplifies such a scenario considering a budget allocation system: The attributes TA of the database table `BudgetTab` stores the total proposed budget for

☆ Editor: Earl Barr.

* Corresponding author.
E-mail addresses: imran.pcs16@iitp.ac.in (M.I. Alam), halder@iitp.ac.in (R. Halder), jsp@di.uminho.pt (J.S. Pinto).

https://doi.org/10.1016/j.jss.2020.110903
0164-1212/

```
CREATE TABLE BudgetTab
(
    Did INT PRIMARY KEY,
    Dname VARCHAR(50),
    TA  NUMBER NOT NULL,
    MP NUMBER NOT NULL,
    EQ  NUMBER NOT NULL,
    CT  NUMBER NOT NULL,
    CS  NUMBER NOT NULL,
    // Set of properties
    CHECK (MP >= 80000),
    CHECK (EQ >= 60000),
    CHECK (CT >= 10000),
    CHECK (CS >= 5000)
);
```

(a) Table definition

```
1. CREATE OR REPLACE PROCEDURE DBprog (y int, x int) IS
2. z int;
3. m int;
4. n int;
5. BEGIN
6.    SELECT TA INTO z FROM BudgetTab
      WHERE Did = y;
7.    if (z ⩾ x) then
8.        m := z − x;
9.        n := m/4;
10.       UPDATE BudgetTab SET MP = MP − n,
          EQ = EQ − n, CT = CT − n,
          CS = CS − n WHERE Did = y;
11.   endif;
12. end;
```

(b) Stored procedure DBprog

Fig. 1. Motivating example.

each department, whereas other attributes MP, EQ, CT, and CS maintain its distribution under four heads *Manpower*, *Equipment*, *Contingency*, and *Consumable* respectively. The procedure DBprog extracts TA into the application variable z for a given department y (at program point 6) and compares it with the department's available budget x (at program point 7). The statement at 10 adjusts the budget by subtracting an equal fraction of the deficit amount (z - x) among four heads. The specification containing the number of constraints or properties in the form of CHECK constraints is specified as part of the BudgetTab table definition, which must be respected by the procedure DBprog on all executions. However, observe that as DBprog accepts run time inputs, the update operation may lead to a violation of the specification by reducing budget amount under some heads below their minimum threshold specified in the CHECK constraints.

1.1. Motivation and contributions

The presence of external database states, along with programs internal states, makes the verification task of database applications more challenging and painstaking. We observed that, although the database is an integral and indispensable part of most computing environments today, the research community has paid little attention in this direction.

On searching exhaustively in the literature, we find relatively few numbers of attempts to verify database applications (Baltopoulos et al., 2011; Benzaken and Schaefer, 1998; Christiansen and Martinenghi, 2003; Itzhaky et al., 2017; Malecha et al., 2010). Let us briefly present them. The proposed work in Itzhaky et al. (2017) manually translates the embedded SQL code into SmpSL script language and then computes verification conditions using the weakest precondition. Unfortunately, the approach suffers from a severe limitation in terms of scalability to adopt it for real-world programs. Precisely, the verification process takes care of only the decidable fragment of the problem within the scope of the two-variable first-order logic formula. As a result, the approach fails to accept SQL codes in the presence of arithmetic operations, aggregate functions, JOIN operations, etc. Predicate abstraction-based integrity constraints verification of extended version of O_2 object-oriented database language is proposed in Benzaken and Schaefer (1998). In order to cope with the verification complexity due to object referencing, the source code is translated into an intermediate form on which the predicate abstraction is applied. We observed that the intermediate language is not expressive enough to accommodate important database language features, such as nested query, arithmetic expressions, aggregate functions, etc. Authors in Christiansen and Martinenghi (2003) propose integrity constraints verification for database applications using transformation operators. The proposed approach expressed every update operation as a predicate $\mathbf{U} = P(\vec{a})$ and integrity constraints defined in a constraint theory τ. The function $\mathbf{after}^U(\tau)$ translates the constraint theory to the weakest precondition of ϕ with respect to the update **U**, and a simplified formula is obtained by applying function $\mathbf{Simp}^U(\phi)$. Verification of RDBMS specification and implementation is proposed in Malecha et al. (2010) using the Coq proof assistant, which has expressive-power limited to relational algebra only. The proposal in Baltopoulos et al. (2011) allows transactions to write in a functional language F# with database table-definitions as refinement types, and verify them using the refinement-type checker Stateful F7. In contrast to Malecha et al. (2010), where the main concern is to address database implementation issues, the approach in Baltopoulos et al. (2011) mainly concerns bugs in the user-defined transactions.

A comparative summary w.r.t. the literature is depicted in Table 1, where the existing proposals are compared w.r.t. ours based on their potential to deal with the type of properties, NULL values, aggregate functions, arithmetic expressions, and language paradigms. The notations T1–T5 represent the type of properties (Ullman, 2020) as follows: T1: Attribute-based (properties involving single attribute), T2: Tuple-based (properties involving multiple attributes), T3: Properties involving NULL values, T4: Properties involving aggregate values, and T5: General Assertions (properties involving both attributes and application variables).

It should be noticed that the applicability of the existing solutions in the literature is relatively poor due to their inability to embrace most crucial SQL features such as aggregate functions, nested queries, NULL values, arithmetic expressions. Moreover, most of them focus only on the verification of SQL statements without the support of the imperative language paradigm. In order to compensate the extra complexity, the approaches follow a common step of translating SQL into an intermediate form, and therefore the expressive power of the intermediate language often imposes a limitation on the verification of SQL features. Usually, database applications are written in popular host programming languages such as C, C++, Java, etc., with embedded data access logic expressed declaratively in Structured Query Language (SQL) or SQL-derived programming languages such as PL/SQL and T/SQL (Elmasri and Navathe, 2011). Therefore, verification of such applications, in contrast to the programs in mainstream languages, demands a different treatment due to the presence of database attributes along with program variables.

In order to facilitate the correctness proof of database applications addressing the above-mentioned challenges, in this paper, we propose a comprehensive set of techniques for the generation

Table 1
Comparative summary w.r.t. the literature.

Proposals	Properties	NULL	Aggregate functions	Arithmetic expressions	Language
Itzhaky et al. (2017)	T1, T2	No	No	No	SQL + Imperative
Baltopoulos et al. (2011)	T1-T3	Yes	No	Yes	SQL
Malecha et al. (2010)	T2	No	Yes	No	SQL
Benzaken and Schaefer (1998)	T2	No	No	No	O_2
Christiansen and Martinenghi (2003)	T2	No	No	No	SQL
Our proposal	T1–T5	Yes	Yes	Yes	SQL + Imperative

of Verification Conditions (VCs) from database programs, adapting Symbolic Execution (SE), Conditional Normal Form (CNF) and Weakest Precondition (WPC). If the generated VCs for a given database program can be discharged (i.e., proved valid by an automated theorem prover), then the program is guaranteed to be correct w.r.t. the specified database properties. The developed prototype `DBverify` based on our theoretical foundation allows us to instantiate VC generation from a set of PL/SQL benchmark codes (PL/SQL Project, 2020a,b,c,d,e,f,g,h), yielding to detailed performance analysis of these three approaches under different circumstances. As reported in Table 1, with respect to the literature, our approach is powerful enough to verify SQL codes embedded within a host imperative language, with a coverage of the above-mentioned crucial SQL features and common database properties (Ullman, 2020). It is worth mentioning that our work is primarily motivated by da Cruz et al. (2012), Frade and Pinto (2011) and Lourenço et al. (2015).

To summarize, the main contributions in this paper are:

- We propose a comprehensive set of techniques to generate Verification Conditions (VCs) from database applications where database statements are embedded into a host imperative language. The generated VCs are then processed by an automated theorem prover for their validity checking in order to prove the program's correctness. To this aim, we adapt Symbolic Execution (SE), Conditional Normal Form (CNF), and Weakest Precondition (WPC). The proposed techniques allow to support important SQL features, including aggregate functions, nested queries, NULL values, and various operations (JOIN, UNION, INTERSECT, and MINUS).
- We formalize the conversion of database programs into a single assignment form, which facilitates the verification process, especially in the case of Symbolic Execution and Conditional Normal Form.
- We develop DBverify a verification tool implemented in Python based on our theoretical foundation, which enables users to verify PL/SQL procedures under three different approaches. DBverify makes use of ANTLR parser Parr (2013) to generate VCs and Microsoft's Z3 theorem prover De Moura and Bjørner (2008) to check the validity.
- Finally, we perform an experimental evaluation on a set of benchmark PL/SQL codes (PL/SQL Project, 2020a,b,c,d,e,f,g,h) under all three different approaches. We present a detailed performance analysis based on the experimental results and establish the effectiveness of the approaches under various circumstances.

The rest of the paper is organized as follows: Section 2 recalls the abstract syntax of the languages under consideration and introduces the conversion of database programs into single assignment form. Section 3 presents in detail all the proposed verification condition generation techniques for database programs. The complexity analysis of the proposed approaches and their correctness proofs are detailed in Section 4. Section 5 presents DBverify, a prototype implementation for the verification of PL/SQL codes. The experimental result on a set of PL/SQL benchmark codes with detailed performance analysis is described in Section 6. Section 7 presents threats to validity. Finally, Section 8 covers the current state-of-the-art in this research line, and Section 9 concludes our work.

2. Database language and single assignment form

In this section, we first recall from Halder and Cortesi (2012) and Jana et al. (2018b) the abstract syntax of the database language under consideration. Then we introduce its single assignment form, an intermediate language representation, which serves as a backbone of some approaches proposed in Section 3.

2.1. Abstract syntax of database language

We consider a generic scenario of database applications where SQL codes are embedded into another high-level host language. To this aim, let us recall from Halder and Cortesi (2012) and Jana et al. (2018b) the abstract syntax of the language, which, for the sake of simplicity in the theoretical formalism, supports SQL data manipulation languages hosted by imperative statements. This is depicted in Table 2. The variables are categorized into two: database attributes set $\mathbb{V}_d$ and application variables set $\mathbb{V}_a$. The arithmetic expressions e and boolean expressions b are defined accordingly, considering the presence of either $v \in \mathbb{V}_a$ or $a \in \mathbb{V}_d$ or both along with possible arithmetic and/or relational operators. Observe that g, r, f, and s represent group-by, distinct/all, order-by, and aggregate functions respectively, where *id* is the identity function.

The SQL statements Q_{sel}, Q_{upd}, Q_{ins}, and Q_{del} consist of an action-part and a condition-part. For example, in the update statement $\langle \vec{a} := \texttt{UPDATE}(\vec{e}),\ cond\rangle$, the first component $\vec{a} := \texttt{UPDATE}(\vec{e})$ represents an action-part and the second component *cond* represents a condition-part which follows well-formed formulas in the first order logic. To exemplify this, let us consider the statement "`UPDATE emp SET sal` $:=$ `sal` $+$ *bonus* `WHERE age` $\geqslant 60$", where "age $\geqslant 60$" corresponds to the condition part and $\langle \texttt{sal}\rangle := \texttt{UPDATE}(\langle \texttt{sal} + bonus\rangle)$ corresponds to the action part. Therefore, its abstract syntax is defined as $\langle\langle \texttt{sal}\rangle := \texttt{UPDATE}(\langle \texttt{sal} + bonus\rangle), age \geqslant 60\rangle$. Informally, the semantics of the update statement can be described as follows: For the database tuples which satisfy the condition-part *cond*, the values of the attributes $a_i \in \vec{a}$ are updated (in sequence) by $e_i \in \vec{e}$. Similarly, given a query "`SELECT DISTINCT Dno, Pno, MAX(Sal) INTO rs FROM Tab WHERE Sal` $>$ `1000 GROUP BY Dno, Pno HAVING MAX(Sal)` $<$ `4000 ORDER BY Dno`". Its abstract syntax, according to Table 2, is $\langle rs := \texttt{SELECT}(f(\vec{e}),\ r(\vec{h}(\vec{a})),\ \phi,\ g(\vec{e}),\ cond)\rangle$, where $cond \triangleq \texttt{Sal} > 1000$, $g(\vec{e}) \triangleq \texttt{GROUP BY}(\langle \texttt{Dno, Pno}\rangle)$, $r(\vec{h}(\vec{a})) \triangleq \texttt{DISTINCT}(\langle \texttt{DISTINCT(Dno), DISTINCT(Pno), MAX}\circ\texttt{ALL(Sal)}\rangle)$, $\phi \triangleq \texttt{MAX(Sal)} < 4000$, and $f(\vec{e}) \triangleq \texttt{ORDER BY ASC}(\langle \texttt{Dno}\rangle)$. In general, ϕ filters a set of tuples from the target table based on the satisfaction of *cond* and the result obtained after processing these tuples using g, ϕ, h, r, f (if present) is stored in the resultset application variable *rs*. The syntax of *cond* in the form of $a \otimes (Q_{sel})$ supports nested queries as well. Observe that, since insertion of a new tuple does not require any condition to satisfy, the *cond* in Q_{ins} is by default *false*.

Table 2
Abstract syntax of database language Halder and Cortesi (2012), Jana et al. (2018b).

Constant:	c	$\in$	$\mathbb{R}$ (Set of Numerical Constants)
Variables:	v	$\in$	$\mathbb{V}_a$ (Set of Application Variables)
	a	$\in$	$\mathbb{V}_d$ (Set of Database Attributes)
	Var	$::=$	$\mathbb{V}_a \cup \mathbb{V}_d$
	$\vec{a}$	$::=$	$\langle a_1, a_2, \ldots, a_n \rangle$ (Ordered sequence of attributes)
Expressions:	e	$\in$	$\mathbb{E}$ (Set of Arithmetic Expressions)
	e	$::=$	$c \mid v \mid a \mid e \oplus e$, where $\oplus \in \{+, -, *, /\}$
	b	$\in$	$\mathbb{B}$ (Set of Boolean Expressions)
	b	$::=$	$true \mid false \mid e \odot e \mid b \vee b \mid b \wedge b \mid \neg b$ where $\odot \in \{<, \leq, >, \geq, =, \neq\}$
SQL Conditions: (Well-formed first order formula)	τ	$\in$	$\mathbb{T}$ (Set of Terms)
	τ	$::=$	$c \mid a \mid v \mid f_n(\tau_1, \tau_2, ..., \tau_n)$ where f_n is an n-ary function.
	a_f	$\in$	$\mathbb{A}_f$ (Set of Atomic Formulas)
	a_f	$::=$	$\tau_1 == \tau_2 \mid R_n(\tau_1, \tau_2, ..., \tau_n)$ where $R_n(\tau_1, ..., \tau_n) \in \{true, false\}$
	ϕ	$\in$	$\mathbb{W}$ (Set of Well-Formed Formula)
	ϕ	$::=$	$a_f \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2$
	$cond$	$::=$	$true \mid false \mid \phi \mid a \otimes (\mathtt{Q})$, where $\otimes \in$ {IN, NOT IN, ANY, EXIST, NOT EXIST}
SQL Functions:	$g(\vec{e})$	$::=$	GROUP BY($\vec{e}$), where $\vec{e} = \langle e_1, ..., e_n \mid e_i \in \mathbb{E} \rangle$
	r	$::=$	DISTINCT | ALL
	s	$::=$	AVG | SUM | MAX | MIN | COUNT | id
	$h(e)$	$::=$	$s \circ r(e)$
	$\vec{h}(\vec{x})$	$::=$	$\langle h_1(x_1), ..., h_n(x_n) \rangle$, $\vec{h} = \langle h_1, ..., h_n \rangle$ and $\vec{x} = \langle x_1, ..., x_n \mid x_i = e \vee x_i = * \rangle$
	$f(\vec{e})$	$::=$	ORDER BY ASC($\vec{e}$) | ORDER BY DESC($\vec{e}$) | id
Commands:	$\mathtt{Q}$	$\in$	$\mathbb{Q}$ (Set of SQL Statements)
	$\mathtt{Q}$	$::=$	$\mathtt{Q}_{sel} \mid \mathtt{Q}_{upd} \mid \mathtt{Q}_{ins} \mid \mathtt{Q}_{del}$
	$\mathtt{Q}_{sel}$	$::=$	$\langle rs := \mathtt{SELECT}(f(\vec{e}),\ r(\vec{h}(\vec{x})),\ \phi,\ g(\vec{e})),\ cond \rangle$
	$\mathtt{Q}_{upd}$	$::=$	$\langle \vec{a} := \mathtt{UPDATE}(\vec{e}),\ cond \rangle$
	$\mathtt{Q}_{ins}$	$::=$	$\langle \vec{a} := \mathtt{INSERT}(\vec{e}),\ false \rangle$
	$\mathtt{Q}_{del}$	$::=$	$\langle \vec{a} := \mathtt{DELETE}(),\ cond \rangle$
	$stmt$	$\in$	$\mathbb{C}$ (Set of Commands)
	$stmt$	$::=$	$\mathtt{Q} \mid v := e \mid$ skip $\mid$ if b then $stmt$ endif
		|	read v | if b then $stmt_1$ else $stmt_2$ endif
		|	$stmt_1$; $stmt_2$
Programs:	$\mathcal{P}$	$\in$	$\mathbb{P}$ (Set of database programs)
	$\mathcal{P}$	$::=$	$stmt \mid stmt\ ;\ \mathcal{P}$

2.2. Assertion language and database properties

As deductive verification is not fully automatic, this requires annotating the source code with *assume* and *assert* statements according to the specifications. The informal semantics of *assume* and *assert* are as follows: the *assume* ψ command excludes all computations which do not satisfy logical expression ψ, whereas *assert* ψ commands at different program points specify the properties to be proven. A program is correct if, for every execution, whenever *assert* ψ is reached, the assertion ψ is satisfied by the current state. Since our verification starts with the assumption that the initial database is in consistent states w.r.t database properties, in order to reflect this, we annotate our program by placing *assume* command at the beginning of the code. As the database commands are responsible for changing database states, to detect any property violation at a higher granularity level, we allow the annotation by placing *assert* commands anywhere within the program, especially after database commands.

We adopt the assertion language from Winskel (1993), defined in Eq. (1), in order to support the following types of database properties (Ullman, 2020): Attribute-based, Tuple-based, Properties involving NULL and aggregate values, and General Assertions.

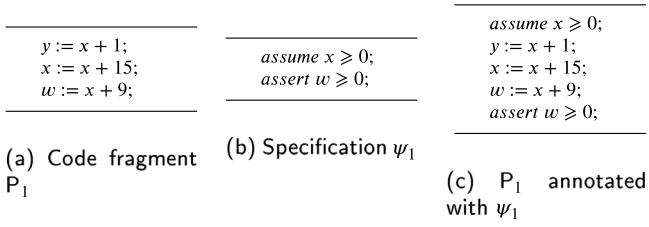
$$\psi ::= true \mid false \mid a_0 = a_1 \mid a_0 \leqslant a_1 \mid \psi_0 \vee \psi_1 \mid \psi_0 \wedge \psi_1 \mid \neg\psi \mid \psi_0 \Rightarrow \psi_1 \mid \forall i.\psi \mid \exists i.\psi \tag{1}$$

where i ranges over integer variables and a is the arithmetic expression defined below:

$$a ::= n \mid i \mid a_0 + a_1 \mid a_0 - a_1 | a_0 \times a_1$$

where $n \in \mathbb{R}$. Readers may refer to Winskel (1993) for the semantics of the assertion language.

To exemplify various property types, let us consider a database schema: Employee(ID, Name, DOJ, Experience, Salary), Department (DID, Dname, MgrID, MgrStartDate). The following assertions 'Salary $>$ 100 $\wedge$ Salary $\leqslant$ 4000' and 'Experience $>$ 5 $\wedge$ Salary $>$ 2000' represent attribute-based and tuple-based properties respectively depending upon the presence of one or more attributes in the assertions, whereas the assertion 'MgrStartDate $>$ DOJ $\wedge$ Experience $>$ y' represents a general assertion due to the involvement of both database attributes and application variables. The examples of assertions involving NULL and aggregate values are 'DID NUMBER NOT NULL', 'Avg(Salary) $>$ 2500' respectively.



(a) Code fragment P_1 (b) Specification ψ_1 (c) P_1 annotated with ψ_1

Fig. 2. A simple code fragment and its annotation.

2.3. Single assignment form of database language

Single Assignment (Aycock and Horspool, 2000) is a semantically equivalent representation of a program in which every new assignment to a variable results in a new version and each version denotes a different logical variable. This ensures that, in single assignment, each variable is defined only once before being used. A versioned variable is defined by the original name of the variable associated with an integer subscript representing its current version.

Obtaining a logical encoding of a program is an essential task in any deduction-based verification framework, allowing for the generation of appropriate verification conditions. Let us demonstrate on a simple code snippet how a single assignment form of program code leads to the generation of the correct form of VCs capturing the actual program semantics. Given the code fragment P_1 and the specification ψ_1 depicted in Figs. 2(a) and 2(b). The logical expression $f \triangleq x \geqslant 0 \wedge y = x + 1 \wedge x = x + 15 \wedge w = x + 9 \implies w \geq 0$ represents a VC of the annotated code in Fig. 2(c).

However, observe that, although P_1 is correct w.r.t. ψ_1, this can never be captured due to the unsatisfiability of f because $x = x + 15$ is always unsatisfiable. To make f satisfiable, one has to differentiate between the variable x appearing at the left and right sides of the assignment operator (:=). The best way to do this is to convert program codes into a single assignment form da Cruz et al. (2012).

Let us now make a quick journey through the standard notions of single assignment form of imperative statements, after which

```
read x;
if x > 0 then
    x := x − 1;
else
    x := x + 1;
endif
y = x + 5;
```

(a) Code fragment P_2

```
read x0;
if x0 > 0 then
    x1 := x0 − 1;
else
    x2 := x0 + 1;
endif
x3 := φ(x1, x2);
y1 = x3 + 5;
```

(b) SSA form of P_2

```
read x0;
if x0 > 0 then
    x1 := x0 − 1;
    x3 := x1;
else
    x2 := x0 + 1;
    x3 := x2;
endif
y1 := x3 + 5;
```

(c) DSA form of P_2

Fig. 3. `If-else` statement and its single assignment form.

```
if (x ⩾ 0) then
    y := y − 5;
endif
x := y;
```

(a) Code fragment P_3

```
if (x0 ⩾ 0) then
    y1 := y0 − 5;
endif
y2 := φ(y1, y0);
x1 := y2;
```

(b) SSA form of P_3

```
if (x0 ⩾ 0) then
    y1 := y0 − 5;
    y2 := y1;
else
    y2 := y0;
endif
x1 := y2;
```

(c) DSA form of P_3

Fig. 4. `if`-statement and its single assignment form.

we will extend these notions for our database language under consideration.

Single Assignment Form of Imperative Statements (Aycock and Horspool, 2000; Briggs et al., 1998). As the prime objective in single assignment form is to ensure single definitions for all variables, in the case of an assignment statement, a new version for the defined variable appeared on the left side of the assignment operator is introduced. For example, consider the code: {`read` x; $x := x + 1$;}, its equivalent single assignment form is {`read` x_0; $x_1 := x_0 + 1$;}, where x_0 and x_1 denote the initial version and the current version of x respectively. In case of conditional statements where two or more control flow paths merge at a point, the single assignment property may get violated since multiple definitions of a variable may reach that merging point. To solve this problem, imaginary assignments are introduced at the merging points by using ϕ-functions and the resultant representation is called Static Single Assignment (SSA) form Briggs et al. (1998). In general, a ϕ-function has arguments corresponding to each incoming control flow path: the ith argument of a ϕ-function is the incoming value along the ith path. For example, the SSA form of the code fragment P_2 in Fig. 3(a) is depicted in Fig. 3(b). Note that, in order to make such SSA form executable, the ϕ-function can equivalently be defined in terms of the ternary conditional operator. For example, the statement $x_3 := \phi(x_1, x_2)$; in Fig. 3(b) can be defined as $x_3 := x_0 > 0\ ?\ x_1 : x_2$;.

Dynamic Single Assignment (DSA) form is an alternative representation where variables are defined along all control paths to ensure that a single definition reaches towards the merging points. Since the meaning of a ϕ-function is a mapping of all incoming values to a single name (say x_3), it is equivalent to place a copy of x_3 at the end of each predecessor-block. The copy moves the values corresponding to the appropriate ϕ-function argument into x_3. This can be seen as a destruction of a ϕ-function into its direct predecessor-blocks in SSA form Briggs et al. (1998). For example, the code snippet in Fig. 3(c) depicts the equivalent DSA form of P_2. Similarly, Fig. 4 highlights the SSA and DSA treatments of `if`-statement.

```
UPDATE T SET b = b − 10 WHERE a > 50 AND a < 100 ;
```

(a) `UPDATE` statement Q

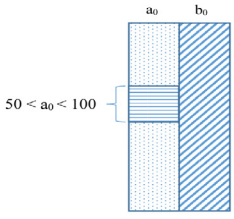
```
UPDATE T̂ SET b1 = b0−10 WHERE a0 > 50 AND a0<100;
```

(b) Wrong single assignment form of Q

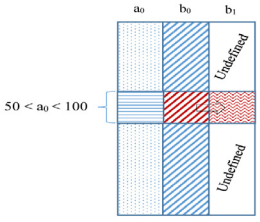
```
UPDATE T̂ SET b1 = b0 WHERE ¬(a0 > 50 AND a0<100);
UPDATE T̂ SET b1 = b0 − 10 WHERE a0 > 50 AND a0<100;
```

(c) Correct single assignment form of Q

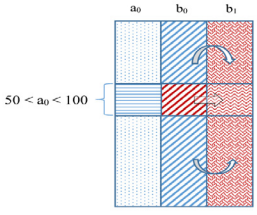
Fig. 5. Single assignment form of `UPDATE` statement.



(a) Initial state of table T



(b) State of T w.r.t. single assignment form in Figure 5b



(c) State of T w.r.t. single assignment form in Figure 5c

Fig. 6. State representation of the table T w.r.t. Q, and their single assignment form in Fig. 5.

Single assignment form of SQL statements. Let us now extend the notions of SSA and DSA to the case of database applications. To exemplify this, consider the `UPDATE` statement Q in Fig. 5(a). Treating Q in a similar way as in the case of imperative statements, we get its single assignment form depicted in Fig. 5(b). Note that, in the conversion process, we assume the existence of a ghost database (denoted by cap$\widehat{()}$ on table names), which contains all versions of the attributes of the original database. This allows us to observe that the resultant single assignment form would be wrong because it reflects only a part of the database state for which $a_0 > 50 \wedge a_0 < 100$ holds. In order to capture the complete database state, we add an auxiliary `UPDATE` statement by considering the negation of *cond* which covers the other part of the database state. The correct single assignment form of Q is depicted in Fig. 5(c). We show this fact pictorially in Fig. 6.

The following formula computed from the correct single assignment form of Q represents a valid VC:

$(b_1 = b_0 - 10 \ \wedge \ a_0 > 5 \ \wedge \ a_0 < 100) \vee \quad (b_1 = b_0 \ \wedge \ \neg(a_0 > 5 \ \wedge \ a_0 < 100))$

Similarly, we can define the single assignment form of INSERT and DELETE by adding auxiliary UPDATE statements before them in order to reflect other parts of the database state also. In these two cases, we have to consider the versioning of all attributes in the table which are referred by the statements. Observe that, the SELECT statement changes the version of the application result-set variables only.

Treating SQL statements under if- and if-else statements. Similarly to statements of an imperative language, the presence of UPDATE or DELETE or INSERT statement in if- and if-else requires special treatment to convert them into SSA or DSA form. The code snippets given in Fig. 7 illustrate this.

Observe that the ϕ-function ϕ_d in the case of multi-definitions of database attributes under SSA form can be defined in terms of UPDATE controlled by if or if-else. For example, $b_2 := \phi_d(b_1, b_0)$; at program point 3 in Fig. 7(b) can be defined as:

```
if z0 ⩾ 3 then
    UPDATE T̂1 SET b2 = b1;
else
    UPDATE T̂1 SET b2 = b0;
endif
```

On the other hand, in the case of DSA of database code, the else-part is introduced whenever necessary (for example, statement 4 in Fig. 7(c)) and ϕ-functions are destructed into their predecessor-blocks (for instance, by introducing statements 3, 4, 21, and 23 in Fig. 7(c)), which ensures the single definition of attributes to flow towards the merging points.

Since our proposed verification approaches make use of only DSA form whenever applicable, we restrict our discussions to DSA translation only in the subsequent sections. Let us now formalize DSA translation of the database language. Let $\sigma \in \Sigma$ be a function which maps program variables to their corresponding versions, defined as: σ: Var $\rightarrow \mathbb{I}$ where $\mathbb{I}$ is the set of natural numbers. The initial version is defined by σ_0: Var $\rightarrow\{0\}$. We define the function TranDSA : $\mathbb{C} \times \Sigma \rightarrow \mathbb{C}^{dsa} \times \Sigma$ for our language which maps a given command $c \in \mathbb{C}$ w.r.t. current variables-version $\sigma \in \Sigma$ into its equivalent DSA form $c^{dsa} \in \mathbb{C}^{dsa}$ resulting into a new variables-version $\sigma' \in \Sigma$. Fig. 8 depicts the detailed definition of TranDSA for various components of our language under consideration. Observe that TranDSA renames variable v or attribute a into their corresponding versioned form v_i or a_i w.r.t. current version σ where $\sigma(v) = i$ or $\sigma(a) = i$. In the case of assignment statements, the current versions of defined variables are incremented by 1 resulting in an updated version σ'.

Let us now introduce a new syntactic form:

$\langle \vec{a} := \text{act}^1 \curvearrowright \text{act}^2, \ cond^1 \curvearrowright cond^2 \rangle$

which indicates that the action-part act^1 on $\vec{a}$ for the tuples satisfying $cond^1$ will be followed by another action act^2 on the same $\vec{a}$ for the tuples satisfying $cond^2$. For example, UPDATE, INSERT and DELETE statements under this syntactic form are represented as:

$\langle \vec{a} :=\texttt{UPDATE}(\vec{a}) \curvearrowright \texttt{UPDATE}(\vec{e}), \ \neg cond \curvearrowright cond \rangle$
$\langle \vec{a} :=\texttt{UPDATE}(\vec{a}) \curvearrowright \texttt{INSERT}(\vec{e}), \ true \curvearrowright false \rangle$
$\langle \vec{a} :=\texttt{UPDATE}(\vec{a}) \curvearrowright \texttt{DELETE}(), \ \neg cond \curvearrowright cond \rangle$

Since the DSA forms of UPDATE, DELETE, and INSERT add an auxiliary UPDATE statement before them in order to capture the definition of attributes for all tuples, we use this syntactic form in the definition of TranDSA. Observe that, like assignment statements, in these cases also σ is modified to σ' by increasing the versions of defined attributes by 1. In the case of conditional statements if- and if-else, we use two functions Destruct and Sync,

```
1.  if z ⩾ 3 then
2.      UPDATE T1 SET b = b − 10 WHERE a = y;
    endif;
3.  SELECT b INTO w FROM T1 WHERE a > 5;
                ⋮
19. if y > 60 then
20.     DELETE FROM T2 WHERE d = y;
    else
21.     INSERT INTO T2 (c, d) VALUES (p, q);
    endif;
22. SELECT d INTO x FROM T2 WHERE c > 5;
```

(a) Original code fragment P_4

```
1.  if z0 ⩾ 3 then
        UPDATE T̂1 SET b1 = b0 WHERE ¬(a0 = y0);
2.      UPDATE T̂1 SET b1 = b0 − 10 WHERE a0=y0;
    endif;
3.  b2 := φd(b1, b0);
4.  SELECT b2 INTO w1 FROM T̂1 WHERE a0 > 5;
                ⋮
19. if y0 > 60 then
        UPDATE T̂2 SET c1=c0, d1=d0 WHERE
        ¬(d0=y0);
20.     DELETE FROM T̂2 (c0, d0) WHERE d0 = y0;
    else
        UPDATE T̂2 SET c2 = c0, d2 = d0;
21.     INSERT INTO T̂2 (c2, d2) VALUES (p0, q0);
    endif;
22. c3 := φd(c1, c2);
23. d3 := φd(d1, d2);
24. SELECT d3 INTO x1 FROM T̂2 WHERE c3 > 5;
```

(b) SSA form of P_4

```
1.  if z0 ⩾ 3 then
        UPDATE T̂1 SET b1 = b0 WHERE ¬(a0 = y0);
2.      UPDATE T̂1 SET b1 = b0 − 10 WHERE a0=y0;
3.      UPDATE T̂1 SET b2 = b1;
    else
4.      UPDATE T̂1 SET b2 = b0;
    endif;
5.  SELECT b2 INTO w1 FROM T̂1 WHERE a0 > 5;
                ⋮
19. if y0 > 60 then
        UPDATE T̂2 SET c1=c0, d1=d0 WHERE
        ¬(d0=y0);
20.     DELETE FROM T̂2 (c0, d0) WHERE d0 = y0;
21.     UPDATE T̂2 SET c3 = c1, d3 = d1;
    else
      UPDATE T̂2 SET c2 = c0, d2 = d0;
22.     INSERT INTO T̂2 (c2, d2) VALUES (p0, q0);
23.     UPDATE T̂2 SET c3 = c2, d3 = d2;
    endif;
24. SELECT d3 INTO x1 FROM T̂2 WHERE c3 > 5;
```

(c) DSA form of P_4

Fig. 7. SQL statements and their single assignment form under if- and if-else.

$\mathbb{I}$: Set of natural numbers
$\mathbb{C}$: Set of Commands
$\mathbb{C}^{dsa}$: Set of Commands in DSA form

Versioning Function : $\sigma \in \Sigma \triangleq \mathsf{Var} \rightarrow \mathbb{I}$
Initial Version : $\sigma_0 \triangleq \mathsf{Var} \rightarrow \{0\},\ \text{where}\ \sigma_0 \in \Sigma$
DSA Translation Function : $\mathsf{TranDSA} \triangleq \mathbb{C} \times \Sigma \rightarrow \mathbb{C}^{dsa} \times \Sigma$

$$\mathsf{TranDSA}[\![c]\!]\sigma \triangleq \langle c, \sigma\rangle \qquad \mathsf{TranDSA}[\![v]\!]\sigma \triangleq \langle v_i,\ \sigma\rangle\ when\ \sigma(v) = i \qquad \mathsf{TranDSA}[\![a]\!]\sigma \triangleq \langle a_i,\ \sigma\rangle\ when\ \sigma(a) = i$$

$$\mathsf{TranDSA}[\![e]\!]\sigma \triangleq \langle \forall x \in \mathsf{Var}[\![e]\!] : e\big[\![\mathsf{TranDSA}[\![x]\!]/x]\!\big],\ \sigma\rangle \qquad \mathsf{TranDSA}[\![b]\!]\sigma \triangleq \langle \forall x \in \mathsf{Var}[\![b]\!] : b\big[\![\mathsf{TranDSA}[\![x]\!]/x]\!\big],\ \sigma\rangle$$

$$\mathsf{TranDSA}[\![\phi]\!]\sigma \triangleq \langle \forall x \in \mathsf{Var}[\![\phi]\!] : \phi\big[\![\mathsf{TranDSA}[\![x]\!]/x]\!\big],\ \sigma\rangle \qquad \mathsf{TranDSA}[\![a \otimes (Q_{sel})]\!]\sigma \triangleq \langle \mathsf{TranDSA}[\![a]\!]\sigma \otimes \mathsf{TranDSA}[\![(Q_{sel})]\!]\sigma,\ \sigma\rangle$$

$$\mathsf{TranDSA}[\![assume\ \phi]\!]\sigma \triangleq \langle assume\ \mathsf{TranDSA}[\![\phi]\!]\sigma,\ \sigma\rangle \qquad \mathsf{TranDSA}[\![assert\ \phi]\!]\sigma \triangleq \langle assert\ \mathsf{TranDSA}[\![\phi]\!]\sigma,\ \sigma\rangle$$

$$\mathsf{TranDSA}[\![v\ :=\ e]\!]\sigma \triangleq \langle \mathsf{TranDSA}[\![v]\!]\sigma' := \mathsf{TranDSA}[\![e]\!]\sigma,\ \sigma'\rangle,\ where\ \sigma' = \sigma[\![v \mapsto \sigma(v) + 1]\!]$$

$$\mathsf{TranDSA}[\![\langle \vec{a} := \mathtt{UPDATE}(\vec{e}),\ cond\rangle]\!]\sigma \triangleq \mathsf{TranDSA}[\![\langle \vec{a} := \mathtt{UPDATE}(\vec{a}) \curvearrowright \mathtt{UPDATE}(\vec{e}),\ \neg cond \curvearrowright cond\rangle]\!]$$
$$\triangleq \big\langle \langle \mathsf{TranDSA}[\![\vec{a}]\!]\sigma' := \mathtt{UPDATE}(\mathsf{TranDSA}[\![\vec{a}]\!]\sigma) \curvearrowright \mathtt{UPDATE}(\mathsf{TranDSA}[\![\vec{e}]\!]\sigma),$$
$$\mathsf{TranDSA}[\![\neg(cond)]\!]\sigma \curvearrowright \mathsf{TranDSA}[\![cond]\!]\sigma\rangle, \sigma'\big\rangle;\ where\ \sigma' = \sigma[\![\vec{a} \mapsto \sigma(\vec{a}) + 1]\!]$$

$$\mathsf{TranDSA}[\![\langle \vec{a} := \langle \mathtt{INSERT}(\vec{e}), false\rangle]\!]\sigma \triangleq \mathsf{TranDSA}[\![\vec{a} := \mathtt{UPDATE}(\vec{a}) \curvearrowright \mathtt{INSERT}(\vec{e}),\ true \curvearrowright false\rangle]\!]$$
$$\triangleq \big\langle \langle \mathsf{TranDSA}[\![\vec{a}]\!]\sigma' := \mathtt{UPDATE}(\mathsf{TranDSA}[\![\vec{a}]\!]\sigma) \curvearrowright \mathtt{INSERT}(\mathsf{TranDSA}[\![\vec{e}]\!]\sigma),$$
$$true \curvearrowright false\rangle, \sigma'\big\rangle;\ where\ \sigma' = \sigma[\![\vec{a} \mapsto \sigma(\vec{a}) + 1]\!]$$

$$\mathsf{TranDSA}[\![\langle \vec{a} := \mathtt{DELETE}(), cond\rangle]\!]\sigma \triangleq \mathsf{TranDSA}[\![\langle \vec{a} := \langle \mathtt{UPDATE}(\vec{a}_i) \curvearrowright \mathtt{DELETE}(),\ \neg cond \curvearrowright cond\rangle]\!]$$
$$\triangleq \big\langle \langle \mathsf{TranDSA}[\![\vec{a}]\!]\sigma' := \mathtt{UPDATE}(\mathsf{TranDSA}[\![\vec{a}]\!]\sigma) \curvearrowright \mathtt{DELETE}(), \mathsf{TranDSA}[\![\neg(cond)]\!]\sigma \curvearrowright$$
$$\mathsf{TranDSA}[\![cond]\!]\sigma\rangle, \sigma'\big\rangle;\ where\ \sigma' = \sigma[\![\vec{a} \mapsto \sigma(\vec{a}) + 1]\!]$$

$$\mathsf{TranDSA}[\![\langle rs\ :=\ \mathtt{SELECT}(f(\vec{e}), r(\vec{h}(\vec{x})),\ \phi, g(\vec{e})),\ cond\rangle]\!]\sigma \triangleq \big\langle \langle \mathsf{TranDSA}[\![rs]\!]\sigma' := \mathtt{SELECT}(f(\mathsf{TranDSA}[\![e]\!]\sigma), r(\vec{h}(\mathsf{TranDSA}[\![\vec{x}]\!]\sigma)), \mathsf{TranDSA}$$
$$[\![\phi]\!]\sigma, g(\mathsf{TranDSA}[\![e]\!]\sigma)),\ \mathsf{TranDSA}[\![cond]\!]\sigma\rangle,\ \sigma'\big\rangle,\ where\ \sigma' = \sigma[\![\vec{a} \mapsto \sigma(\vec{a}) + 1]\!]$$

$$\mathsf{TranDSA}[\![\mathtt{if}\ b\ \mathtt{then}\ stmt\ \mathtt{endif}]\!]\sigma \triangleq \mathtt{if}\ \mathsf{TranDSA}[\![b]\!]\sigma\ \mathtt{then}\ \mathsf{TranDSA}[\![stmt; \mathsf{Destruct}(stmt)]\!]\sigma\ \mathtt{else}$$
$$\mathsf{TranDSA}[\![\mathsf{Destruct}(stmt)]\!]\sigma\ \mathtt{endif}$$

$$\mathsf{TranDSA}[\![\mathtt{if}\ b\ \mathtt{then}\ stmt_1\ \mathtt{else}\ stmt_2\ \mathtt{endif}]\!]\sigma \triangleq \mathtt{if}\ \mathsf{TranDSA}[\![b]\!]\sigma\ \mathtt{then}\ \mathsf{TranDSA}[\![\mathsf{Sync}(stmt_1, stmt_2);\ \mathsf{Destruct}(stmt_1)]\!]\sigma\ \mathtt{else}$$
$$\mathsf{TranDSA}[\![\mathsf{Sync}(stmt_2, stmt_1);\ \mathsf{Destruct}(stmt_2)]\!]\sigma\ \mathtt{endif}$$

$$\mathsf{TranDSA}[\![stmt_1; stmt_2]\!]\sigma \triangleq \langle stmt_1^{dsa}\ ;\ stmt_2^{dsa},\ \sigma''\rangle\ where, \mathsf{TranDSA}[\![stmt_1]\!]\sigma = \langle stmt_1^{dsa},\ \sigma'\rangle$$
$$and\ \mathsf{TranDSA}[\![stmt_2]\!]\sigma' = \langle stmt_2^{dsa},\ \sigma''\rangle$$

Fig. 8. DSA translation function TranDSA.

defined below:

$\mathsf{Destruct}(v\ :=\ e) \triangleq v\ :=\ v$
$\mathsf{Destruct}(\langle \vec{a} := \mathtt{UPDATE}(\vec{e}),\ cond\rangle) \triangleq \langle \vec{a} := \mathtt{UPDATE}(\vec{a}),\ true\rangle$
$\mathsf{Destruct}(\langle \vec{a} := \mathtt{INSERT}(\vec{e}),\ false\rangle) \triangleq \langle \vec{a} := \mathtt{UPDATE}(\vec{a}), true\rangle$
$\mathsf{Destruct}(\langle \vec{a} := \mathtt{DELETE}(), cond\rangle) \triangleq \langle \vec{a} := \mathtt{UPDATE}(\vec{a}), true\rangle$
For Other Statements $stmt$: $\mathsf{Destruct}(stmt) \triangleq skip$
$\mathsf{Destruct}(stmt_1;\ stmt_2) \triangleq \mathsf{Destruct}(stmt_1);\ \mathsf{Destruct}(stmt_2)$
$\mathsf{Sync}(stmt_i, stmt_j) \triangleq stmt_i; \mathsf{Destruct}(stmt_j - stmt_i)$

where ($stmt_j$ - $stmt_i$) represents only those defining statements in $stmt_j$ which are not common to both $stmt_i$ and $stmt_j$. Note that the task of the Sync function is to make variable definitions in both if- and else-blocks consistent by adding a new part ($stmt_j$ - $stmt_i$) only after applying Destruct on it ensuring that no change in the values of the variables takes place before and after their inclusion. For example, given the following if-else code snippet "if ($x \geqslant 0$) then (1) UPDATE T SET $a := a + 10$ WHERE $c > 10$ AND $c < 20$; else (2) UPDATE T SET $b := b - 15$ WHERE $c > 25$ AND $c < 35$; endif". According to the definition of TranDSA in the case of if-else, the use of Sync and Destruct functions yields the following DSA form:

if $x_0 > 0$ then
(1) UPDATE $\widehat{\mathsf{T}}$ SET $a_1 = a_0 + 10$ WHERE $c > 10$ AND $c < 20$;
UPDATE $\widehat{\mathsf{T}}$ SET $a_1 = a_0$ WHERE $\neg(c > 10$ AND $c < 20)$;
(1a) UPDATE $\widehat{\mathsf{T}}$ SET $b_2 = b_0$;
(1b) UPDATE $\widehat{\mathsf{T}}$ SET $a_2 = a_1$;
else
(2) UPDATE $\widehat{\mathsf{T}}$ SET $b_1 = b_0 - 15$ WHERE $c > 25$ AND $c < 35$;
UPDATE $\widehat{\mathsf{T}}$ SET $b_1 = b_0$ WHERE $\neg(c > 25$ AND $c < 35)$;
(2a) UPDATE $\widehat{\mathsf{T}}$ SET $a_2 = a_0$;
(2b) UPDATE $\widehat{\mathsf{T}}$ SET $b_2 = b_1$;
endif

$$\mathsf{Path}(\Phi,\ \mathtt{skip}) = \Phi \qquad \mathsf{Path}(\Phi,\ \mathtt{assume}\ \phi^{dsa}) = \Phi\ \wedge\ \phi^{dsa} \qquad \mathsf{Path}(\Phi,\ v_{i+1} := e_i) = \Phi\ \wedge\ v_{i+1} = e_i \qquad \mathsf{Path}(\Phi,\ \mathtt{assert}\ \phi^{dsa}) = \Phi\ \wedge\ \phi^{dsa}$$

$$\mathsf{Path}(\Phi,\ \langle rs_{i+1} := \mathtt{SELECT}(f(\vec{e}_i), r(\vec{h}(\vec{a}_i)),\ \phi',\ g(\vec{e}_i)),\ cond_i\rangle) = \Phi \wedge ((cond_i \wedge rs_{i+1} = \mathsf{F}(\vec{a}_i)) \vee (\neg cond_i \wedge\ rs_{i+1} = rs_i))$$

$$\mathsf{Path}(\Phi,\ \langle \vec{a}_{i+1} := \mathsf{act}_i^1 \curvearrowright \mathsf{act}_i^2,\ cond_i^1 \curvearrowright cond_i^2\rangle) = \mathsf{Path}(\Phi, \langle \vec{a}_{i+1} := \mathsf{act}_i^1,\ cond_i^1\rangle) \vee \mathsf{Path}(\Phi, \langle \vec{a}_{i+1} := \mathsf{act}_i^2,\ cond_i^2\rangle)$$

$$\mathsf{Path}(\Phi,\ \langle \vec{a}_{i+1} := \mathtt{UPDATE}(\vec{e}_i),\ cond_i\rangle) = \Phi \wedge (cond_i \wedge \bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j = e_i^j) \qquad \mathsf{Path}(\Phi,\ \langle \vec{a}_{i+1} := \mathtt{INSERT}(\vec{e}_i),\ false\rangle) = \Phi\ \wedge\ (\bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j = e_i^j)$$

$$\mathsf{Path}(\Phi,\ \langle \vec{a}_{i+1} := \mathtt{DELETE}(),\ cond_i\rangle) = \Phi \wedge (\neg cond_i \wedge \bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j = a_i^j) \qquad \mathsf{Path}(\Phi,\ stmt_1^{dsa};\ stmt_2^{dsa}) = \mathsf{Path}(\mathsf{Path}(\Phi,\ stmt_1^{dsa}),\ stmt_2^{dsa})$$

$$\mathsf{Path}(\Phi,\ \mathtt{if}\ b_i\ \mathtt{then}\ stmt_1^{dsa}\ \mathtt{else}\ stmt_2^{dsa}) = \mathsf{Path}(\Phi \wedge b_i,\ stmt_1^{dsa}) \cup \mathsf{Path}(\Phi \wedge \neg b_i,\ stmt_2^{dsa})$$

Fig. 9. Encoding of database statements into logical formula.

Observe that Sync(①, ②) introduces statement 1a under the if-block of the resultant DSA form, followed by statement 1b which is obtained by Destruct(①). Similarly, statements 2a and 2b are introduced under the else-block.

In general, TranDSA can be implemented based on the standard DSA construction algorithm (Briggs et al., 1998) with an extension to cover SQL statements.

We are now in a position to propose Verification Condition Generation techniques, namely symbolic execution, conditional normal form, and weakest precondition, in the subsequent section.

3. Verification condition generation techniques

As we mentioned earlier that the fundamental step involved in the deductive-based approach is to generate VCs from program codes annotated with specifications. Once VCs are generated, they are fed to the theorem provers to check their validity which proves the correctness of the programs w.r.t. given specifications. Although VCs have been widely used in some well-known verification tools (Barnett et al., 2005; Clarke et al., 2004; Filliâtre and Paskevich, 2013), they have not been systematically analyzed by the research community. Authors in da Cruz et al. (2012), Frade and Pinto (2011) and Lourenço et al. (2015) first revisited various VC generation techniques for iteration free imperative programs and performed a detailed comparison in term of efficiency. To the best of our knowledge, this has never been explored in the realm of database applications where database statements are embedded within a general purpose host imperative language. Our main objective in this section is to extend these VC generation techniques, namely symbolic execution, conditional normal form and weakest precondition, to the case of database applications, enabling to formally verify underlying database properties and to perform a detailed comparative analysis with respect to performance.

3.1. Symbolic execution

The very first and simplest method to verify the correctness of a program is to generate logical formulas along all execution paths of a program using symbolic execution, taking the given specification into account. Symbolic execution considers symbolic input values (rather than concrete values) and therefore execution proceeds along all control paths covering the entire execution space of the programs. Notice that the method produces one VC for each *assert* command in a path and each VC encodes only the part of the program that is relevant for that assert command. Therefore, there exists a one-to-one relation between the execution path and the VC. The validity of each logical formula certifies that execution going through the corresponding path meets the assertions, and consequently, successful validation of all formulas ensures the program's correctness w.r.t. the given specification. This technique has an advantage from the point of view of *traceability*: an invalid VC allows one to immediately identify the executions that may violate a property.

Let us now formalize below two functions Path and VC^{se}. Note that, in order to indicate a statement in DSA form, we use either the superscript *dsa* or the subscripts i and $i+1$ as the current version and the updated version respectively. We denote by $\langle \vec{a}_{i+1} := \mathsf{act}_i^1 \curvearrowright \mathsf{act}_i^2,\ cond_i^1 \curvearrowright cond_i^2\rangle$ the DSA form of either UPDATE or DELETE or INSERT as follows:

$\langle \vec{a}_{i+1} := \mathtt{UPDATE}(\vec{a}_i) \curvearrowright \mathtt{UPDATE}(\vec{e}_i),\ \neg cond_i \curvearrowright cond_i\rangle$
$\langle \vec{a}_{i+1} := \mathtt{UPDATE}(\vec{a}_i) \curvearrowright \mathtt{INSERT}(\vec{e}_i),\ true \curvearrowright false\rangle$
$\langle \vec{a}_{i+1} := \mathtt{UPDATE}(\vec{a}_i) \curvearrowright \mathtt{DELETE}(),\ \neg cond_i \curvearrowright cond_i\rangle$

The functions Path and VC^{se} are defined below:

(1) Let Φ be a logical encoding of an execution path π up to program point ℓ. On encountering $stmt^{dsa}$ at program point $\ell+1$ along the path, the function $\mathsf{Path}(\Phi, stmt^{dsa})$ returns a logical formula that encodes π up to the program point $\ell+1$. Fig. 9 defines the function Path for all statements in our language. Observe that the helper function F(.) represents the composition of functions g, ϕ, h, r, f (if present) in SELECT.

(2) Given a program in the form of a sequence of statements, the function VC^{se} recursively collects the logical encoding of all program paths by invoking the auxiliary function Path defined above. Finally, on encountering an *assert* statement, the function VC^{se} returns a VC corresponding to an implication to the *assert* constraints. This is defined in Fig. 10.

The overall algorithm of symbolic execution-based VC generation is depicted as Algorithm 1. $\mathsf{CFG}(\mathcal{P}^{dsa})$ denotes the control-flow graph of $\mathcal{P}^{dsa}$.

Algorithm 1: SE-based VCG

Input: DSA form $\mathcal{P}^{dsa}$ of annotated database program $\mathcal{P}$
Output: A set of VCs

```
1 X := ∅;
2 for each path n_1 n_2 ... n_l ∈ CFG(P^dsa) do
3 |   Φ := ∅;
  |   // Let S = s_1; s_2; ...; s_n; where s_i is the statement
  |      in P^dsa corresponding to node n_i
4 |   X := X ∪ VC^se(Φ, S);
5 Return X;
6 End
```

$$\mathrm{VC}^{se}(\Phi,\ \mathtt{stmt}_1^{dsa};\ \mathtt{stmt}_2^{dsa}) = \mathrm{VC}^{se}(\Phi, \mathtt{stmt}_1^{dsa}) \cup \mathrm{VC}^{se}(\mathrm{Path}(\Phi, \mathtt{stmt}_1^{dsa}), \mathtt{stmt}_2^{dsa})$$

$$\mathrm{VC}^{se}(\Phi,\ \mathtt{if}\ b\ \mathtt{then}\ \mathtt{stmt}_1^{dsa}\ \mathtt{else}\ \mathtt{stmt}_2^{dsa}) = \mathrm{VC}^{se}(\Phi \wedge \mathrm{b}, \mathtt{stmt}_1^{dsa}) \cup \mathrm{VC}^{se}(\Phi \wedge \neg \mathrm{b}, \mathtt{stmt}_2^{dsa})$$

$$\mathrm{VC}^{se}(\Phi,\ \mathtt{assert}\ \phi^{dsa}) = \Phi \Rightarrow \phi^{dsa}$$

$$\mathrm{VC}^{se}(\Phi,\ \mathtt{assume}\ \phi^{dsa}) = \emptyset$$

$$\mathrm{VC}^{se}(\Phi,\ v_{i+1} := e_i) = \emptyset \qquad \mathrm{VC}^{se}(\Phi,\ \mathtt{skip}) = \emptyset$$

$$\mathrm{VC}^{se}(\Phi,\ Q_{sel}^{dsa}) = \emptyset \qquad \mathrm{VC}^{se}(\Phi,\ Q_{upd}^{dsa}) = \emptyset$$

$$\mathrm{VC}^{se}(\Phi,\ Q_{ins}^{dsa}) = \emptyset \qquad \mathrm{VC}^{se}(\Phi,\ Q_{del}^{dsa}) = \emptyset$$

Fig. 10. VC generation using symbolic execution.

Let us now illustrate the function VC^{se} on our motivating example in Fig. 1.

Example 1. Given the code snippet DBprog in Fig. 1, its DSA form annotated with assume and assert statements is shown in Fig. 11. Initially Φ is the empty set $\emptyset$. Consider the sequence of statements $\langle 6;\ 7;\ 8;\ \ldots\rangle$ in DBprog^{dsa}. According to the definition of VC^{se}, applying Algorithm 1 on the sequence of statements, $\mathrm{VC}^{se}(\Phi, \langle 6;\ 7;\ 8;\ \ldots\rangle)$ results in $\mathrm{VC}^{se}(\Phi, 6) \cup \mathrm{VC}^{se}(\mathrm{Path}(\Phi, \langle\ 6\ \rangle), \langle\ 7;\ 8;\ \ldots\ \rangle)$ which initiates recursive calls on the subproblems. The auxiliary function Path accumulates logical formulas along the program paths, and finally the function VC^{se}, on encountering the assert statement at program point 19, generates the following set of VCs.

VC_1 : $((\mathrm{MP}_0 \geqslant 80000 \wedge \mathrm{EQ}_0 \geqslant 60000 \wedge \mathrm{CT}_0 \geqslant 10000 \wedge \mathrm{CS}_0 \geqslant 5000) \wedge ((z_1 == \mathrm{TA}_0 \wedge \mathrm{Did}_0 == y_0) \vee (z_1 == z_0 \wedge \neg(\mathrm{Did}_0 == y_0))) \wedge (z_1 \geqslant x_0 \wedge m_1 == z_1 - x_0 \wedge n_1 == (m_1/4)) \wedge ((\mathrm{MP}_1 == \mathrm{MP}_0 - n_1 \wedge \mathrm{EQ}_1 == \mathrm{EQ}_0 - n_1 \wedge \mathrm{CT}_1 == \mathrm{CT}_0 - n_1 \wedge \mathrm{CS}_1 == \mathrm{CS}_0 - n_1 \wedge \mathrm{Did}_1 == x_0) \vee (\mathrm{MP}_1 == \mathrm{MP}_0 \wedge \mathrm{EQ}_1 == \mathrm{EQ}_0 \wedge \mathrm{CT}_1 == \mathrm{CT}_0 \wedge \mathrm{CS}_1 == \mathrm{CS}_0 \wedge \neg(\mathrm{Did}_1 == x_0))) \wedge (m_2 == m_1 \wedge n_2 == n_1) \wedge (\mathrm{MP}_2 == \mathrm{MP}_1 \wedge \mathrm{EQ}_2 == \mathrm{EQ}_1 \wedge \mathrm{CT}_2 == \mathrm{CT}_1 \wedge \mathrm{CS}_2 == \mathrm{CS}_1)) \Rightarrow (\mathrm{MP}_2 \geqslant 80000 \wedge \mathrm{EQ}_2 \geqslant 60000 \wedge \mathrm{CT}_2 \geqslant 10000 \wedge \mathrm{CS}_2 \geqslant 5000)$

VC_2 : $((\mathrm{MP}_0 \geqslant 80000 \wedge \mathrm{EQ}_0 \geqslant 60000 \wedge \mathrm{CT}_0 \geqslant 10000 \wedge \mathrm{CS}_0 \geqslant 5000) \wedge ((z_1 == \mathrm{TA}_0 \wedge \mathrm{Did}_0 == y_0) \vee (z_1 == z_0 \wedge \neg(\mathrm{Did}_0 == y_0))) \wedge \neg(z_1 \geqslant x_0) \wedge \mathrm{MP}_2 == \mathrm{MP}_0 \wedge \mathrm{EQ}_2 == \mathrm{EQ}_0 \wedge \mathrm{CT}_2 == \mathrm{CT}_0 \wedge \mathrm{CS}_2 == \mathrm{CS}_0 \wedge m_2 == m_0 \wedge n_2 == n_0) \Rightarrow (\mathrm{MP}_2 \geqslant 80000 \wedge \mathrm{EQ}_2 \geqslant 60000 \wedge \mathrm{CT}_2 \geqslant 10000 \wedge \mathrm{CS}_2 \geqslant 5000)$

The fact that VC_1 is not satisfied in some cases (for example, $z_1 = 20, x_0 = 12, \mathrm{MP}_2 = 79999, \mathrm{EQ}_2 = 59\,999, \mathrm{CT}_2 = 9999, \mathrm{CS}_2 = 4999$) indicates that DBprog does not respect the given specification all the time. In particular, the violation happens due to statement 10 in DBprog where attribute values are subtracted by some amount which is influenced by a runtime input.

Limitation. The major problem of symbolic execution-based VCG is that the number of VCs will be exponential in the worst case scenario. For instance, for a program with n conditionals the number of VCs may be in $O(2^n)$.

3.2. Conditional normal form

We have seen in the previous section that VCG using symbolic execution leads to an exponential number of VCs in the worst case scenario. To overcome this problem, we now describe the second method of VCG which was first used in Bounded Model Checking of software (BMC) (Clarke et al., 2004) as a way to unroll loops avoiding path enumerations. In the BMC technique, the transformation of iteration free DSA programs form is basically guided by the following three rules (Clarke et al., 2004):

```
1. CREATE OR REPLACE PROCEDURE DBprog (y0 int, x0 int)
     IS
2.  z0 int;
3.  m0 int;
4.  n0 int;
5.  BEGIN
6.    assume MP0 ⩾ 80000 and EQ0 ⩾ 60000 and CT0 ⩾ 10000 and
      CS0 ⩾ 5000;
7.    SELECT TA0 INTO z1 FROM Budget WHERE Did0= y0;
8.    if z1 ⩾ x0 then
9.        m1 := z1 − x0;
10.       n1 := m1/4;
11.       UPDATE Budget SET MP1 = MP0−n1, EQ1 = EQ0−n1,
          CT1 = CT0−n1, CS1 = CS−n1 WHERE Did0 = x0;
          UPDATE Budget SET MP1 = MP0, EQ1 = EQ0, CT1 =
           CT0, CS1 = CS0 WHERE ¬(Did0 = x0);
12.       m2 := m1;
13.       n2 := n1;
14.       UPDATE Budget SET MP2 = MP1, EQ2 = EQ1,
          CT2 = CT1, CS2 = CS1;
      else
15.       UPDATE Budget SET MP2 = MP0, EQ2 = EQ0,
          CT2 = CT0, CS2 = CS0;
16.       m2 := m0;
17.       n2 := n0;
18.   endif;
19.   assert MP2 ⩾ 80000 and EQ2 ⩾ 60000 and CT2 ⩾ 10000 and
      CS2 ⩾ 5000;
20. end;
```

Fig. 11. DBprog^{dsa}: DSA form of DBprog.

R_1 : $\mathtt{if}\ (b^{dsa})\ stmt_1^{dsa}\ \mathtt{else}\ stmt_2^{dsa} \Rightarrow \mathtt{if}\ (b^{dsa})\ stmt_1^{dsa}; \mathtt{if}\ (\neg b^{dsa})\ stmt_2^{dsa};$

R_2 : $\mathtt{if}\ (b^{dsa})\ \{stmt_1^{dsa};\ stmt_2^{dsa}\} \Rightarrow \mathtt{if}\ (b^{dsa})\ stmt_1^{dsa};\ \mathtt{if}\ (b^{dsa})\ stmt_2^{dsa};$

R_3 : $\mathtt{if}\ (b_1^{dsa})\ \{\mathtt{if}\ (b_2^{dsa})\ stmt^{dsa}\} \Rightarrow \mathtt{if}\ (b_1^{dsa} \wedge b_2^{dsa})\ stmt^{dsa}$

The first rule R_1 states that branches of conditional statements can be sequentialized. The second rule R_2 states that conditions can be distributed through the sequence of statements present in the body of conditional statements. The third rule R_3 states that nested conditions can be combined together. Observe that the above rules rewrite a given DSA program into another semantically equivalent form where every statement $stmt^{dsa}$ is guarded by a condition b^{dsa}. If a $stmt^{dsa}$ is not in the body of any conditional statement then it will be guarded by *true*. This representation of the DSA programs is known as Conditional Normal Form (CNF).

Let us now define the CNF-based VCG for our database language by following the below steps:

(1) Transformation of database program $\mathcal{P}^{dsa}$ in DSA form into its equivalent CNF form $\mathcal{P}^{cnf}$. This is defined by function toCNF(.): $\mathbb{C}^{dsa} \times \Upsilon \rightarrow \mathbb{C}^{cnf}$, which transforms a given statement $c^{dsa} \in \mathbb{C}^{dsa}$ into its equivalent CNF form $c^{cnf} \in \mathbb{C}^{cnf}$ under the path condition $\rho \in \Upsilon$ upon which the execution of c^{dsa} depends.

$$\mathrm{toCNF}(\rho,\ \mathtt{skip}) = \mathtt{if}\ \rho\ \mathtt{then}\ \mathtt{skip}$$

$$\mathrm{toCNF}(\rho,\ v_i := e_i) = \mathtt{if}\ \rho\ \mathtt{then}\ v_i := e_i$$

$$\mathrm{toCNF}(\rho,\ \mathit{assume}\ \phi^{dsa}) = \mathtt{if}\ \rho\ \mathtt{then}\ \mathit{assume}\ \phi^{dsa}$$

$$\mathrm{toCNF}(\rho,\ Q_{sel}^{dsa}) = \mathtt{if}\ \rho\ \mathtt{then}\ Q_{sel}^{dsa} \qquad \mathrm{toCNF}(\rho,\ Q_{ins}^{dsa}) = \mathtt{if}\ \rho\ \mathtt{then}\ Q_{ins}^{dsa}$$

$$\mathrm{toCNF}(\rho,\ Q_{upd}^{dsa}) = \mathtt{if}\ \rho\ \mathtt{then}\ Q_{upd}^{dsa} \qquad \mathrm{toCNF}(\rho,\ Q_{del}^{dsa}) = \mathtt{if}\ \rho\ \mathtt{then}\ Q_{del}^{dsa}$$

$$\mathrm{toCNF}(\rho,\ stmt_1^{dsa};\ stmt_2^{dsa}) = \mathrm{toCNF}(\rho,\ stmt_1^{dsa}); \mathrm{toCNF}(\rho,\ stmt_2^{dsa})$$

$$\mathrm{toCNF}(\rho,\ \mathtt{if}\ b\ \mathtt{then}\ stmt_1^{dsa}\ \mathtt{else}\ stmt_2^{dsa}\ \mathtt{endif}) = \mathrm{toCNF}(\rho \wedge b,\ stmt_1^{dsa}); \mathrm{toCNF}(\rho \wedge \neg b,\ stmt_2^{dsa})$$

$$\mathrm{toCNF}(\rho,\ \mathit{assert}\ \phi^{dsa}) = \mathtt{if}\ \rho\ \mathtt{then}\ \mathit{assert}\ \phi^{dsa}$$

(2) Extraction of two sets of formulas f_{stmt} and f_{pr} from each statement in $\mathcal{P}^{cnf}$. To this purpose, we define the function

$$\mathsf{Conf}(\texttt{if } b_i \texttt{ then } \langle rs_{i+1} := \texttt{SELECT}(f(\vec{e}), r(\vec{h}(\vec{x})), \phi, g(\vec{e})), cond_i\rangle) = (\{(b_i \wedge cond_i \Rightarrow rs_{i+1} = \mathsf{F}(\vec{a}_i)) \vee (b_i \wedge \neg cond_i \Rightarrow rs_{i+1} = rs_i)\}, \emptyset)$$

$$\mathsf{Conf}(\langle \vec{a}_{i+1} := \mathsf{act}_i^1 \curvearrowright \mathsf{act}_i^2,\ cond_i^1 \curvearrowright cond_i^2\rangle) = \mathsf{Conf}(\langle \vec{a}_{i+1} := \mathsf{act}_i^1,\ cond_i^1\rangle) \vee \mathsf{Conf}(\langle \vec{a}_{i+1} := \mathsf{act}_i^2,\ cond_i^2\rangle) \qquad \mathsf{Conf}(\texttt{if } b_i \texttt{ then skip}) = (\emptyset, \emptyset)$$

$$\mathsf{Conf}(\texttt{if } b_i \texttt{ then } \langle \vec{a}_{i+1} := \texttt{UPDATE}(\vec{e}_i),\ cond_i\rangle) = (\{(b \wedge cond_i \Rightarrow \bigwedge_{i=1}^{|\vec{a}_i|}(\vec{a}_{i+1} = \vec{e}_i))\},\ \emptyset) \qquad \mathsf{Conf}(\texttt{if } b_i \texttt{ then } v_i := e_i) = (\{b_i \Rightarrow v_i = e_i\},\ \emptyset)$$

$$\mathsf{Conf}(\texttt{if } b_i \texttt{ then } \langle \vec{a}_{i+1} := \texttt{INSERT}(\vec{e}_i), false\rangle) = (\{b \Rightarrow (\bigwedge_{i=1}^{|\vec{a}_i|}(\vec{a}_{i+1} = \vec{e}_i)\},\ \emptyset) \qquad \mathsf{Conf}(\texttt{if } b_i \texttt{ then } \textit{assume } \phi_i) = (\{b_i \Rightarrow \phi_i\},\ \emptyset)$$

$$\mathsf{Conf}(\texttt{if } b_i \texttt{ then } \langle \vec{a}_{i+1} := \texttt{DELETE}(),\ cond_i\rangle) = (\{\rho \wedge \neg cond_i \Rightarrow \bigwedge_{i=1}^{|\vec{a}_i|}(\vec{a}_{i+1} = \vec{a}_i))\},\ \emptyset) \qquad \mathsf{Conf}(\texttt{if } b_i \texttt{ then } \textit{assert } \phi_i) = (\emptyset,\ \{b_i \Rightarrow \phi_i\})$$

$$\mathsf{Conf}(stmt_1^{dsa};\ stmt_2^{dsa}) = (\mathsf{f}_{stmt_1} \cup \mathsf{f}_{stmt_2},\ \mathsf{f}_{pr_1} \cup \mathsf{f}_{pr_2}),\ where\ \mathsf{Conf}(stmt_1^{dsa}) = (\mathsf{f}_{stmt_1},\ \mathsf{f}_{pr_1}),\ \mathsf{Conf}(stmt_2^{dsa}) = (\mathsf{f}_{stmt_2},\ \mathsf{f}_{pr2})$$

Fig. 12. Function to compute $\bigwedge \mathsf{f}_{stmt}$ and $\bigwedge \mathsf{f}_{pr}$.

$$\mathsf{VC}^{cnf}(\rho,\ \texttt{skip}) = (\emptyset,\ \emptyset) \qquad \mathsf{VC}^{cnf}(\rho,\ v_i := e_i) = (\{\rho \Rightarrow v_i = e_i\},\ \emptyset) \qquad \mathsf{VC}^{cnf}(\rho,\ \textit{assume } \phi_i) = (\{\rho \Rightarrow \phi_i\},\ \emptyset)$$

$$\mathsf{VC}^{cnf}(\rho, \langle \vec{a}_{i+1} := \texttt{UPDATE}(\vec{e}_i),\ cond_i\rangle) = (\{(\rho \wedge cond_i \Rightarrow \bigwedge_{i=1}^{|\vec{a}_i|}(\vec{a}_{i+1} = \vec{e}_i))\},\ \emptyset) \qquad \mathsf{VC}^{cnf}(\rho,\ \langle \vec{a}_{i+1} := \texttt{INSERT}(\vec{e}_i), false\rangle) = (\{\rho \Rightarrow (\bigwedge_{i=1}^{|\vec{a}_i|}(\vec{a}_{i+1} = \vec{e}_i))\},\ \emptyset)$$

$$\mathsf{VC}^{cnf}(\rho,\ \langle \vec{a}_{i+1} := \texttt{DELETE}(),\ cond_i\rangle) = (\{\rho \wedge \neg cond_i \Rightarrow \bigwedge_{i=1}^{|\vec{a}_i|}(\vec{a}_{i+1} = \vec{a}_i))\},\ \emptyset) \qquad \mathsf{VC}^{cnf}(\rho,\ \textit{assert } \phi_i) = (\emptyset,\ \{\rho \Rightarrow \phi_i\})$$

$$\mathsf{VC}^{cnf}(\rho,\ \langle rs_{i+1} := \texttt{SELECT}(f(\vec{e}), r(\vec{h}(\vec{x})), \phi, g(\vec{e})), cond_i\rangle) = (\{(\rho \wedge cond_i \Rightarrow rs_{i+1} = \mathsf{F}(\vec{a}_i)) \vee (\rho \wedge \neg cond_i \Rightarrow rs_{i+1} = rs_i)\}, \emptyset)$$

$$\mathsf{VC}^{cnf}(\rho,\ \langle \vec{a}_{i+1} := \mathsf{act}_i^1 \curvearrowright \mathsf{act}_i^2,\ cond_i^1 \curvearrowright cond_i^2\rangle) = \mathsf{VC}^{cnf}(\rho,\ \langle \vec{a}_{i+1} := \mathsf{act}_i^1,\ cond_i^1\rangle) \vee \mathsf{VC}^{cnf}(\rho,\ \langle \vec{a}_{i+1} := \mathsf{act}_i^2,\ cond_i^2\rangle)$$

$$\mathsf{VC}^{cnf}(\rho,\ stmt_1^{dsa};\ stmt_2^{dsa}) = (\mathsf{f}_{stmt_1} \cup \mathsf{f}_{stmt_2},\ \mathsf{f}_{pr1} \cup \mathsf{f}_{pr2}),\ where\ (\mathsf{f}_{stmt_1},\ \mathsf{f}_{pr1}) = \mathsf{VC}^{cnf}(\rho,\ stmt_1^{dsa}), (\mathsf{f}_{stmt_2},\ \mathsf{f}_{pr2}) = \mathsf{VC}^{cnf}(\rho,\ stmt_2^{dsa})$$

$$\mathsf{VC}^{cnf}(\rho,\ \texttt{if } b_i \texttt{ then } stmt_1^{dsa} \texttt{ else } stmt_2^{dsa} \texttt{ endif}) = (\mathsf{f}_{stmt_1} \cup \mathsf{f}_{stmt_2},\ \mathsf{f}_{pr1} \cup \mathsf{f}_{pr2}),\ where\ (\mathsf{f}_{stmt_1},\ \mathsf{f}_{pr1}) = \mathsf{VC}^{cnf}(\rho \wedge b_i,\ stmt_1^{dsa}), (\mathsf{f}_{stmt_2},\ \mathsf{f}_{pr2}) = \mathsf{VC}^{cnf}(\rho \wedge \neg b_i,\ stmt_2^{dsa})$$

Fig. 13. Function to generate CNF-based VC.

Conf(.): $\mathbb{C}^{cnf} \rightarrow \mathbb{W} \times \mathbb{W}$, where $\mathbb{W}$ is the set of well-formed first order logic formulas and $\mathsf{f}_{stmt}, \mathsf{f}_{pr} \in \mathbb{W}$. This is depicted in Fig. 12. Finally, a single verification condition is constructed in the form of $\bigwedge \mathsf{f}_{stmt} \Rightarrow \bigwedge \mathsf{f}_{pr}$, where f_{stmt} contains the encoding of program statements and f_{pr} contains the encoding of all *assert* commands in the program. Note that, unlike symbolic execution, in this technique only one verification condition is generated.

Fig. 13 depicts the function VC^{cnf} which combines these two steps and generates a VC from a given program in DSA form. Algorithm 2 is the overall algorithm to generate the CNF-based VC. Example 2 illustrates the CNF-based VCG technique.

Algorithm 2: CNF-based VCG

Input: DSA form $\mathcal{P}^{dsa}$ of annotated database program $\mathcal{P}$
Output: Single VC

1 ρ := *true*;
2 Let $\mathcal{P}^{dsa}$ be a sequence of statements $s_1; s_2; \ldots; s_n$;
3 $(\mathsf{f}_{stmt}, \mathsf{f}_{pr})$:= $\mathsf{VC}^{cnf}(\rho, \mathcal{P}^{dsa})$
4 VC := $(\bigwedge \mathsf{f}_{stmt} \Rightarrow \bigwedge \mathsf{f}_{pr})$
5 Return VC
6 **End**

Example 2. Given the DSA form of DBprog in Fig. 11, its CNF form applying ToCNF(.) as depicted in Fig. 14. Fig. 15(a) shows the output formula set f_{stmt} and f_{pr} obtained by applying Conf(.). Therefore, the VC is constructed by combining f_{stmt} and f_{pr} in the form $\bigwedge \mathsf{f}_{stmt} \Rightarrow \bigwedge \mathsf{f}_{pr}$ is depicted in Fig. 15(b).

Observe that, similarly to when the symbolic execution technique was used, the validity checking of VC^{cnf} fails in some cases, which indicate a violation of the property MP $\geqslant$ 80000 $\wedge$ EQ $\geqslant$ 60000 $\wedge$ CT $\geqslant$ 10000 $\wedge$ CS $\geqslant$ 5000.

3.3. Weakest precondition

Hoare logic is a widely-used deductive verification formalism for computer programs (Hoare, 1969). The axioms and inference rules of this proof system are based on Hoare triples of the form {Pre}*stmt*{Post}. This means any terminating execution of the statement *stmt* on a state satisfying the precondition Pre results in a new state satisfying the postcondition Post.

Dijkstra et al. (1976) introduced predicate transformers, a set of rules for transforming predicates on states, as a way to specify program semantics. In particular, he defined "Weakest Precondition (wp)" and "Strongest Postcondition (sp)", treating assertions like preconditions and postconditions as predicates on program states. The predicate transformers technique generates VCs by propagating predicates either backward (weakest preconditions) or forward (strongest postconditions) along the program. As VCG based on the strongest postcondition is costly due to the presence of existential quantifiers in the formula, in this section, we extend VCG based on the weakest precondition to the case of database language. Given a program statement *stmt* and a postcondition ψ, the weakest precondition of an iteration free imperative program is computed as follows:

```
1. CREATE OR REPLACE PROCEDURE Proc_Budget_Adjust
     (y_0 int, x_0 int) IS
                ⋮
5. BEGIN
6.    if true then
7.       assume MP_0 ⩾ 80000 and EQ_0 ⩾ 60000 and CT_0 ⩾ 10000
     and CS_0 ⩾ 5000;
9.    endif
7.    if true then
8.       SELECT TA_0 INTO z_1 FROM Budget WHERE Did_0 = y_0;
9.    endif
10.   if z_1 ⩾ x_0 then
11.      m_1 := z_1 − x_0;
12.    endif
                ⋮
                ⋮
38.    if true then
39.      assert MP_2 ⩾ 80000 and EQ_2 ⩾ 60000 and CT_2 ⩾ 10000 and
     CS_2 ⩾ 5000;
40.   endif
41. end;
```

Fig. 14. DBprog^{cnf}: CNF form of DBprog^{dsa}.

f_{stmt} : { $(true \Rightarrow MP_0 \geqslant 80000 \wedge EQ_0 \geqslant 60000 \wedge CT_0 \geqslant 10000 \wedge CS_0 \geqslant 5000)$
$(true \Rightarrow ((z_1 == TA_0 \wedge Did_0 == y_0) \vee (z_1 == z_0 \wedge \neg(Did_0 == y_0))))$,
$(z_1 \geqslant x_0 \Rightarrow m_1 == z_1 - x_0), (z_1 \geqslant x_0 \Rightarrow n_1 == m_1/4)$, $((z_1 \geqslant x_0 \wedge$
$Did_1 == x_0 \Rightarrow (MP_1 == MP_0 - n_1 \wedge EQ_1 == EQ_0 - n_1\ CT_1 == CT_0 \wedge$
$- n_1 \wedge CS_1 == CS_0 - n_1)) \vee (z_1 \geqslant x_0 \wedge \neg(Did_1 == x_0) \Rightarrow (MP_1 ==$
$MP_0 \wedge EQ_1 == EQ_0 \wedge CT_1 == CT_0 \wedge CS_1 == CS_0))$, $(z_1 \geqslant x_0 \Rightarrow$
$m_2 == m_1)$, $(z_1 \geqslant x_0 \Rightarrow n_2 == n_1)$, $(z_1 \geqslant x_0 \Rightarrow MP_2 == MP_1 \wedge EQ_2$
$== EQ_1 \wedge CT_2 == CT_1 \wedge CS_2 ==\ CS_1)$, $(\neg(z_1 \geqslant x_0) \Rightarrow MP_2 == MP_0$
$\wedge EQ_2 == EQ_0 \wedge CT_2 == CT_0 \wedge CS_2 == CS_0)$, $(\neg(z_1 \geqslant x_0) \Rightarrow$
$m_2 == m_0)$, $(\neg(z_1 \geqslant x_0) \Rightarrow n_2 == n_0)$}
f_{pr} : { $(true \Rightarrow MP_2 \geqslant 80000 \wedge EQ_2 \geqslant 60000 \wedge CT_2 \geqslant 10000 \wedge CS_2 \geqslant 5000)$}

(a) Formula sets f_{stmt} and f_{pr} from DBprog^{cnf} by applying Conf(.)

VC : $((true \Rightarrow MP_0 \geqslant 80000 \wedge EQ_0 \geqslant 60000 \wedge CT_0 \geqslant 10000 \wedge CS_0 \geqslant 5000) \wedge ($
$true \Rightarrow ((z_1 == TA_0 \wedge Did_0 == y_0) \vee (z_1 == z_0 \wedge \neg(Did_0 == y_0)))) \wedge\ (z_1 \geqslant$
$x_0 \Rightarrow m_1 == z_1 - x_0) \wedge (z_1 \geqslant x_0 \Rightarrow n_1 == m_1/4) \wedge ((z_1 \geqslant x_0 \wedge Did_1 == x_0$
$\Rightarrow (MP_1 == MP_0 - n_1 \wedge EQ_1 == EQ_0 - n_1 \wedge CT_1 == CT_0 - n_1 \wedge CS_1 ==$
$CS_0 - n_1)) \vee (z_1 \geqslant x_0 \wedge \neg(Did_1 == x_0) \Rightarrow (MP_1 == MP_0 \wedge EQ_1 == EQ_0$
$\wedge CT_1 == CT_0 \wedge CS_1 == CS_0)) \wedge (z_1 \geqslant x_0 \Rightarrow m_2 == m_1) \wedge (z_1 \geqslant x_0 \Rightarrow n_2$
$== n_1) \wedge (z_1 \geqslant x_0 \Rightarrow MP_2 == MP_1 \wedge EQ_2 == EQ_1 \wedge CT_2 == CT_1 \wedge CS_2$
$== CS_1) \wedge (\neg(z_1 \geqslant x_0) \Rightarrow MP_2 == MP_0 \wedge EQ_2 == EQ_0 \wedge CT_2 == CT_0 \wedge$
$CS_2 == CS_0) \wedge (\neg(z_1 \geqslant x_0) \Rightarrow m_2 == m_0) \wedge (\neg(z_1 \geqslant x_0) \Rightarrow n_2 == n_0))) \Rightarrow$
$(true \Rightarrow MP_2 \geqslant 80000\ \wedge EQ_2 \geqslant 60000 \wedge CT_2 \geqslant 10000 \wedge CS_2 \geqslant 5000)$

(b) Verification Condition generated based on f_{stmt} and f_{pr}

Fig. 15. Verification Condition of DBprog^{dsa} based on CNF.

$$\mathtt{wp}(\mathtt{skip},\ \psi) = \psi \qquad \mathtt{wp}(v := e,\ \psi) = \psi[e/v]$$
$$\mathtt{wp}(stmt_1;\ stmt_2,\ \psi) = \mathtt{wp}(stmt_1,\ \mathtt{wp}(stmt_2,\ \psi))$$
$$\mathtt{wp}(\mathtt{if}\ b\ \mathtt{then}\ stmt_1\ \mathtt{else}\ stmt_2\ \mathtt{endif}, \psi) = (b \wedge \mathtt{wp}(stmt_1, \psi)) \vee (\neg b \wedge \mathtt{wp}(stmt_2, \psi))$$

In order to adopt VCG of database programs, we define wp on database statements in Fig. 16. Observe that, this process does not require to convert input programs into DSA form.

Given a program $\mathcal{P}$ annotated with *assume* and *assert* in the form {*assume* ψ_1; $\mathcal{P}$; *assert* ψ_2;}, the VC is constructed as follows: $\mathtt{wp}(assume\ \psi_1, \mathtt{wp}(\mathcal{P}, \psi_2)) = \psi_1 \Rightarrow \mathtt{wp}(\mathcal{P}, \psi_2)$. The complete algorithmic steps to generate the wp-based VC are depicted in Algorithm 3 and this is illustrated with our motivating example in Example 3.

$$\mathtt{wp}(\mathtt{skip},\ \psi) = \psi \qquad \mathtt{wp}(v := e,\ \psi) = \psi[e/v]$$
$$\mathtt{wp}(assume\ \psi_1,\ \psi_2) = \psi_1 \Rightarrow \psi_2 \qquad \mathtt{wp}(assert\ \psi_1,\ \psi_2) = \psi_1 \wedge \psi_2$$
$$\mathtt{wp}(\langle rs := \mathtt{SELECT}(f(\vec{e}), r(\vec{h}(\vec{x})), \phi, g(\vec{e})), cond\rangle, \psi) = (\psi[\mathrm{F}(\vec{a})/rs] \wedge cond) \vee (\psi \wedge \neg cond)$$
$$\mathtt{wp}(\langle \vec{a} := \mathtt{UPDATE}(\vec{e}),\ cond\rangle, \psi) = ((\psi \wedge \neg cond) \vee (\psi[\vec{e}/\vec{a}] \wedge cond))$$
$$\mathtt{wp}(\langle \vec{a} := \mathtt{INSERT}(\vec{e}),\ false\rangle, \psi) = \psi[\vec{e}/\vec{a}] \vee \psi$$
$$\mathtt{wp}(\langle \vec{a} := \mathtt{DELETE}(),\ cond\rangle, \psi) = \psi \wedge \neg cond$$
$$\mathtt{wp}(stmt_1;\ stmt_2,\ \psi) = \mathtt{wp}(stmt_1,\ \mathtt{wp}(stmt_2,\ \psi))$$
$$\mathtt{wp}(\mathtt{if}\ b\ \mathtt{then}\ stmt\ \mathtt{endif},\ \psi) = (b \wedge \mathtt{wp}(stmt,\ \psi)) \vee (\neg b \wedge \psi)$$
$$\mathtt{wp}(\mathtt{if}\ b\ \mathtt{then}\ stmt_1\ \mathtt{else}\ stmt_2\ \mathtt{endif}, \psi) = (b \wedge \mathtt{wp}(stmt_1,\ \psi)) \vee (\neg b \wedge \mathtt{wp}(stmt_2,\ \psi))$$

Fig. 16. wp computation on our database language.

```
1. CREATE OR REPLACE PROCEDURE DBprog (y int, x int) IS
2. z int;
3. m int;
4. n int;
5. assume MP ⩾ 80000 ∧ EQ ⩾ 60000 ∧ CT ⩾ 10000 ∧ CS ⩾ 5000;
6. BEGIN
7.    SELECT TA INTO z FROM Budget WHERE Did = y;
8.    if (z ⩾ x) then
9.       m := z − x;
10.      n := m/4;
11.      UPDATE Budget SET MP = MP−n, EQ = EQ − n, CT = CT − n, CS
       = CS − n WHERE Did = y;
12.   endif;
13. assert MP ⩾ 80000 ∧ EQ ⩾ 60000 ∧ CT ⩾ 10000 ∧ CS ⩾ 5000;
14. end;
```

Fig. 17. Annotated form of DBprog.

Algorithm 3: Weakest Precondition Computation

Input: Database program $\mathcal{P}$, specification ψ
Output: Single VC

1 Let $\mathcal{P}$ be a sequence of statements $s_1; s_2; \ldots; s_n$;
2 Annotate $\mathcal{P}$ in the form of *assume* ψ_1; $\mathcal{P}$; *assert* ψ_2;
3 Apply $\mathtt{wp}(\mathcal{P},\ \psi_2)$ which results into ψ_3
4 Apply $\mathtt{wp}(assume\ \psi_1,\ \psi_3)$ which generates the VC as $\psi_1 \Rightarrow \psi_3$
5 Return VC
6 **End**

Example 3. Consider the motivating program DBprog depicted in Fig. 1. The annotated form of DBprog is depicted in Fig. 17.

The verification condition using weakest precondition wp of the above annotated program is computed as follows:

Let $c \triangleq$ $m := z - x$;
$n := m/4$;
UPDATE *Budget* SET MP = MP − n, EQ = EQ − n, CT = CT − n,
CS = CS − n WHERE Did = y;
and $\psi \triangleq$ MP ⩾ 80000 ∧ EQ ⩾ 60000 ∧ CT ⩾ 10000 ∧ CS ⩾ 5000
Then $\mathtt{wp}(c,\ \psi) \triangleq \mathtt{wp}([$
$m := z - x$;
$n := m/4$;
UPDATE *Budget* SET MP = MP − n, EQ = EQ − n,
CT = CT − n, CS = CS − n WHERE Did = y;
$],\ \psi)$
$\triangleq \mathtt{wp}([$
$m := z - x$;
$n := m/4$;

$$
\begin{aligned}
&\quad\ ],\ \psi_1)\\
&\quad\ \vdots\\
&= \psi_3\\
\text{Therefore, } &\ \mathsf{wp}([\\
&\quad \texttt{SELECT TA INTO } z \texttt{ FROM } \mathit{Budget} \texttt{ WHERE } \mathrm{Did} = y;\\
&\quad \texttt{if } (z \geqslant x) \texttt{ then}\\
&\quad\quad c\\
&\quad],\ \psi)\\
&\triangleq \mathsf{wp}([\\
&\quad \texttt{SELECT TA INTO } z \texttt{ FROM } \mathit{Budget} \texttt{ WHERE } \mathrm{Did} = y;\\
&\quad],\ \psi_4)\\
&\text{where } \psi_4 = (z \geqslant x \wedge\ \psi_3) \vee (\neg(z \geqslant x) \wedge \psi)\\
&= \psi_5
\end{aligned}
$$

Finally, VC is generated as follows:

$$
\begin{aligned}
&\mathsf{wp}([\\
&\quad \mathit{assume}\ \mathrm{MP} \geqslant 80000 \wedge \mathrm{EQ} \geqslant 60000 \wedge \mathrm{CT} \geqslant 10000 \wedge \mathrm{CS} \geqslant 5000\\
&\quad],\ \psi_5)\\
&= \mathrm{MP} \geqslant 80000 \wedge \mathrm{EQ} \geqslant 60000 \wedge \mathrm{CT} \geqslant 10000 \wedge \mathrm{CS} \geqslant 5000 \implies \psi_5\\
&= (\mathrm{MP} \geqslant 80000 \wedge \mathrm{EQ} \geqslant 60000 \wedge \mathrm{CT} \geqslant 10000 \wedge \mathrm{CS} \geqslant 5000) \implies ((\mathrm{Did} = y \wedge (\mathrm{TA}\\
&\quad \geqslant x \wedge ((\mathrm{Did} = y \wedge \mathrm{MP} - (\mathrm{TA} - x)/4 \geqslant 80000 \wedge \mathrm{EQ} - (\mathrm{TA} - x)/4 \geqslant 60000 \wedge \mathrm{CT} -\\
&\quad (\mathrm{TA} - x)/4 \geqslant 10000 \wedge \mathrm{CS} - (\mathrm{TA} - x)/4 \geqslant 5000) \vee (\neg(\mathrm{Did} = y) \wedge \mathrm{MP} \geqslant 80000\ \wedge\\
&\quad \mathrm{EQ} \geqslant 60000 \wedge \mathrm{CT} \geqslant 10000 \wedge \mathrm{CS} \geqslant 5000))) \vee ((\neg(\mathrm{TA} \geqslant x) \wedge \mathrm{MP} \geqslant 80000 \wedge \mathrm{EQ}\\
&\quad \geqslant 60000 \wedge \mathrm{CT} \geqslant 10000\ \ \wedge \mathrm{CS} \geqslant 5000))) \vee\ \ (\neg(\mathrm{Did} = y) \wedge (z \geqslant x \wedge ((\mathrm{Did} = y \wedge\\
&\quad \mathrm{MP} - (z - x)/4 \geqslant 80000 \wedge \mathrm{EQ} - (z - x)/4 \geqslant 60000 \wedge \mathrm{CT} - (z - x)/4 \geqslant 10000 \wedge\\
&\quad \mathrm{CS} - (z - x)/4 \geqslant 5000) \vee (\neg(\mathrm{Did} = y) \wedge \mathrm{MP} \geqslant 80000\ \ \wedge \mathrm{EQ} \geqslant 60000 \wedge \mathrm{CT} \geqslant\\
&\quad 10000 \wedge \mathrm{CS} \geqslant 5000))) \vee ((\neg(z \geqslant x) \wedge \mathrm{MP} \geqslant 80000 \wedge \mathrm{EQ} \geqslant 60000 \wedge \mathrm{CT}\\
&\quad \geqslant 10000 \wedge \mathrm{CS} \geqslant 5000))))
\end{aligned}
$$

Note that, similarly to the SE and CNF methods, in this case validity checking of VC also fails in some cases, indicating property violation.

Limitations. Although this approach generates a single verification condition, the length of the VC may be exponential w.r.t. program size. For instance, in the case of `if-else` statement, the weakest precondition computation considers both branches into a single formula, increasing its length by duplicating the propagated condition. Therefore, a program with n conditional statements generates a single VC of length $O(2^n)$.

Efficient weakest precondition

We observed that for non-DSA programs, the weakest precondition technique produces VCs whose size is, in the worse case, exponential with respect to the size of the program. Flanagan and Saxe showed that when the technique was applied to programs in DSA form, the size of the generated VCs was, in the worst case, quadratic (Flanagan and Saxe, 2001). The main point to understanding the simplified definition of predicate transformers for DSA programs is to observe that the set of execution paths of such a program can be encoded logically in a compact way that does not require duplicating assert formula. We call this encoding the *program formula*. The program formula of an assignment statement is simply the corresponding equality, and the formula of a sequence of statements is the conjunction of formulas of the sub-statements. For conditional, the formula is $(b \wedge \psi_1) \vee (\neg b \wedge \psi_2)$, where ψ_1 and ψ_2 are the formulas of the branch statements. The program formula of various statements of the language under consideration are depicted in Fig. 18, where the function $\Im(\mathit{stmt}^{dsa})$ denotes the encoding of stmt^{dsa}.

Authors in Leino and Rustan (2005) later proposed a simplified solution emphasizing that the technique could be seen as a special case of weakest precondition computation by introducing the following "dream property".

$$\mathsf{wlp}(\mathit{stmt}^{dsa},\ \psi) \triangleq \Im(\mathit{stmt}^{dsa}) \Rightarrow \psi$$

$$
\begin{aligned}
\Im(\texttt{skip}) = \top \quad \Im(\texttt{assume } \phi^{dsa}) = \phi^{dsa} \quad \Im(v_{i+1} := e_i) &= v_{i+1} = e_i\\
\Im(\langle rs_{i+1} := \texttt{SELECT}(f(\vec{e}_i), r(\vec{h}(\vec{a}_i)), \phi', g(\vec{e}_i)), cond_i\rangle) &= ((cond_i \wedge rs_{i+1} = \mathsf{F}(\vec{a}_i))\\
&\quad \vee (rs_{i+1} = rs_i))\\
\Im(\langle \vec{a}_{i+1} := \mathsf{act}_i^1 \curvearrowright \mathsf{act}_i^2,\ cond_i^1 \curvearrowright cond_i^2\rangle) &= \Im(\langle \vec{a}_{i+1} := \mathsf{act}_i^1,\ cond_i^1\rangle)\\
&\quad \vee \Im(\langle \vec{a}_{i+1} := \mathsf{act}_i^2,\ cond_i^2\rangle)\\
\Im(\langle \vec{a}_{i+1} := \texttt{UPDATE}(\vec{e}_i),\ cond_i\rangle) &= (cond_i \wedge \bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j = e_i^j)\\
\Im(\langle \vec{a}_{i+1} := \texttt{INSERT}(\vec{e}_i),\ false\rangle) &= (\bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j = e_i^j)\\
\Im(\langle \vec{a}_{i+1} := \texttt{DELETE}(),\ cond_i\rangle) = \top \quad \Im(\texttt{assert } \phi^{dsa}) &= \phi^{dsa}\\
\Im(\mathit{stmt}_1^{dsa};\ \mathit{stmt}_2^{dsa}) &= \Im(\ \mathit{stmt}_1^{dsa}) \ \wedge\ \Im(\mathit{stmt}_2^{dsa})\\
\Im(\texttt{if } b_i \texttt{ then } \mathit{stmt}_1^{dsa} \texttt{ else } \mathit{stmt}_2^{dsa}) &= (b_i \ \wedge\ \Im(\mathit{stmt}_1^{dsa}) \ \vee (\neg b_i \ \wedge \Im(\mathit{stmt}_2^{dsa}))
\end{aligned}
$$

Fig. 18. Definition of function $\Im(.)$ for language statements.

where $\mathsf{wlp}(\mathit{stmt}^{dsa},\ \psi)$ is the weakest formula (known as *weakest liberal precondition* of stmt^{dsa} w.r.t ψ) that characterizes the pre-states from which every non-blocking execution of stmt^{dsa} either goes wrong or terminates in a state satisfying ψ.

WPC is conservative in the sense that $\mathsf{wp}(\mathcal{P}^{dsa},\ \psi)$ is a predicate over pre-states of $\mathcal{P}^{dsa}$ such that all possible executions of $\mathcal{P}^{dsa}$ terminate in a state satisfying ψ and in all executions the assertion ψ is satisfied. However, in $\mathsf{wlp}(\mathcal{P}^{dsa},\ \psi)$, the execution may go wrong or terminate in a state satisfying ψ. Therefore, the author in Leino and Rustan (2005) derived a notable relation between wp and wlp as follows :

$$\mathsf{wp}(\mathcal{P}^{dsa},\ \psi) \triangleq \mathsf{wlp}(\mathcal{P}^{dsa},\ \psi) \wedge \mathsf{wp}(\mathcal{P}^{dsa},\ \mathit{true})$$

In case of no assert statements in $\mathcal{P}^{dsa}$, we have $\mathsf{wp}(\mathcal{P}^{dsa},\ \psi) = \mathsf{wlp}(\mathcal{P}^{dsa},\ \psi)$, since $\mathsf{wp}(\mathcal{P}^{dsa},\ \mathit{true}) = \mathit{true}$. Therefore, by following the dream property, the wlp of a program $\mathcal{P}^{dsa}$ w.r.t. ψ can be computed as follows:

$$\mathsf{wlp}(\mathcal{P}^{dsa},\ \psi) \triangleq \Im(\mathcal{P}^{dsa}) \Rightarrow \psi$$

The wlp of programs is defined in the same way as of wp (as shown in Fig. 16) except for *assume* and *assert* statements:

$$\mathsf{wlp}(\mathit{assume}\ \phi_1,\ \psi_1) = \phi_1 \Rightarrow \psi_1$$
$$\mathsf{wlp}(\mathit{assert}\ \phi_2,\ \psi_2) = \phi_2 \Rightarrow \psi_2$$

Therefore, for a given database program $\mathcal{P}^{dsa}$; *assert* ψ, where $\mathcal{P}^{dsa}$ does not contain other assert statements, the following function computes a single VC.

$$\mathsf{wp}(\mathcal{P}^{dsa},\ \mathit{assert}\ \psi,\ \mathit{true}) = \mathsf{wlp}(\mathcal{P}^{dsa},\ \psi) = \Im(\mathcal{P}^{dsa}) \Rightarrow \psi$$

Since $\Im(\mathcal{P}^{dsa})$ generates a linear size program formula, unlike the WPC-based VC, the size of the efficient weakest precondition-based VC is linear in the size of the program. The following example illustrates this.

Example 4. Consider annotated DBprog^{dsa} shown in Fig. 11.

$$
\begin{aligned}
\mathit{Let}\ \mathit{stmt}_1^{dsa} = &\ \mathit{assume}\ \mathrm{MP}_0 \geqslant 80000\ \mathit{and}\ \mathrm{EQ}_0 \geqslant 60000\ \mathit{and}\ \mathrm{CT}_0 \geqslant 10000\\
&\ \mathit{and}\ \mathrm{CS}_0 \geqslant 5000;\\
&\ \texttt{SELECT } \mathrm{TA}_0 \texttt{ INTO } z_1 \texttt{ FROM Budget WHERE } \mathrm{Did}_0 = y_0;\\
\mathit{stmt}_2^{dsa} = &\ \texttt{if } z_1 \geqslant x_0 \texttt{ then}\\
&\ m_1 := z_1 - x_0;\\
&\ n_1 := m_1/4;\\
&\ \texttt{UPDATE Budget SET } \mathrm{MP}_1 = \mathrm{MP}_0 - n_1,\ \mathrm{EQ}_1 = \mathrm{EQ}_0 - n_1\\
&\ , \mathrm{CT}_1 = \mathrm{CT}_0 - n_1,\ \mathrm{CS}_1 = \mathrm{CS} - n_1 \texttt{ WHERE } \mathrm{Did}_0 = x_0;\\
&\ \vdots
\end{aligned}
$$

$$n_2 := n_0;$$
$$\text{endif};$$
$$\psi = \textit{assert } \mathrm{MP}_2 \geqslant 80000 \textit{ and } \mathrm{EQ}_2 \geqslant 60000 \textit{ and } \mathrm{CT}_2 \geqslant 10000 \textit{ and } \mathrm{CS}_2 \geqslant 5000;$$

$$\mathrm{wp}(\mathrm{DBprog}^{dsa}, \text{ assert } \psi, \textit{ true}) = \mathrm{wlp}(\mathrm{DBprog}^{dsa}, \psi) = \Im(\mathrm{DBprog}^{dsa}) \Rightarrow \psi$$

$$\Im(\mathrm{DBprog}^{dsa}) \Rightarrow \psi = \Im(stmt_1^{dsa}; \ stmt_2^{dsa}) \Rightarrow \psi$$

$$\Im(stmt_1^{dsa} \ ; \ stmt_2^{dsa}) = \Im(stmt_1^{dsa}) \wedge \Im(stmt_2^{dsa})$$

$$\Im(stmt_1^{dsa}) = (\mathrm{MP}_0 \geqslant 80000 \wedge \mathrm{EQ}_0 \geqslant 60000 \wedge \mathrm{CT}_0 \geqslant 10000 \wedge \mathrm{CS}_0 \geqslant 5000) \wedge (z_1 = \mathrm{TA}_0 \wedge \mathrm{Did}_0 = y_0) \vee (z_1 = z_0 \wedge \neg(\mathrm{Did}_0 = y_0))$$

$$\Im(stmt_2^{dsa}) = (z_1 \geqslant x_0 \wedge \Im(stmt_t^{dsa}) \vee (\neg z_1 \geqslant x_0 \wedge \Im(stmt_f^{dsa})))$$

where $\Im(stmt_t^{dsa}) = (m_1 = z_1 - x_0 \wedge n_1 = m_1/4 \wedge ((\mathrm{Did}_0 = x_0 \wedge \mathrm{MP}_1 = \mathrm{MP}_0 - n_1 \wedge \mathrm{EQ}_1 = \mathrm{EQ}_0 - n_1 \wedge \mathrm{CT}_1 = \mathrm{CT}_0 - n_1 \wedge \mathrm{CS}_1 = \mathrm{CS} - n_1) \vee (\neg(\mathrm{Did}_0 = x_0) \wedge \mathrm{MP}_1 = \mathrm{MP}_0 \wedge \mathrm{EQ}_1 = \mathrm{EQ}_0 \wedge \mathrm{CT}_1 = \mathrm{CT}_0 \wedge \mathrm{CS}_1 = \mathrm{CS}_0)) \wedge m_2 = m_1 \wedge n_2 = n_1 \wedge \mathrm{MP}_2 = \mathrm{MP}_1 \wedge \mathrm{EQ}_2 = \mathrm{EQ}_1 \wedge \mathrm{CT}_2 = \mathrm{CT}_1 \wedge \mathrm{CS}_2 = \mathrm{CS}_1)$

$$\Im(stmt_f^{dsa}) = (\mathrm{MP}_2 = \mathrm{MP}_0 \wedge \mathrm{EQ}_2 = \mathrm{EQ}_0 \wedge \mathrm{CT}_2 = \mathrm{CT}_0 \wedge \mathrm{CS}_2 = \mathrm{CS}_0 \wedge m_2 = m_0 \wedge n_2 = n_0)$$

$$\mathrm{wlp}(\mathrm{DBprog}^{dsa}, \psi) = (\mathrm{MP}_0 \geqslant 80000 \wedge \mathrm{EQ}_0 \geqslant 60000 \wedge \mathrm{CT}_0 \geqslant 10000 \wedge \mathrm{CS}_0 \geqslant 5000) \wedge (z_1 = \mathrm{TA}_0 \wedge \mathrm{Did}_0 = y_0) \vee (z_1 = z_0 \wedge \neg(\mathrm{Did}_0 = y_0)) \wedge ((z_1 \geqslant x_0 \wedge (m_1 = z_1 - x_0 \wedge n_1 = m_1/4 \wedge \mathrm{Did}_0 = x_0 \wedge \mathrm{MP}_1 = \mathrm{MP}_0 - n_1 \wedge \mathrm{EQ}_1 = \mathrm{EQ}_0 - n_1 \wedge \mathrm{CT}_1 = \mathrm{CT}_0 - n_1 \wedge \mathrm{CS}_1 = \mathrm{CS} - n_1) \vee (\neg(\mathrm{Did}_0 = x_0) \wedge \mathrm{MP}_1 = \mathrm{MP}_0 \wedge \mathrm{EQ}_1 = \mathrm{EQ}_0 \wedge \mathrm{CT}_1 = \mathrm{CT}_0 \wedge \mathrm{CS}_1 = \mathrm{CS}_0) \wedge m_2 = m_1 \wedge n_2 = n_1 \wedge \mathrm{MP}_2 = \mathrm{MP}_1 \wedge \mathrm{EQ}_2 = \mathrm{EQ}_1 \wedge \mathrm{CT}_2 = \mathrm{CT}_1 \wedge \mathrm{CS}_2 = \mathrm{CS}_1) \vee ((\neg z_1 \geqslant x_0) \wedge \mathrm{MP}_2 = \mathrm{MP}_0 \wedge \mathrm{EQ}_2 = \mathrm{EQ}_0 \wedge \mathrm{CT}_2 = \mathrm{CT}_0 \wedge \mathrm{CS}_2 = \mathrm{CS}_0 \wedge m_2 = m_0 \wedge n_2 = n_0)) \Rightarrow \mathrm{MP}_2 \geqslant 80000 \wedge \mathrm{EQ}_2 \geqslant 60000 \wedge \mathrm{CT}_2 \geqslant 10000 \wedge \mathrm{CS}_2 \geqslant 5000$$

Observe that the above VC is linear in size and does not have duplicate assert expressions.

3.4. Addressing aggregate functions, NULL values, sub-query, JOIN, UNION, INTERSECT, and MINUS operations

This section provides guidance to deal with crucial database-specific features and operations.

Aggregate functions. The aggregate functions only appear in SELECT statements which usually return a single value as the answer to a posed query. This means that violation is only possible if the values obtained through aggregate functions are stored in a result-set variable which in turn may affect, directly or indirectly, the specification representing a database property.

Given a database program containing aggregate function $h(x)$ on attribute x, our VC-based approaches ensure the presence of $h(x)$ in the resultant VCs. If we treat $h(x)$ in the VCs as a new variable and if the VC violates the specification, let m be the value of $h(x)$ for which a violation is observed. Since our goal is to identify one instance which leads to some properties violation, we adopt the following approach to deal with various aggregate functions:

(a) $h(x) \triangleq \min(x)$.
There exists a minimum value m of attribute x in a database instance which leads to this violation. However, this may be a false positive because of the treatment of x and $\min(x)$ as different variables. As a solution, we propose to adopt all the constraints over x as the constraints over $\min(x)$ in the VCs.

(b) $h(x) \triangleq \max(x)$.
Same as $\min(x)$.

(c) $h(x) \triangleq \mathrm{sum}(x)$.
In this case, we can adopt only the constraints which define the lowest bound of x-values as the constraints over the variable $\mathrm{sum}(x)$. Observe that this over-approximation may lead to false positives.

(d) $h(x) \triangleq \mathrm{avg}(x)$.
In this case, we can adopt only the constraints which define both the lower- and upper-bound of x-values as the constraints over the variable $\mathrm{avg}(x)$.

(e) $h(x) \triangleq \mathrm{count}(*)$.
This is applicable only when we consider table-level properties as a part of the specification. An example of a table-level property is "*each department must have at least two employees*". As we consider only tuple-level properties, this case is out of the scope of this work, and we consider it as our future plan.

Treating NULL values. In order to address NULL values, we provide a separate treatment for the properties which specify whether attribute values may accept NULL or must be NOT NULL. To be specific, NULL property-violation may take place in the following situations:

(a) The value of an expression e is set to an attribute "a" in INSERT or UPDATE, where "a" is NOT NULL and the value of e may be NULL.
(b) The presence of a condition of the form "x is NULL/x is NOT NULL" in conditional statement.

In case (a), special care should be taken by checking the possibility of NULL values occurring in e against the NULL/NOT NULL property of the attribute "a". In such a case, a warning report is generated. In case (b), the presence of a condition of the form of "x IS NULL" or "x IS NOT NULL" always leads to "always false" or "always true" respectively w.r.t the given NULL constraints "x IS NOT NULL". Therefore, the VC generation can decide statically whether to ignore or to include the logical encoding of the other part of the database statement.

Sub-query. A sub-query can be nested inside the WHERE or HAVING clause of an outer SELECT, INSERT, UPDATE, or DELETE statement, or inside another sub-query. A sub-query can appear anywhere an expression can be used, if it returns a single value. SQL statements that include a sub-query take one of the following format:

... WHERE ⟨e⟩⟨*comparison_operator*[ANY|ALL]⟩⟨(sub-query)⟩.
... WHERE ⟨e [NOT] IN⟩⟨(sub-query)⟩.
... WHERE ⟨[NOT] EXIST⟩⟨(sub-query)⟩

In the case of a sub-query of the format "... WHERE ⟨e⟩ ⟨*comparison_operator*[ANY|ALL]⟩ ⟨(sub-query)⟩", we first convert the inner query into a logical sub-formula which becomes a part of the final VC involving attributes, operators and co-relations present in the outer query. Example 5 illustrates this scenario.

Example 5. Consider the UPDATE statement $Q_{upd} \triangleq$ UPDATE ROUTES SET P = P ∗((100 − d)/100) WHERE R_ID ⩾ (SELECT L_ID FROM LOADS WHERE CID = z);. The logical sub-formula generated from the sub-query using symbolic execution is $\mathrm{R\_ID}_0 \geqslant \mathrm{L\_ID}_0 \wedge \mathrm{CID}_0 = z_0$. Therefore, the resultant VC would be:

$$((\mathrm{P}_1 = \mathrm{P}_0 * ((100 - d_0)/100) \wedge \mathrm{R\_ID}_0 \geqslant \mathrm{L\_ID}_0 \wedge \mathrm{CID}_0 = z_0) \vee (\mathrm{P}_1 = \mathrm{P}_0 \wedge \neg(\mathrm{R\_ID}_0 \geqslant \mathrm{L\_ID}_0 \wedge \mathrm{CID}_0 = z_0)))$$

In the case of a nested query of the format "... WHERE ⟨e [NOT] IN⟩ ⟨(sub-query)⟩", we replace the IN operator by the assignment operator (:=) during VC generation. Since our main aim is to determine property violations by the database code, any instance invalidating the resultant VC fulfills our objective.

JOIN. Without loss of generality, let us assume that all attribute names in the database are unique. The presence of a query containing θ-JOIN is addressed easily by incorporating the condition θ in the VC, in addition. Observe that equi-JOIN and natural-JOIN are special cases of θ-JOIN.

$$\{\psi\}\ \text{skip}\ \{\psi\} \qquad \text{(Skip)}$$

$$\{\psi[e/v_a]\}\ v_a\ :=\ e\ \{\psi\} \qquad \text{(Assignment)}$$

$$\frac{\{(\psi[\vec{e}/\vec{a}] \wedge cond) \vee (\psi \wedge \neg cond)\}\langle \vec{a} := \texttt{UPDATE}(\vec{e}),\ cond\rangle\{\psi\}}{} \qquad \text{(UPDATE)}$$

$$\{\psi[\vec{e}/\vec{a}] \vee \psi\}\langle \vec{a} := \texttt{INSERT}(\vec{e}),\ false\rangle\{\psi\} \qquad \text{(INSERT)}$$

$$\{\psi \wedge \neg cond\}\langle \vec{a} := \texttt{DELETE}(),\ cond\rangle\{\psi\} \qquad \text{(DELETE)}$$

$$\frac{\{(\psi[\mathrm{F}(\vec{a})/rs] \wedge cond)) \vee (\psi \wedge \neg cond)\}}{\langle rs\ :=\ \texttt{SELECT}(f(\vec{e}), r(\vec{h}(\vec{x})), \phi, g(\vec{e})),\ cond\rangle\{\psi\}} \qquad \text{(SELECT)}$$

$$\frac{\{\phi\}stmt_1\{\phi'\} \qquad \{\phi'\}stmt_2\{\psi\}}{\{\phi\}stmt_1; stmt_2\{\psi\}} \qquad \text{(Sequence)}$$

$$\frac{\{\phi \wedge b\}stmt_1\{\psi\} \qquad \{\phi \wedge \neg b\}stmt_2\{\psi\}}{\{\phi\}\text{if } b \text{ then } stmt_1 \text{ else } stmt_2\{\psi\}} \qquad \text{(Conditional)}$$

$$\frac{\{\phi \wedge b\}stmt\{\phi\}}{\{\phi\}\text{while } b \text{ do } stmt\{\phi \wedge \neg b\}} \qquad \text{(While loop)}$$

Fig. 19. H_{db}: Hoare rules for database programs.

UNION, INTERSECT and MINUS operations. Two or more queries can be combined using set operators UNION, INTERSECT and MINUS. The logical encoding of UNION (or INTERSECT) is achieved by using logical OR $\vee$ (or logical AND $\wedge$) operator. That is, logical formulas obtained from operands of UNION (or INTERSECT) are composed using $\vee$ (or $\wedge$). Since MINUS operation can be replaced by UNION and INTERSECT operations, VC generation in the presence of MINUS can be achieved by using $\vee$ and $\wedge$.

3.5. Treating loops

As mentioned earlier, deductive verification approaches require user's guidance and expertise for program annotation. Naturally, the presence of loops in a program makes this process more challenging. The fundamental step in such a case is to infer a loop invariant which remains true throughout the loop iterations.

The following are examples of situations where iteration is required for effective coding for database programs:

- If the action over the tuples of a table is parameterized with changeable parameters for different tuples. In this case, rather than writing separate database statements in the code for different tuples of the table, the parameterized action can collectively be expressed in terms of a loop.
- To iterate over the result-set values using a cursor.
- Recursive Queries.

Hoare logic was proposed to deal with iterating While-programs, based on the loop invariants. Formally, the following classic inference rule (Winskel, 1993) uses the invariant ϕ to express the partial correctness of a loop:

$$\frac{\vdash \{\phi \wedge cond\}stmt\{\phi\}}{\vdash \{\phi\}\text{while } cond \text{ do } stmt\ \{\phi \wedge \neg cond\}}$$

Fig. 19 defines the Hoare logic H_{db} for database programs by extending the same for imperative programs.

The relation between Hoare triples and weakest precondition is that $\{\phi\}\mathcal{P}\{\psi\}$ iff $\phi \implies \mathsf{wp}(\mathcal{P}, \psi)$ or $\mathsf{sp}(\mathcal{P}, \phi) \implies \psi$. Observe that, since wp generates quantifier free formulas, it is often used in Hoare logic-based verifiers, instead of sp. The computation of loop invariants in host imperative languages relies on existing approaches (Winskel, 1993), that can be adopted for database programs as well. In particular, the computation of inductive invariants in these cases, by following the approaches in Furia et al. (2014) and Kroening and Weissenbacher (2010), can be used with our proposed verification approaches. On the other hand, as an alternative solution, we can also extend the existing works on abstract interpretation of database programs (Jana et al., 2018b). The definition of widening operation, in addition, covers recursive queries as well (Cortesi and Halder, 2013).

4. Complexity analysis and correctness proofs

We are now in a position to perform a complexity analysis of the proposed VC generation algorithms and to prove their correctness.

4.1. Complexity analysis

Let us describe the asymptotic characteristics of VCs generation by various approaches. VC generation based on symbolic execution generates VCs along all execution paths of a program. For a given program with n conditional statements, there exist 2^n execution paths, and therefore this results in a number of VCs in $O(2^n)$. In contrary, the CNF-based approach sequentializes branches of conditional statements which increases the program's length to some extent. Assuming that the program contains a chain of nested conditions up to depth m, the approach generates a single VC of size $O(n+m^2)$. The weakest precondition-based approach considers both branches into a single formula. Therefore, a program with n conditional statements generates a single VC of size $O(2^n)$.

4.2. Correctness proofs

We have defined a number of functions, namely Path, VC^{se}, VC^{cnf} and wp, which act as the core of different VC generation techniques in Section 3. Therefore, the correctness of the proposed verification techniques can be guaranteed by proving the correctness of these functions. We achieve this by considering the validity of input and output logical formulas in terms of their semantics w.r.t. states and transition semantics of database applications.

To this aim, let us first recall from Jana et al. (2018b) the notions of states and state transition semantics of database programs.

State Transition Semantics of Database Language. Since our database language involves both host imperative variables and database attributes, the state is defined by considering the semantics domains for both of them.

Definition 1 (*Application Environment*)**.** Given the set of application variables $\mathbb{V}_a$ and the domain of values Val, let $\mathfrak{E}_a \triangleq [\mathbb{V}_a \mapsto \texttt{Val}]$ be the set of all functions with domain $\mathbb{V}_a$ and range included in Val. An application environment $\rho_a \in \mathfrak{E}_a$ maps application variables to the domain of values Val.

Definition 2 (*Database Environment*)**.** A database d 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 $i \in I_x$, $\rho_d(i) = t_i$.

Definition 3 (*Table Environment*)**.** Consider a database table t with attribute $\texttt{attr}(t) = \{a_1, a_2, \ldots, a_k\}$. So, $t \subseteq D_1 \times D_2 \times \cdots \times D_k$ where a_i is the attribute corresponding to the typed domain D_i. A *table environment* ρ_t for t is defined as a function such that for any attribute $a_i \in \texttt{attr}(t)$, $\rho_t(a_i) = \langle \pi_i(l_j) \mid l_j \in t\rangle$ where π is the projection operator and $\pi_i(l_j)$ represents the ith element of the l_jth row. In other words, ρ_t maps a_i to the ordered set of values over the rows of the table t.

Definition 4 (*States and Concrete Semantics*)**.** Let Σ_{dba} be the set of states for the database language under consideration, defined by $\Sigma_{dba} \triangleq \mathfrak{E}_{dbs} \times \mathfrak{E}_{aps}$ where $\mathfrak{E}_{dbs}$ and $\mathfrak{E}_{aps}$ denote the set of all database environments and the set of all application environments respectively. Therefore, a state $\rho \in \Sigma_{dba}$ is denoted by a tuple (ρ_d, ρ_a) where $\rho_d \in \mathfrak{E}_{dbs}$ and $\rho_a \in \mathfrak{E}_{aps}$. The transition relation

$$\mathscr{T}_{dba} \in \left[(\mathbb{C} \times \Sigma_{dba}) \mapsto \wp(\Sigma_{dba})\right] \tag{2}$$

specifies which successor states $(\rho_{d'}, \rho_{a'}) \in \Sigma_{dba}$ can follow when a statement $c \in \mathbb{C}$ executes on state $(\rho_d, \rho_a) \in \Sigma_{dba}$. Therefore, the transitional semantics $\mathscr{T}_{dba}[\![\mathcal{P}]\!] \in \big[\ (\mathcal{P} \times \Sigma_{dba}\ [\![\mathcal{P}]\!]) \mapsto \wp(\Sigma_{dba}[\![\mathcal{P}]\!])\ \big]$ of a program $\mathcal{P}$ restricts the transition relation to program instructions only, i.e.

$$\mathscr{T}_{dba}[\![\mathcal{P}]\!](\rho_d, \rho_a) = \Big\{\ (\rho_{d'}, \rho_{a'}) \mid (\rho_d, \rho_a), (\rho_{d'}, \rho_{a'}) \in \Sigma_{dba}[\![\mathcal{P}]\!] \\ \wedge\, c \in \mathcal{P} \wedge (\rho_{d'}, \rho_{a'}) \in \mathscr{T}_{dba}[\![c]\!](\rho_d, \rho_a)\ \Big\}$$

Example 6 illustrates the concrete semantics of an update statement.

Example 6. Consider the database table t in Table 3(a) and the following update statement:

Q_{upd} : UPDATE t SET $sal := sal + 100$ WHERE $age \geqslant 35$

The abstract syntax is denoted by $\langle \texttt{UPDATE}(\vec{v}_d, \vec{e}),\ \phi\rangle$, where $\phi = (age \geqslant 35)$ and $\vec{v}_d = \langle sal\rangle$ and $\vec{e} = \langle sal + 100\rangle$.

The table targeted by Q_{upd} is $target(Q_{upd}) = \{t\}$. The semantics of Q_{upd} is:

$$\begin{aligned}
&\mathscr{T}_{dba}[\![Q_{upd}]\!](\rho_d, \rho_a)\\
&= \mathscr{T}_{dba}[\![\big(\texttt{UPDATE}(\langle sal\rangle, \langle sal + 100\rangle),\ (age \geqslant 35)\big)]\!](\rho_d, \rho_a)\\
&= \mathscr{T}_{dba}[\![\big(\texttt{UPDATE}(\langle sal\rangle, \langle sal + 100\rangle),\ (age \geqslant 35)\big)]\!](\rho_t, \rho_a)\\
&\qquad [\textbf{Since,}\ target(Q_{upd}) = \{t\}]\\
&= \mathscr{T}_{dba}[\![\texttt{UPDATE}(\langle sal\rangle, \langle sal + 100\rangle)]\!](\rho_{t\downarrow(age\geqslant 35)}, \rho_a) \sqcup\\
&\quad (\rho_{t\downarrow\neg(age\geqslant 35)}, \rho_a) \quad [\textbf{Absorbing}\ \phi = (age \geqslant 35)]\\
&= (\rho_{t'}, \rho_a) \sqcup (\rho_{t\downarrow\neg(age\geqslant 35)}, \rho_a) = (\rho_{t'} \sqcup \rho_{t\downarrow\neg(age\geqslant 35)},\ \rho_a \sqcup \rho_a)\\
&= (\rho_{t''}, \rho_a)\\
&\quad \text{where}\ \rho_{t'} \equiv \rho_{t\downarrow(age\geqslant 35)}\big[sal \leftarrow E[\![sal + 100]\!](\rho_{t\downarrow(age\geqslant 35)}, \rho_a)\big]\\
&\qquad\quad = \rho_{t\downarrow(age\geqslant 35)}[sal \leftarrow \langle 1600, 2600, 3100\rangle]
\end{aligned}$$

The notation $(t \downarrow (age \geqslant 35))$ denotes the set of tuples in t for which $(age \geqslant 35)$ is true (denoted by the red part in t of Table 3(a)). $E[\![.]\!]$ is a semantic function for arithmetic expressions which maps "$sal + 100$" to a list of values $\langle 1600, 2600, 3100\rangle$ on the table environment $\rho_{t\downarrow(age\geqslant 35)}$. The notation $\leftarrow$ denotes a substitution by new values. Observe that the substitution of '*sal*' by the list of values in $\rho_{t\downarrow(age\geqslant 35)}$ results in a new table environment $\rho_{t'}$ (denoted by the red part in Table 3(b)). Finally, the least upper bound (denoted $\sqcup$) of the two states results in a new state $(\rho_{t''}, \rho_a)$ where t'' is depicted in Table 3(b).

Lemmas and Theorems. Let us now state a number of lemmas and theorems aiming to guarantee the correctness of various VC generation functions in our verification approaches. As mentioned earlier, we make use of state and transition semantics defined above to prove these lemmas and theorems.

Extending the semantics of the assertion language defined in Winskel (1993) to the case of database applications, let the relation $(\rho_d, \rho_a) \models_I \Phi$ mean that the state (ρ_d, ρ_a) satisfies the assertion Φ under the interpretation I.

Lemma 1. *Let Φ be the logical encoding of an execution path π up to program point ℓ. Let $stmt^{dsa}$ be a DSA form of program statement at program point $\ell+1$. Let Ψ be the logical encoding computed using the function* Path$(\Phi, stmt^{dsa})$*. The function* Path *is correct if* $\forall\ (\rho_d, \rho_a) :\ (\rho_d, \rho_a) \models_I \Phi$ *under interpretation I and* $\mathscr{T}_{dba}[\![stmt^{dsa}]\!](\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$*, implies* $(\rho_{d'}, \rho_{a'}) \models_I \Psi$.

Proof. The proof is established based on structural induction. Assume that (ρ_d, ρ_a) satisfies Φ under the interpretation I, i.e. $(\rho_d, \rho_a) \models_I \Phi$.

Assignment statement $v_{i+1} := e_i$. Let $\mathscr{T}_{dba}[\![v_{i+1} := e_i]\!](\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$, where $(\rho_{a'}, \rho_{a'})$ is obtained by substituting all occurrences of v_{i+1} in (ρ_d, ρ_a) by $\mathscr{T}_{dba}\ [\![e_i]\!](\rho_d, \rho_a)$. Since e_i does not involve v_{i+1}, its semantics are the same w.r.t. both the states (ρ_d, ρ_a) and $(\rho_{d'}, \rho_{a'})$. Therefore,

$$(\rho_{d'}, \rho_{a'}) \models_I (v_{i+1} == e_i) \tag{3}$$

On the other hand, since v_{i+1} is not present in Φ due to the DSA property and only v_{i+1} is affected in $(\rho_{d'}, \rho_{a'})$, we have

$$(\rho_{d'}, \rho_{a'}) \models_I \Phi \tag{4}$$

Combining Eqs. (3) and (4), we get $(\rho_{d'}, \rho_{a'}) \models_I (\Phi \wedge v_{i+1} == e_i)$.

Database statements Q. Since a database statement Q involves action 'A' and condition 'cond', this can be treated as a guarded command equivalent to "if *cond* then A". Consider the DSA form $\langle \vec{a}_{i+1} := \text{act}_i^1 \curvearrowright \text{act}_i^2,\ cond_i^1 \curvearrowright cond_i^2\rangle$ of different database statements. By the definition of the function Path, we have:

$$\begin{aligned}
\text{Path}(\Phi,\ \langle \vec{a}_{i+1} :=\ &\text{act}_i^1 \curvearrowright \text{act}_i^2,\ cond_i^1 \curvearrowright cond_i^2\rangle) =\\
&\text{Path}(\Phi, \langle \vec{a}_{i+1} := \text{act}_i^1,\ cond_i^1\rangle) \vee\\
&\text{Path}(\Phi, \langle \vec{a}_{i+1} := \text{act}_i^2,\ cond_i^2\rangle)
\end{aligned}$$

Let us now prove the lemma for each database action:

- $Q_{upd}^{dsa} \triangleq \langle \vec{a}_{i+1} := \texttt{UPDATE}(\vec{a}_i) \curvearrowright \texttt{UPDATE}(\vec{e}_i), \neg cond_i \curvearrowright cond_i\rangle$. Given a state (ρ_d, ρ_a), assume that $\mathscr{T}_{dba}[\![Q_{upd}^{dsa}]\!]\ (\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$. By the definition of the function Path, we have

$$\Psi = \text{Path}(\Phi,\ Q_{upd}^{dsa}) = \Phi \wedge ((\neg cond_i \wedge \bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j == a_i^j) \vee (cond_i \wedge \bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j == e_i^j))$$

  Since $(\rho_d, \rho_a) \models_I \Phi$ and Φ does not involve $\vec{a}_{i+1}$ due to the DSA property, the following holds.

$$(\rho_{d'}, \rho_{a'}) \models_I \Phi \tag{5}$$

  Now we have two possibilities: either $(\rho_{d'}, \rho_{a'}) \models_I \neg cond_i$ or $(\rho_{d'}, \rho_{a'}) \models_I cond_i$. In the former case, the action $\vec{a}_{i+1} :=$ $\texttt{UPDATE}(\vec{a}_i)$ results in the same semantics for $\vec{a}_i$ and $\vec{a}_{i+1}$ w.r.t. $(\rho_{d'}, \rho_{a'})$, i.e. $\mathscr{T}_{dba}[\![\vec{a}_i]\!]\ (\rho_{d'}, \rho_{a'}) = \mathscr{T}_{dba}[\![\vec{a}_{i+1}]\!](\rho_{d'}, \rho_{a'})$. Therefore,

$$(\rho_{d'}, \rho_{a'}) \models_I (\neg cond_i \wedge \bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j == a_i^j) \tag{6}$$

  In the latter case, the attributes $a_{i+1}^j \in \vec{a}_{i+1}$ in $(\rho_{d'}, \rho_{a'})$, where $j = 1 \ldots |\vec{a}_{i+1}|$, are substituted by $\mathscr{T}_{dba}[\![e_i^j]\!]\ (\rho_d, \rho_a)$ respectively. Since only the value of $\vec{a}_{i+1}$ is affected in $(\rho_{d'}, \rho_{a'})$, we have $\mathscr{T}_{dba}[\![\vec{a}_{i+1}]\!](\rho_{d'},\ \rho_{a'}) = \mathscr{T}_{dba}[\![\vec{e}_i]\!](\rho_{d'},\ \rho_{a'})$, and therefore,

$$(\rho_{d'}, \rho_{a'}) \models_I (cond_i \wedge \bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j == e_i^j) \tag{7}$$

  Hence, combining Eqs. (5)–(7), $(\rho_{d'},\ \rho_{a'}) \models_I \Psi$ is proved.

Table 3
Database before and after the update operation.

(a) table t

eid	*sal*	*age*	*dno*
1	1500	35	10
2	800	28	20
3	2500	50	10
4	3000	62	10

(b) table t''

eid	*sal*	*age*	*dno*
1	1600	35	10
2	800	28	20
3	2600	50	10
4	3100	62	10

Proofs for other statements follow a similar direction. The reader is directed to Appendix A.1 for details. □

Theorem 1. *Given a database program $\mathcal{P}$ and its annotated DSA form $\{assume\ \phi_1^{dsa};\ \mathcal{P}^{dsa};\ assert\ \phi_2^{dsa};\}$, let X be the set of verification conditions derived by the function VC^{se} which includes initial condition ϕ_1^{dsa}. If for all $\omega_i \in X$, ω_i is valid (denoted by $\vdash \omega_i$), then $\mathcal{P}$ satisfies the property ϕ_2^{dsa} and vice versa.*

Proof. On applying $\mathsf{VC}^{se}(\emptyset,\ \{assume\ \phi_1^{dsa};\ \mathcal{P}^{dsa};\ assert\ \phi_2^{dsa}\ ;\})$, according to the definition in Fig. 10, we get a set of VCs X along all execution paths, each of the form $\omega_i \triangleq (\Phi_i \implies \phi_2^{dsa}) \in X$. For all initial states $(\rho_d,\ \rho_a)$ satisfying ϕ_1^{dsa}, i.e. $(\rho_d,\ \rho_a) \models_I \phi_1^{dsa}$, if $\mathcal{T}_{dba}[\![\{assume\ \phi_1^{dsa};\ \mathcal{P}^{dsa}\}]\!](\rho_d,\ \rho_a) = (\rho_{d'},\ \rho_{a'})$, then according to Lemma 1 we have

$$for\ all\ \omega_i :\ (\rho_{d'},\ \rho_{a'}) \models_I \Phi_i \tag{8}$$

If the program satisfies the assert ϕ_2^{dsa}, this means $(\rho_{d'},\ \rho_{a'}) \models_I \phi_2^{dsa}$. Therefore, all $\omega_i \triangleq (\Phi_i \implies \phi_2^{dsa})$ are valid. This proves the theorem. □

Theorem 2. *Let ω be a verification condition generated from an annotated database program $\{assume\ \phi_1^{dsa};\ \mathcal{P}^{cnf};\ assert\ \phi_2^{dsa};\}$ in its CNF form, by applying the function VC^{cnf} which includes initial condition ϕ_1^{dsa}. If ω is valid (denoted as $\vdash \omega$), then $\mathcal{P}$ satisfies the property ϕ_2^{dsa} and vice versa.*

Proof. Since $\mathcal{P}^{cnf}$ is semantically equivalent to $\mathcal{P}^{dsa}$, the proof is similar to Theorem 1. □

Theorem 3. *Given an annotated program $\{assume\ \phi_1;\ \mathcal{P};\ assert\ \phi_2;\}$. If the verification condition $\phi_1 \implies wp(\mathcal{P}, \phi_2)$ is valid, then $\mathcal{P}$ satisfies the property ϕ_2 and vice versa.*

Proof. The proof is based on structural induction on $stmt \in \mathcal{P}$. Recall the definition of wp in Fig. 16 and let $\mathsf{wp}(stmt,\ \psi_2) = \psi_1$. We have to prove that, if $(\rho_d,\ \rho_a) \models_I \phi_1$ and $\phi_1 \rightarrow \psi_1$ and $\psi_2 \rightarrow \phi_2$ and $\mathcal{T}_{dba}[\![stmt]\!](\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$, then $(\rho_{d'}, \rho_{a'}) \models_I \psi_2$. Let us now consider the different cases:

Case $\mathcal{P} \triangleq v := e$. Let $(\rho_d,\ \rho_a) \models_I \phi_1$. According to the definition, $\mathsf{wp}(v := e,\ \psi_2) = \psi_2[e/v]$. Assume that $\phi_1 \rightarrow \psi_2[e/v]$. Therefore,

$$(\rho_d,\ \rho_a) \models_I \psi_2[e/v] \tag{9}$$

According to the transition semantics, assume $\mathcal{T}_{dba}\ [\![v := e]\!](\rho_d, \rho_a) = (\rho_{d'},\ \rho_{a'})$ such that

$$\rho_a(z) = \begin{cases} m,\ if\ z = v \\ \rho_a(z),\ Otherwise \end{cases}$$

where $m = \mathcal{T}_{dba}[\![e]\!](\rho_d, \rho_a)$. Since, in $(\rho_{d'},\ \rho_{a'})$ all occurrences of x are replaced by m, according to Eq. (9), we have $(\rho_{d'},\ \rho_{a'}) \models_I \Psi_2$. Assuming $\psi_2 \rightarrow \phi_2$, this case is proved.

Case $\mathcal{P} \triangleq \langle \vec{a} := \mathtt{UPDATE}(\vec{e}), cond\rangle$. Let $(\rho_d, \rho_a) \models_I \phi_1$. According to the definition, $\mathsf{wp}(\langle \vec{a} := \mathtt{UPDATE}(\vec{e}), cond\rangle,\ \psi_2) = ((\psi_2 \wedge \neg cond) \vee (\psi_2[\vec{e}/\vec{a}] \wedge cond))$. Assume that $\phi_1 \rightarrow ((\psi_2 \wedge \neg cond) \vee (\psi_2[\vec{e}/\vec{a}] \wedge cond))$. Therefore,

$$(\rho_d,\ \rho_a) \models_I ((\psi_2 \wedge \neg cond) \vee (\psi_2[\vec{e}/\vec{a}] \wedge cond)) \tag{10}$$

According to the semantics function, let us assume $\mathcal{T}_{dba}[\![Q_{upd}]\!](\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$, where

(i) Values of $\vec{a}$ will be updated by $\mathcal{T}_{dba}[\![\vec{e}]\!](\rho_d, \rho_a)$ for the tuples satisfying '*cond*'. That is, similarly to the assignment statement, we have

$$(\rho_{d'}, \rho_{a'}) \models_I \Psi_2\ when\ (\rho_d, \rho_a) \models_I \Psi_2[\vec{e}/\vec{a}] \wedge cond \tag{11}$$

(ii) Values of $\vec{a}$ will remain unchanged for those tuples which do not satisfy '*cond*'. That is,

$$(\rho_{d'},\ \rho_{a'}) \models_I \Psi_2\ when\ (\rho_d,\ \rho_a) \models_I \Psi_2 \wedge \neg cond \tag{12}$$

Combining Eqs. (10)–(12), the lemma is proved for Q_{upd} assuming $\Psi_2 \rightarrow \phi_2$.

Proofs for other statements are similar. The reader is referred to Appendix A.1 for details. □

Recall the Hoare triple in Section 3.3 and the Hoare logic system H_{db} for the database language defined in Fig. 19. The semantics of Hoare triples is given below:

Definition 5. The Hoare triple $\{\phi\}stmt\{\psi\}$ is said to be valid, denoted as $\models \{\phi\}stmt\{\psi\}$, whenever for all $((\rho_d,\ \rho_a), (\rho_{d'},\ \rho_{a'})) \in \Sigma$, if $(\rho_d,\ \rho_a) \models_I \phi$ and $\mathcal{T}_{dba}[\![stmt]\!](\rho_d,\ \rho_a) = (\rho_{d'},\ \rho_{a'})$, then $(\rho_{d'},\ \rho_{a'}) \models_I \psi$.

Let us now prove the soundness of H_{db}. We denote by $\vdash_{H_{db}} \{\phi\}\mathcal{P}\{\psi\}$ the fact that the Hoare triple $\{\phi\}\mathcal{P}\{\psi\}$ is derivable in the Hoare logic system H_{db}.

Theorem 4 (*Soundness of Hoare Logic System*)**.** *Let $stmt \in \mathcal{P}$ and let ϕ, ψ be the pre-condition and post-condition respectively. If $\vdash_{H_{db}} \{\phi\}stmt\{\psi\}$, then $\models \{\phi\}stmt\{\psi\}$.*

Proof. The proofs for database statements follow the proof of Theorem 3. For assignment, sequence, conditional, and iteration statements, the reader is referred to Winskel (1993). □

5. DBverify: A database code verifier

In this section, we present DBverify, a prototype implementation of our proposed verification approaches for PL/SQL language, which is developed in Python with roughly 6500 lines of codes. The workflow of DBverify is shown in Fig. 20. DBverify consists of four modules: (1) `DSA-translator`, (2) `SE-verify`, (3) `CNF-verify`, and (4) `WP-verify`. The overall schematic diagram of DBverify is depicted in Fig. 21. Observe that the first module converts a given PL/SQL code into its equivalent DSA form. The modules `SE-verify`, `CNF-verify`, and `WP-verify` implement verification condition generation under three deductive-based approaches. Let us now describe each of the modules in detail.

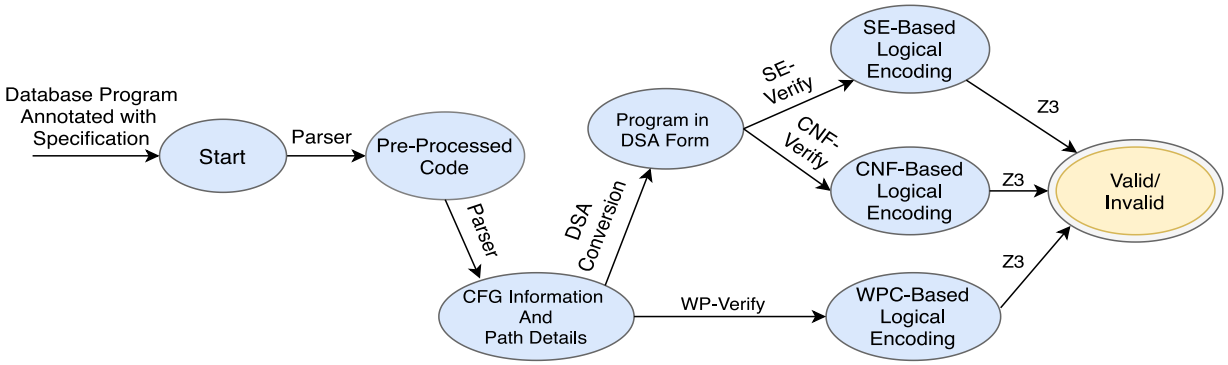


Fig. 20. DBverify workflow.

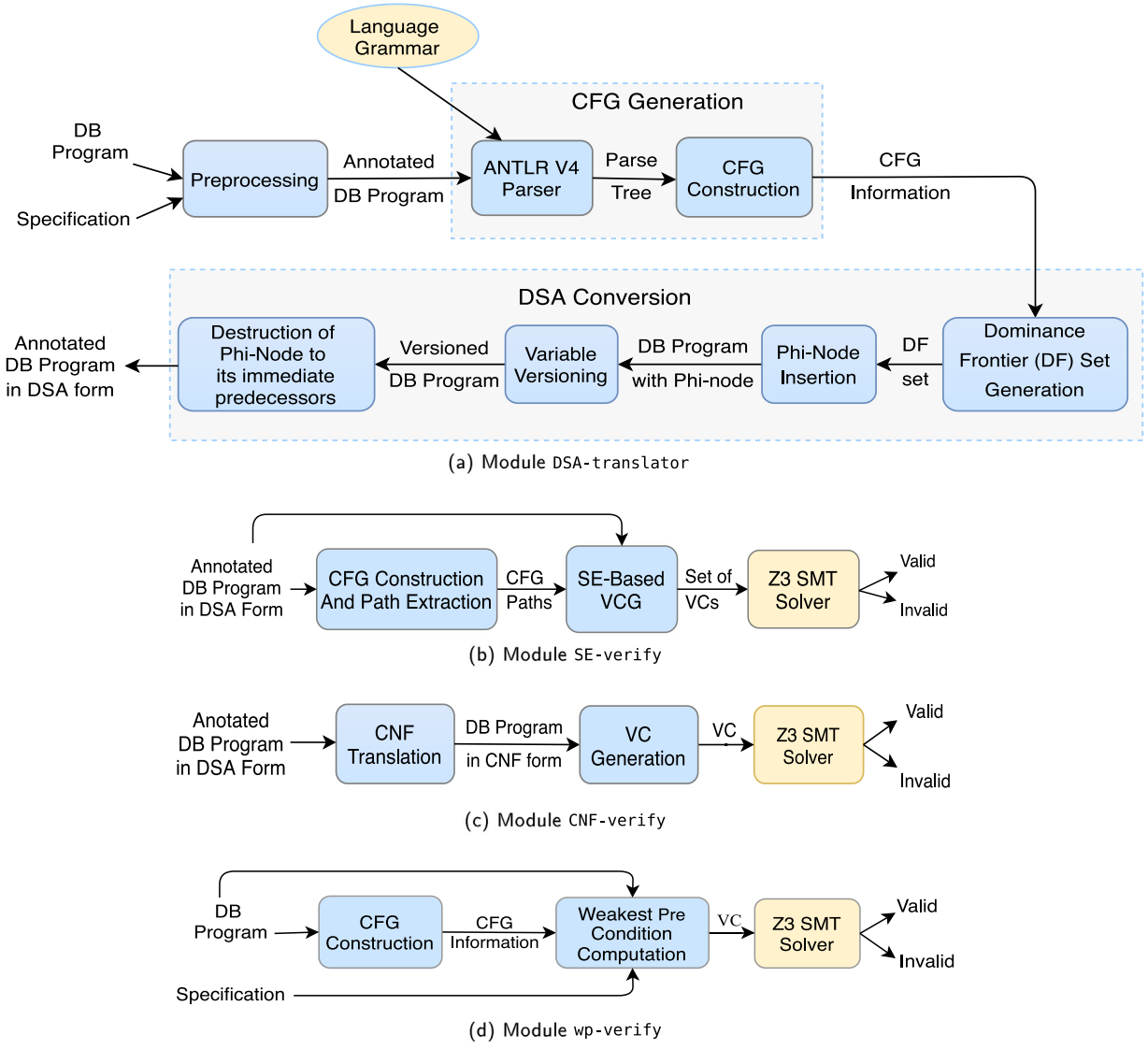


Fig. 21. Schematic diagram of DBverify.

(1) Module `DSA-translator`: This module converts a given PL/SQL code into its equivalent DSA form. The initial tasks of the module are to annotate PL/SQL code by *assume* and *assert* statements taking the given specification into consideration, and then to construct the Control Flow Graph (CFG) of the annotated code using parsing techniques. We have used the ANTLR4 parser (Parr, 2013) for this purpose. Finally, taking CFG information as input, the module performs variable versioning and destruction of ϕ-nodes into its immediate predecessor-blocks to generate DSA form by following the standard algorithm (Briggs et al., 1998) with an extension to cover the PL/SQL language. The schematic diagram of this module is depicted in Fig. 21(a).

Table 4
Description of benchmark database applications.

DB applications	Description	LOC	#Procedures	#Attributes	#Application variables
Courier_Company (CC) (PL/SQL Project, 2020a)	Application to manages courier related information	935	16	40	122
Inventory_Management (IM) (PL/SQL Project, 2020b)	Maintain auto-part and assembling information	550	05	37	73
CableCity(CB) (PL/SQL Project, 2020c)	Manages database of stocks, sales and customer information	890	09	25	54
Hotel_Reservation (HR) (PL/SQL Project, 2020d)	Maintains details of guest, rooms, reservation, etc	504	05	19	24
Retail_Business Management (RM) (PL/SQL Project, 2020e)	Web application to keep records of sales of a retailer	552	04	35	24
Computer Store Management (CS) (PL/SQL Project, 2020f)	Application to manage the quantity, price, product details, etc. information of computer hardware store	738	05	38	34
Banking Management (BS) (PL/SQL Project, 2020g)	Application that manages information about accounts, debit, credit transaction details, etc. of customers.	511	04	23	34
Course Registration (CR) (PL/SQL Project, 2020h)	Application to maintain the course registration details of students in university	268	01	31	15

(2) Module `SE-verify`: Given an annotated PL/SQL program in its DSA form, this module generates a set of VCs according to Algorithm 1 depicted in Section 3.1. The resultant VCs are further passed to the SMT Solver for validity checking. We have used Z3 for this purpose. The schematic diagram is depicted in Fig. 21(b).
(3) Module `CNF-verify`: Unlike `SE-verifier`, this module first converts the annotated input program (in DSA form) into its equivalent CNF representation and then generates a single VC according to Algorithm 2 given in Section 3.2. Finally, this VC is fed to the Z3 SMT Solver for validity checking. The schematic diagram of this module is depicted in Fig. 21(c).
(4) Module `WP-verify`: For a given database program and its specification, this module computes a single VC according to Algorithm 3 in Section 3.3. The schematic diagram of this module is depicted in Fig. 21(d).

As suggested in Section 3.4, DBverify considers the presence of aggregate operations as new variables. To deal with NULL, we maintain a dictionary of the form {tableName : [(attr-1, nullity), (attr-2, nullity), . . . , (attr-n, nullity)]}, where nullity represents attribute constraints such as NULL or NOT NULL. On assigning an expression containing NULL to any of these attributes, we check and report (if any) constraint violations. When a NULLvalue appears in the guard of any conditional statements (e.g. `if` ... IS NULL {...}), we replace it by $*$ representing non-determinism. Therefore, the VC generation can decide statically whether to ignore or to include the logical encoding of the other part of the database statement. In the case of nested queries, DBverify proceeds from the inner-most sub-query to the outer-most sub-query based on the generated Abstract Syntax Trees (AST) of sub-queries to generate logical formulas, according to the formalism described in Section 3.4. In the case of θ-JOIN operation, DBverify first extracts the logical formula from the AST of the condition θ, and then adds it with the logical formula of the other part of the query. In the case of outer-JOIN, UNION, INTERSECT, and MINUS operations, we follow the formalism described in Section 3.4.

6. Experimental evaluation

This section presents experimental results on a set of PL/SQL benchmark codes (PL/SQL Project, 2020a,b,c,d,e,f,g,h) using our prototype tool DBverify. A summary of the benchmark codes is depicted in Table 4.

We organize our experimental reports in two subsections. In the first of these, we assess the performance of deductive-based verification (symbolic execution, conditional normal form, and weakest precondition) techniques, and in the second we discuss about electing the most efficient algorithm. All experiments are conducted using a computer system equipped with core i7, 3.60 GHz CPU, 4GB memory, and Ubuntu 14.04 operating system.

6.1. Assessing the deductive-based verification techniques

DBverify accepts PL/SQL code annotated with assertions (representing properties of interest) as input, and it generates a set of VCs expressed in Z3 Language. The validity of VCs is then checked by providing their *negation* to Z3. The result `Unsat` indicates that the program satisfies its properties, whereas the result `Sat` for at least one case indicates a counter-example. For example, Z3 reports Sat for the VC (in negation form) generated from the procedure "budget" using weakest precondition and exhibits a model "[X = 1/2, CS = 5001, MP = 160001/2, DID = 0, EQ = 60001, CT = 10001, Z = 0, TA = 5/2, Y = 0]". This model serves as a counter-example for the correctness of the "budget" procedure.

Table 5 depicts detailed verification results of the benchmark applications under three deductive approaches. The procedures defined under each benchmark provide various services and they are verified at the individual level as they are independent w.r.t. each other.

Since our special focus in this experiment is to verify database properties, the third and fourth columns of the table denote the number and type of assertions each procedure is instrumented with. The assertions with which we have annotated the PL/SQL procedures are either defined as part of the table definitions or chosen based on their practical relevance w.r.t. the

Table 5
Verification results under deductive-based approaches.

DB App	Procedures	# Assertion	Assertion type	# VCs(SE)	Verification time (in millisec)			Verification results	
					SE	CNF	WP	#Valid	#Invalid
CC	update_client.sql (P1)	3	T1,T3	7	1132.382	429.915	605.012	2	1
	add_dimension_class.sql (P2)	1	T1,T3	3	489.062	141.523	223.742	1	0
	add_car.sql (P3)	2	T1-T4	36	5715.589	571.418	729.936	1	1
	add_courier.sql (P4)	7	T1-T4	726	158 809.247	1610.636	2652.89	6	1
	add_status.sql (P5)	4	T1,T3	24	3623.771	317.157	397.077	4	0
	add_parcel_type.sql (P6)	1	T1,T3,T4	3	456.85	170.738	193.391	1	0
	add_client.sql (P7)	2	T1,T3,T4	18	3681.689	305.573	323.188	0	2
	add_delivery_attempt.sql (P8)	4	T1,T3	27	3967.09	296.309	379.003	3	1
	add_parcel.sql (P9)	3	T1,T3,T4	40	7840.22	776.464	972.306	3	0
	add_warehouse.sql (P10)	2	T1,T3	24	6005.307	473.43	576.8599	2	0
	update_warehouse.sql (P11)	3	T1,T3	3	553.536	213.336	270.883	2	1
	driving_license_category (P12)	1	T1,T3	3	450.966	134.064	152.183	1	0
	get_contract_type_id.sql (P13)	1	T1,T3	3	469.39	170.085	253.615	1	0
	get_delivery_status.sql (P14)	1	T1,T3	3	444.132	163.679	197.397	1	0
	get_country_id.sql (P15)	1	T1,T3	3	450.762	171.882	256.791	1	0
	delete_client.sql (P16)	1	T1	7	141.092	107.117	119	1	0
IM	add-to-inventory.sql (P1)	1	T1	1	150.933	148.536	150.597	1	0
	assemble-module.sql (P2)	4	T1,T2	30	5822.301	195.399	437.9089	0	4
	assemble-component.sql (P3)	10	T1–T5	78	11 235.171	397.668	22 882.824	8	2
	procinventory.sql (P4)	7	T1,T3,T5	15	2577.703	730.68	854.946	6	1
	deliver.sql (P5)	3	T1,T3	3	405.958	205.872	345.739	2	1
RM	add_cust.sql (P1)	2	T1,T3,T5	9	3740.924	620.314	2561.225	0	2
	qoh_update.sql (P2)	4	T1,T3	7	1115.73	458.236	989.809	1	3
	retail-business-logic.sql (P3)	3	T1,T2,T5	72	13 464.077	153.664	45 707.734	2	1
	budget.sql (P4)	1	T1,T2,T5	2	510.087	231.96	393.028	0	1
HR	bill.sql (P1)	1	T1,T5	2	332.12	278.318	303.683	0	1
	award-bonus.sql (P2)	1	T2	2	347.44	148.783	291.376	0	1
	discount.sql (P3)	1	T1,T5	4	897.481	217.413	369.509	0	1
	resrvation-proc.sql (P4)	5	T1,T3	30	6427.385	348.296	488.749	4	1
CS	cust-emp-proc.sql (P1)	2	T1,T2	6	850.02	141.256	189.866	2	0
	cartsell.sql (P2)	5	T1-T3	30	4944.753	454.442	567.582	5	0
	loginproc.sql (P3)	2	T1-T3	9	1951.349	188.744	206.902	0	2
	product.sql (P4)	4	T1,T3	79	17 081.274	770.027	3453.8509	3	1
	transfer.sql (P5)	4	T1,T3	4	686.796	314.784	416.193	4	0
BS	credit-account.sql (P1)	1	T1	1	305.4	129.629	121.744	1	0
	debit-account.sql (P2)	1	T1,T5	2	302.246	127.123	134.39	0	1
	Procedure_transactions.sql (P3)	7	T1,T3,T4	64	19 009.815	749.565	3019.334	6	1
	check.sql (P4)	1	T1	2	294.594	118.144	123.658	0	1
CaC	AddCustomerPoints.sql (P1)	1	T1	4	952.831	222.605	232.99	0	1
	CheckPassword.sql (P2)	1	T1,T2	3	469.283	132.547	154.568	1	0
	UpdateQuantity.sql (P3)	1	T1,T2	6	902.813	150.781	233.769	1	0
	RecordNewSale.sql (P4)	1	T1	16	2544.719	307.056	911.324	0	1
	PopulateProducts.sql (P5)	1	T1,T3	4	974.6	264.795	135.582	0	1
	PopulateCustomers.sql (6P)	1	T1,T3	4	678.47	377.758	162.135	0	1
	PopulateSales.sql (P7)	4	T1–T5	1024	169 602.727	662.117	80 132.817	3	1
	DecreaseDispStock.sql (P8)	1	T1	5	765.373	190.366	313.94	0	1
	IncreaseDispStock.sql (P9)	1	T1	5	782.142	197.925	312.692	0	1
CR	isEnrollable.sql (P1)	1	T1,T3	14	4164.848	434.723	634.9329	1	0

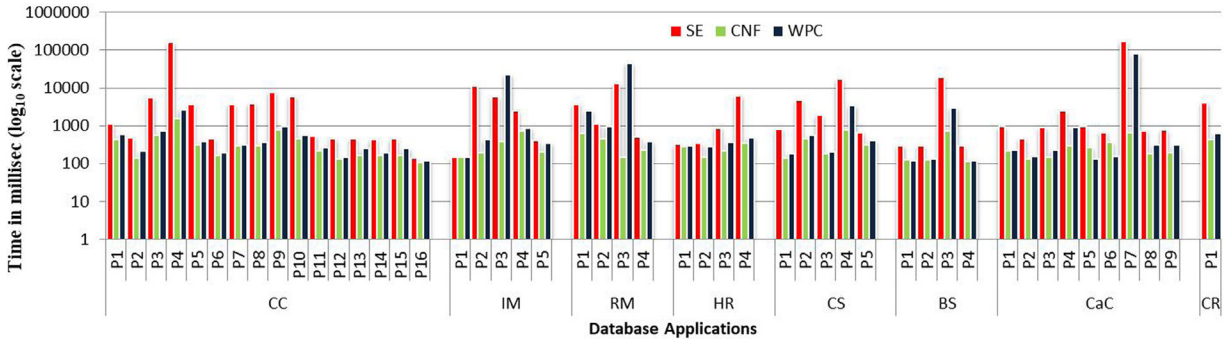


Fig. 22. Total verification time of database procedures under deductive approaches.

procedures' behaviors. The assertion types are indicated by numbers, as follows: T1: Attribute-based, T2: Tuple-based, T3: Null Value, T4: Aggregate Functions, T5: General Assertions. The fifth column indicates the number of verification conditions (VCs) generated from the procedures under Symbolic Execution-based (SE) verification. Observe that Conditional Normal Form (CNF)

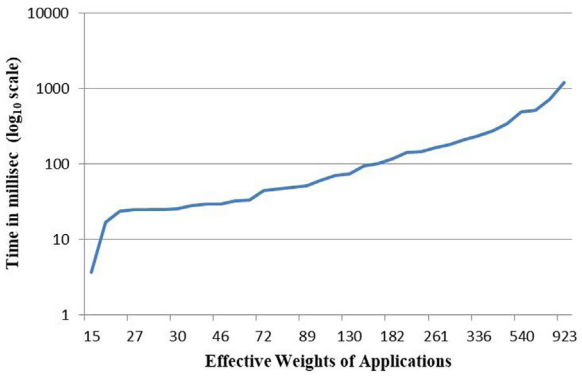


Fig. 23. DSAgen time VS effective weights of procedures.

and Weakest Precondition (WPC) always generate a single VC. We have recorded the total verification time (in millisec) for all procedures under the three approaches in columns 6–8, and their comparison is depicted in Fig. 22. It is worthwhile to observe that verification under CNF always takes less time as compared to SE and WPC. Interestingly, in the case of our benchmark codes, we also experience that SE requires longer verification time as compared to WPC in most of the cases. Arguably, the reason behind this is the generation of multiple VCs (which may contain multiple copies of the same logical encoding) in SE and their validity checking by Z3 individually (depicted in Fig. 27). However, an exception is observed in two procedures, namely P3 in the IM and RM applications, where longer VC generation time is experienced (depicted in Fig. 24) due to the presence of a high number of attributes in the postconditions as well as a higher number of SQL statements which define these attributes. Precisely, this affects the computation time from postcondition to weakest preconditions in the backward direction significantly. The verification outcomes indicating the number of assertions proved as valid and invalid are reported in columns 9 and 10 respectively. The results clearly show that only 38% of the benchmark procedures satisfy the annotated database properties, while 62% procedures violate either all or part of the annotated properties. The primary cause behind this is that most of the SQL statements in the procedures accept runtime inputs without any proper checking. On other hand, only 68% assertions are satisfied by benchmark procedures.

Let us now draw a few other crucial observations based on our experimental results:

(I) DSA Generation (DSAgen) Time: Conversion of PL/SQL codes into their equivalent DSA form is a key step in SE- and CNF-based verification. Intuitively, various factors such as the number and types of statements in the code, number of variables and attributes defined or used by the statements, number of conditions and nesting depth, etc., contribute differently to DSAgen time. To extract insightful observations on the variation of DSAgen time w.r.t. the PL/SQL codes, we have classified statements into three different categories *low*, *medium* and *high* effectiveness (denoted *l*, *m* and *h* respectively) depending upon their contributions in DSA-gen time and computed the overall weights of PL/SQL codes according to the following equation:

$$W = \sum_{i=1}^{n} (\texttt{Weight}(S_i^x) + \alpha_i) \tag{13}$$

where n is the number of statements and S_i^x denotes ith statement in the category $x \in \{l, m, h\}$. $\texttt{Weight}(S_i^x)$ returns the weight of S_i^x under its category x. α_i is an additional weight factor whose value reflects the complexity level of S_i^x, i.e. the number of attributes and variables used and defined in S_i^x. Notice that this factor is implementation-dependent. We have classified the statements as follows: *Low effectiveness* category includes variables declaration, assume, assert, cursor operations and exception handling statements. *Medium effectiveness* category includes assignment statements. *Most effectiveness* category include conditional and SQL statements. Fig. 23 depicts the variation of DSAgen time w.r.t. the weights of the codes, considering $\texttt{Weight}(S_i^l) = 1$, $\texttt{Weight}(S_i^m) = 2$, and $\texttt{Weight}(S_i^h) = 4$.

(II) VC Generation (VCgen) Time: VCgen time for all procedures under SE, CNF and WPC are depicted in Fig. 24. As expected, VCgen in CNF always takes less time as compared to the others. However, if we compare VCgen in SE and WPC, we observe that for some procedures SE performs better than WPC, while for the rest WPC performs better than SE. Although VC generation in both approaches gets highly affected by the presence of conditional statements, the primary reason behind this difference in VCgen time is as follows: In SE, the logical encoding of all statements along each path appears in its corresponding VC. This creates multiple copies of the logical encoding of the same statement in multiple VCs, if the statement appears in all those paths. Therefore, procedures having a large number of execution paths which comprise multiple copies of the same statements' logical encoding experience higher VCgen time (for example, procedures P4, P9 and P10 in CC, P2 in IM, P4 in RH, and P3 in BS). In contrast, in WPC, VC generation deals with the computation of the weakest precondition from the postcondition in a backward direction and this primarily gets affected by only those statements which define the attributes or variables that appeared in the postcondition. In particular, SQL statements as defining statements are more influential in this case, as they involve `WHERE` clause. Therefore, procedures having more such defining statements experience higher VCgen in WPC (for example, P3 in IM and RM, P4 in CS, etc.).
Let us now observe the variation of VCgen w.r.t. PL/SQL codes, identifying the influence of different types of statements and their operational complexity to the VCgen. The computation of weights of PL/SQL codes in case of SE, CNF and WPC follows similar approach as in the case of DSAgen time, with minor changes either in equation or in statements' classification. In SE and CNF, conditional statements are considered as *medium* category, whereas WPC considers the assert and conditional statements as *high* category. Although the weight computation in CNF and WPC follows Eq. (13), it differs in case of SE by taking into account various paths in the code as defined in Eq. (14).

$$W = \sum_{r=1}^{p} \Big(\sum_{i=1}^{n} (\texttt{Weight}(S_{i,r}^x) + \alpha_{i,r}) \Big) \tag{14}$$

Following Eqs. (13) and (14), the variation of VCgen time w.r.t. weights of different procedures in the case of SE, CNF, and WPC are depicted in Figs. 25(a), 25(b), and 25(c) respectively.

(III) Number of VCs: The number of VCs in the SE-based technique is same as the number of execution paths that exist in the code, and it depends on the number of conditions and their nesting structure. The presence of conditional statements without any nesting in the code yields a maximum number of paths, whereas a balanced form of nesting in both if-block and else-block yields a minimum number

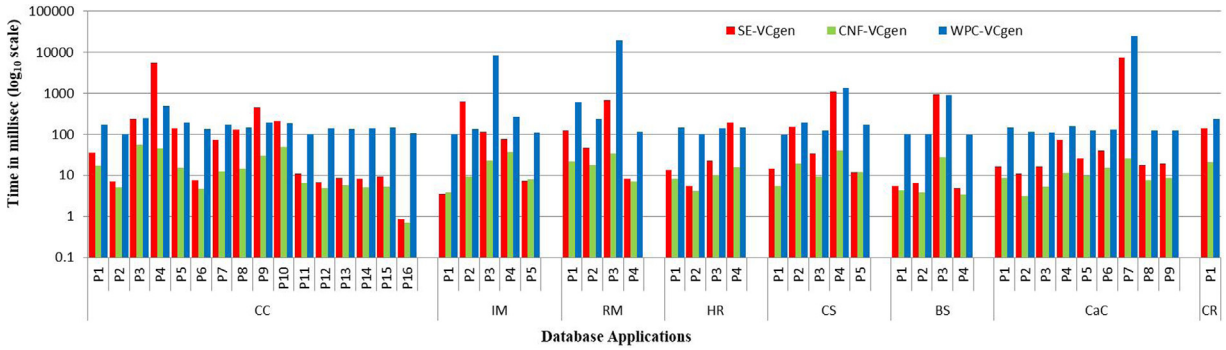
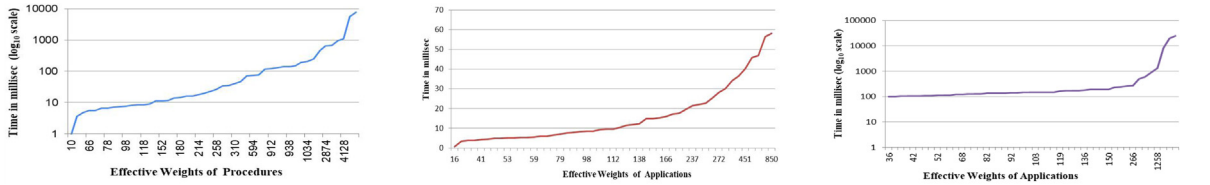


Fig. 24. VC generation (VCgen) time under deductive approaches.



(a) SE-VCgen Time VS Effective Weights of Procedures

(b) CNF-VCgen Time VS Weight of Effective of of Procedures

(c) WPC-VCgen Time VS Weight of Effective of Procedures

Fig. 25. VCgen time under deductive approaches.

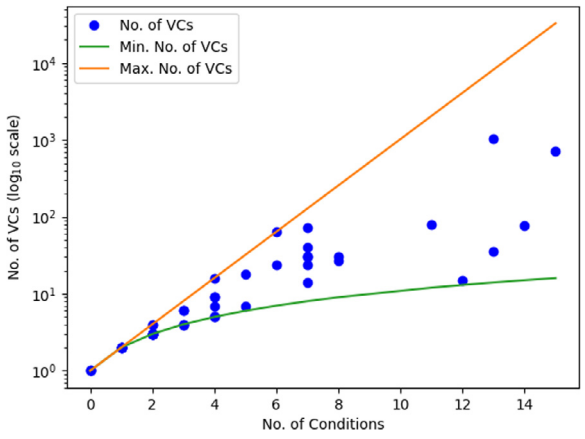


Fig. 26. Number of VCs generated in SE.

of paths in the code. Therefore, given n number of conditional statements in the code, the number of paths (hence VCs) lie within the interval $[n+1, 2^n]$. We observe that our results on the number of generated VCs lie between this allowable range depicted in Fig. 26.

(IV) Z3 Execution time (Z3exe): Since we have built our tool on the top of SMT solver Z3, the time taken by Z3 to validate the generated formula for all procedures under three deductive approaches is depicted in Fig. 27. It is observed that Z3 execution time for the VCs generated in CNF always takes less time compared to others. However, in the case of SE and WPC, it depends on the number of VCs generated in SE versus the size of VC generated in WPC.

6.2. Electing most efficient verification algorithm

While evaluating different deductive verification algorithms with a common goal, the first question that comes to mind is "which one is the most efficient or suitable algorithm?". Let us address this question by analyzing the results from the both theoretical and the practical perspectives.

As already discussed in Section 4, for a given program with n conditional statements, there exist $O(2^n)$ execution paths and therefore the SE-based algorithm results in $O(2^n)$ number of VCs. In contrast, the CNF algorithm avoids path enumeration by transforming the procedure into CNF-form. Assuming that the program contains n conditional statements with nesting depth up to m, the CNF algorithm generates a single VC of size $O(n + m^2)$. The WPC algorithm, on the other hand, generates a single VC by OR-ing the weakest preconditions generated from the postcondition along all branches of the code. This exhibits exponential $O(2^n)$ size of VC for programs with n conditional statements.

Let us now do an analysis based on our empirical study. As is clear in Fig. 24, CNF always takes less VC generation time, thus supporting the theoretical analysis results. However, in the case of SE and WPC, although both are influenced by conditional statements, the VCgen time varies due to the presence of other statements. In particular, as already mentioned before, SE performs path enumeration and takes into account the same statements multiple times during the VC generation, if the statements appear over multiple paths. This consumes a significant amount of time which varies depending upon the number of paths and their length. On the other hand, VC generation in WPC effectively depends on the computation of weakest precondition from postconditions. This consumes a considerable amount of time which increases if the postcondition involves a higher number of attributes and variables and there exist more statements which define those attributes or variables. Although, our empirical study does not allow us to quantify these factors exactly, one can conclude that programs having a lower number of defining statements and a higher number of execution paths experience higher VCgen time in SE than that in case of WPC.

A similar observation can also be drawn from the Z3 execution time, depicted in Fig. 27, where VCs generated in CNF always takes less time as compared to the execution of VCs generated in SE and WPC. However, we observe in most of the cases that Z3 execution time in SE is higher than in WPC, due to the presence of multiple VCs (with repetition of logical encoding)

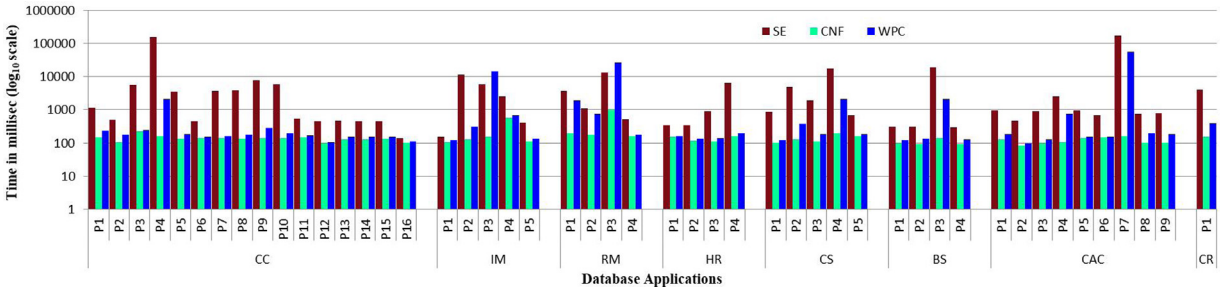


Fig. 27. Z3 execution time of deductive approaches.

whose validity is checked individually. An exception is observed in two cases, namely P3 in IM and RM, where the size of the VC generated in WPC is much higher than the total size of all VCs generated in SE. This happens due to the presence of a higher number of attributes in postcondition and also a higher number of SQL statements which define those attributes. As WPC does not require DSA conversion, this becomes a key factor on the overall verification time depicted in Fig. 22. Based on this analysis, one can now easily judge that CNF is the most efficient verification algorithm among these three. However, the decision to choose one between SE and WPC depends on the program's structure, size of the postcondition and the types of statements involved in the program.

7. Threats to validity

We first discuss the threats to external validity, which are about the generalization of our findings. Three proposed VC generation techniques in this paper are, in general, applicable to the case of database applications dealing with relational databases. In particular, our focus is to verify their correctness w.r.t. database properties. In terms of expressive power of the assertion language, although we are able to express most common database properties (Ullman, 2020), a failure is observed in the case of property involving referential integrity or a count of database records. Notably, our theoretical formalism is based on the abstract syntax of a database language embedded within an imperative host language. Although this paper considers a simple form of host imperative language, the support of dynamic memory data structures, floating points, pointers, etc., can be provided with more engineering without affecting the general idea of the proposed techniques. Besides, the abstract syntax of database extension captures crucial features of the languages used in popular structured database systems, such as MySQL, Oracle, DB2, Microsoft SQL Server, PostgreSQL, etc., with a complete support for data manipulation operations. It should be mentioned that the proposed approach does not support verification of dynamically generated database statements in the applications. The developed tool DBverify currently supports loop-free PL/SQL only. We are in the process of extending it to support loops (as highlighted in Section Section 3.5) and to the languages of other database management systems with their embedding within popular host languages (such as C, Python, Java, etc.) in the next release of the tool. For the latter case, we require little effort to modify the parser-based language processing module in DBverify according to the language's concrete syntax. Observe that, in the case of concurrent database transactions, as two or more transactions can interleave in a concrete program run, this threat can be mitigated by our current proposal through a static identification of all possible permutations of database statements present in the transactions (Beckert and Klebanov, 2007; Chrysanthis and Ramamritham, 1998), incurring an exponential computation cost (Ábrahám et al., 2005; Bruns, 2015).

Let us now discuss the threats to internal validity, which refer to experimental bias and errors. In our experiment, we have annotated benchmark PL/SQL procedures with the assertions that are already part of the underlying database table definitions or chosen based on their practical relevance w.r.t. the procedures' behavior. To this aim, we have considered the most common relational database properties, as reported in Ullman (2020). Even though our proposal covers crucial SQL features, the aggregate functions (except COUNT) are treated by means of an abstraction, which may result in false positives. The experimental results, as expected, reveal that CNF is the most efficient verification algorithm among these three. However, the decision to choose one between SE and WPC depends on the program's structure, size of the postcondition and the types of statements involved in the program.

8. Related work

Theorem proving (Deductive reasoning) (Ahrendt et al., 2016; Filliâtre, 2011; Hähnle and Huisman, 2019), model checking (algorithmic verification) (Clarke et al., 1994; Clarke Jr et al., 2018; Jhala and Majumdar, 2009) and process algebra (Fokkink, 2013) are the main categories of techniques for formally verifying properties of both hardware and software systems. The theorem proving and model checking approaches have complementary strengths and weaknesses, and their combination promises to enhance the capabilities of each (Reed et al., 1999; Uribe, 2000). Process algebra, on the other hand, constitutes a framework for formal verification of processes and data, with the emphasis on processes that are executed concurrently.

Over the decades, numerous proposals based on the above-mentioned methods have been proposed in the literature for the verification of general purpose programming languages, addressing various language features such as variables, control structures, pointers, objects, etc. (Ahrendt et al., 2016; Filliâtre, 2011; Hähnle and Huisman, 2019; Cuoq et al., 2012; Hoare, 1969). In addition, database researchers have also shown significant interest in verifying database applications using theorem proving (Baltopoulos et al., 2011; Benzaken and Schaefer, 1998; Christiansen and Martinenghi, 2003; Itzhaky et al., 2017; Malecha et al., 2010) and model checking (Abdulla et al., 2016; Calvanese et al., 2018; Deutsch et al., 2014; Gligoric and Majumdar, 2013; Vianu, 2009; Wang et al., 2017; Jana et al., 2018a). Apart from this, there have also been some proposals based on testing (Artzi et al., 2010; Chays et al., 2000; Emmi et al., 2007; Wassermann et al., 2008), equivalence checking (Chu et al., 2018, 2017), schema refactoring (Caruccio et al., 2016; Visser, 2008) as well as synthesis (Feng et al., 2017; Wang et al., 2019).

The authors in Itzhaky et al. (2017) propose an approach for verification of web applications embedded with SQL. This requires the translation of embedded SQL scripts into SmpSL functions followed by the computation of verification conditions of SmpSL functions using *weakest precondition*. The purpose is to verify

integrity constraints defined on the underlying database. The limited expressibility of SmpSL does not cover aggregate functions or arithmetic operations. Integrity constraints verification in the object-oriented database programming language O_2 is proposed in Benzaken and Schaefer (1998). For a given method m and constraint C, m cannot violate C if $\overrightarrow{m}(C) \Rightarrow C$ or $C \Rightarrow \overleftarrow{m}(C)$, where $\overrightarrow{m}(\phi)$ is a postcondition for a given precondition ϕ and $\overleftarrow{m}(\phi)$ is a precondition for a given postcondition ϕ. The proposed approach computes verification conditions using either *weakest precondition* or *strongest postcondition* computations. Authors in Christiansen and Martinenghi (2003) propose integrity constraints verification for database applications using transformation operators. The proposed approach expresses every update operation as a predicate $\mathbf{U} = P(\vec{a})$, where $\vec{a}$ denotes a sequence of constants. Integrity constraints are defined in a constraint theory τ. The function $\mathbf{after}^U(\tau)$ translates the constraint theory to the weakest precondition of ϕ with respect to the update $\mathbf{U}$ and a simplified formula is obtained by applying function $\mathbf{Simp}^U(\phi)$. The resultant formula is executed as a query and if it returns empty then the database is consistent, otherwise the returned tuple provides hints for extending the update in order to restore the consistency. Verification of database integrity constraints using refinement types is proposed in Baltopoulos et al. (2011). The proposed tool maps an SQL schema S to a Stateful F7 module $[\![S]\!]$ by using a sequence of type definitions, predicate definitions, and function signatures. User transactions are written using the functional language F#. Verification of integrity constraints starts by generating a set of verification conditions from F# and SQL codes, which are then passed to an automatic theorem prover. Unfortunately, the syntax of SQL considered in the proposed work does not include nested queries or aggregate functions. An in-memory relational database management system (RDBMS) fully verified using Coq has been proposed in Malecha et al. (2010). The authors mainly verify whether or not the RDBMS correctly executes queries w.r.t. the denotational semantics of SQL and relations.

Verification of the functional correctness of database-driven applications using model checking is proposed in Vianu (2009). The author introduces the WAVE tool, which allows the users to specify functional correctness properties using LTL formulas and verify a given database-driven application against these functional properties. The authors in Abdulla et al. (2016) consider the verification of monadic second-order properties of runs in a model where the underlying database can be updated by insertion or deletion. Decidability is obtained for recency bounded artifacts, in which only recently introduced values are retained in the current data. A theoretical approach for the verification of database-driven systems is proposed in Calvanese et al. (2018), where the authors use symbolic model-checking via model completions (equivalently, via *covers*). Interesting works on real-time and distributed systems using temporal logic to obtain verification models are proposed in Souri and Norouzi (2015) and Souri et al. (2019b). The correctness verification of service composition methods in a multi-cloud computing environment based on event-based QoS factors is presented in Souri et al. (2019a).

Our work draws motivation from da Cruz et al. (2012), Frade and Pinto (2011) and Lourenço et al. (2015). In da Cruz et al. (2012), Frade and Pinto (2011) and Lourenço et al. (2015), the authors describe in detail all three approaches, namely symbolic execution, conditional normal form, and weakest precondition, to generate VCs for imperative languages. Our work can be seen as an extension of the latter approach to the case of database applications.

9. Conclusions

This work contributes towards the verification of database applications by proposing a set of comprehensive techniques to generate Verification Conditions from database programs. With respect to the literature, the proposed approach shows its ability to support crucial SQL features along with its embedding into other host imperative languages and allow for the verification of common database properties. We also introduce DBverify, a verification tool implemented in Python based on our theoretical foundation, which enables users to verify PL/SQL procedures under three different approaches. The detailed performance analysis based on the experimental results on a set of benchmark PL/SQL codes demonstrates the effectiveness of the approaches under various circumstances. Notably, the performance of the CNF algorithm is observed better than the other two approaches in all the cases. For the given set of PL/SQL codes with chosen properties, the experimental results show that only 38% of the benchmark procedures satisfy the annotated database properties, while 62% of procedures violate either all or part of the annotated properties. The primary cause for the latter case is acceptance of runtime inputs in SQL statements without any proper checking.

CRediT authorship contribution statement

Md. Imran Alam: Conceptualization, Methodology, Prototype tool development, Benchmark codes preparation, Writing - Original and revised draft preparation, Validation. **Raju Halder:** Conceptualization, Supervision, Writing - review & editing, Validation. **Jorge Sousa Pinto:** Supervision, Writing - reviewing, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

We would like to thank Mr. Moolchandra Mridul and Mr. Nabeel Qaiser for their help in the development of DBverify. We also thank the anonymous reviewers, Area Editor, and Editor-in-Chief of the journal for their valuable comments and helpful suggestions. This work is partially supported by the IMPRINT-2 Project (IMP/2018/000523) from the Science and Engineering Research Board (SERB), Department of Science and Technology, Government of India and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project UIDB/50014/2020.

Appendix

A.1. Proofs

Lemma 2. *Let Φ be the logical encoding of an execution path π up to program point ℓ. Let $stmt^{dsa}$ be a DSA form of program statement at program point $\ell+1$. Let Ψ be the logical encoding computed using the function* Path$(\Phi, stmt^{dsa})$. *The function* Path *is correct if* $\forall\ (\rho_d, \rho_a) : (\rho_d, \rho_a) \models_I \Phi$ *under interpretation I and* $\mathscr{T}_{dba}[\![stmt^{dsa}]\!](\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$, *implies* $(\rho_{d'}, \rho_{a'}) \models_I \Psi$.

Proof. The proof is established based on structural induction. Assume that (ρ_d, ρ_a) satisfies Φ under the interpretation I, i.e. $(\rho_d, \rho_a) \models_I \Phi$.

- Assignment Statement $v_{i+1} := e_i$. Let $\mathscr{T}_{dba}[\![v_{i+1} := e_i]\!](\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$, where $(\rho_{d'}, \rho_{a'})$ is obtained by substituting all occurrences of v_{i+1} in (ρ_d, ρ_a) by $\mathscr{T}_{dba}\ [\![e_i]\!](\rho_d, \rho_a)$. Since e_i does not involve v_{i+1}, its semantics are same w.r.t. both the states (ρ_d, ρ_a) and $(\rho_{d'}, \rho_{a'})$. Therefore,

$$(\rho_{d'}, \rho_{a'}) \models_I (v_{i+1} == e_i) \tag{15}$$

On the other hand, since v_{i+1} is not present in Φ due to DSA property and only v_{i+1} is affected in $(\rho_{d'}, \rho_{a'})$, we have

$$(\rho_{d'}, \rho_{a'}) \models_I \Phi \tag{16}$$

Combining Eqs. (15) and (16), we get $(\rho_{d'}, \rho_{a'}) \models_I (\Phi \wedge v_{i+1} == e_i)$.

- Database Statements Q. Since a database statement Q involves action 'A' and condition 'cond', this can be treated as a guarded command equivalent to "if *cond* then A". Consider the DSA form $\langle \vec{a}_{i+1} := \text{act}_i^1 \curvearrowright \text{act}_i^2,\ cond_i^1 \curvearrowright cond_i^2 \rangle$ of different database statements. By the definition of the function Path, we have:

$$\begin{aligned}\text{Path}(\Phi,\ \langle \vec{a}_{i+1} :=\ & \text{act}_i^1 \curvearrowright \text{act}_i^2,\ cond_i^1 \curvearrowright cond_i^2 \rangle) = \\ & \text{Path}(\Phi, \langle \vec{a}_{i+1} := \text{act}_i^1,\ cond_i^1 \rangle) \vee \\ & \text{Path}(\Phi, \langle \vec{a}_{i+1} := \text{act}_i^2,\ cond_i^2 \rangle)\end{aligned}$$

Let us now prove the lemma for each database action:

  - $Q_{upd}^{dsa} \triangleq \langle \vec{a}_{i+1} := \texttt{UPDATE}(\vec{a}_i) \curvearrowright \texttt{UPDATE}(\vec{e}_i),\ \neg cond_i \curvearrowright cond_i \rangle$.
  Given a state (ρ_d, ρ_a), assume that $\mathscr{T}_{dba}[\![Q_{upd}^{dsa}]\!](\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$. By the definition of the function Path, we have

$$\Psi = \text{Path}(\Phi,\ Q_{upd}^{dsa}) = \Phi \wedge ((\neg cond_i \wedge \bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j == a_i^j) \vee (cond_i \wedge \bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j == e_i^j))$$

  Since $(\rho_d, \rho_a) \models_I \Phi$ and Φ does not involve $\vec{a}_{i+1}$ due to DSA property, the following holds

$$(\rho_{d'}, \rho_{a'}) \models_I \Phi \tag{17}$$

  Now we have two possibilities: either $(\rho_{d'}, \rho_{a'}) \models_I \neg cond_i$ or $(\rho_{d'}, \rho_{a'}) \models_I cond_i$. In the former case, the action $\vec{a}_{i+1} := \texttt{UPDATE}(\vec{a}_i)$ results into the same semantics for $\vec{a}_i$ and $\vec{a}_{i+1}$ w.r.t. $(\rho_{d'}, \rho_{a'})$, i.e. $\mathscr{T}_{dba}[\![\vec{a}_i]\!](\rho_{d'}, \rho_{a'}) = \mathscr{T}_{dba}[\![\vec{a}_{i+1}]\!](\rho_{d'}, \rho_{a'})$. Therefore,

$$(\rho_{d'}, \rho_{a'}) \models_I (\neg cond_i \wedge \bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j == a_i^j) \tag{18}$$

  In the latter case, the attributes $a_{i+1}^j \in \vec{a}_{i+1}$ in $(\rho_{d'}, \rho_{a'})$, where $j = 1 \ldots |\vec{a}_{i+1}|$, are substituted by $\mathscr{T}_{dba}[\![e_i^j]\!](\rho_d, \rho_a)$ respectively. Since only the value of $\vec{a}_{i+1}$ is affected in $(\rho_{d'}, \rho_{a'})$, we have $\mathscr{T}_{dba}[\![\vec{a}_{i+1}]\!](\rho_{d'}, \rho_{a'}) = \mathscr{T}_{dba}[\![\vec{e}_i]\!](\rho_{d'}, \rho_{a'})$, and therefore

$$(\rho_{d'}, \rho_{a'}) \models_I (cond_i \wedge \bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j == e_i^j) \tag{19}$$

  Hence, combining Eqs. (17)–(19), $(\rho_{d'}, \rho_{a'}) \models_I \Psi$ is proved.
  - $Q_{ins}^{dsa} \triangleq \langle \vec{a}_{i+1} := \texttt{UPDATE}(\vec{a}_i) \curvearrowright \texttt{INSERT}(\vec{e}_i), true \curvearrowright false \rangle$. Given a state (ρ_d, ρ_a), assume that $\mathscr{T}_{dba}[\![Q_{ins}^{dsa}]\!]$ $(\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$. By the definition of the function Path, we have

$$\Psi = \text{Path}(\Phi,\ Q_{ins}^{dsa}) = \Phi \wedge ((\bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j == a_i^j) \vee (\bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j == e_i^j))$$

  Since $(\rho_d, \rho_a) \models_I \Phi$ and Φ does not involve $\vec{a}_{i+1}$ due to DSA property, we have

$$(\rho_{d'}, \rho_{a'}) \models_I \Phi \tag{20}$$

  According to the action $\vec{a}_{i+1} := \texttt{UPDATE}(\vec{a}_i)$ in Q_{ins}^{dsa}, the semantics of $\vec{a}_i$ and $\vec{a}_{i+1}$ w.r.t. (ρ_d, ρ_a) are same, i.e., $\mathscr{T}_{dba}[\![\vec{a}_i]\!](\rho_{d'}, \rho_{a'}) = \mathscr{T}_{dba}[\![\vec{a}_{i+1}]\!](\rho_{d'}, \rho_{a'})$. Therefore,

$$(\rho_{d'}, \rho_{a'}) \models_I (\bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j == a_i^j) \tag{21}$$

  However, according to the second action $\vec{a}_{i+1} := \texttt{INSERT}(\vec{e}_i)$, a single tuple containing values $\mathscr{T}_{dba}[\![e_i^j]\!]$ (ρ_d, ρ_a) corresponding to attributes a_{i+1}^j where $j = 1 \ldots |\vec{a}_{i+1}|$ is inserted into the database. Since only the value of $\vec{a}_{i+1}$ is affected in $(\rho_{d'}, \rho_{a'})$, we have $\mathscr{T}_{dba}[\![\vec{a}_{i+1}]\!](\rho_{d'}, \rho_{a'}) = \mathscr{T}_{dba}[\![\vec{e}_i]\!](\rho_{d'}, \rho_{a'})$, and therefore

$$(\rho_{d'}, \rho_{a'}) \models_I (\bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j == e_i^j) \tag{22}$$

  Hence, combining Eqs. (20)–(22), $(\rho_{d'}, \rho_{a'}) \models_I \Psi$ is proved.
  - $Q_{del}^{dsa} \triangleq \langle \vec{a}_{i+1} := \texttt{UPDATE}(\vec{a}_i) \curvearrowright \texttt{DELETE}(),\ \neg cond_i \curvearrowright cond_i \rangle$. Given a state (ρ_d, ρ_a), assume that $\mathscr{T}_{dba}[\![Q_{ins}^{dsa}]\!]$ $(\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$. By the definition of the function Path, we have

$$\Psi = \text{Path}(\Phi,\ Q_{del}^{dsa}) = \Phi \wedge (\neg cond_i \bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j == a_i^j)$$

  Since $(\rho_d, \rho_a) \models_I \Phi$ and Φ does not involve $\vec{a}_{i+1}$ due to DSA property, the following holds

$$(\rho_{d'}, \rho_{a'}) \models_I \Phi \tag{23}$$

  Like previous cases, we have two possibilities: either $(\rho_{d'}, \rho_{a'}) \models_I \neg cond_i$ or $(\rho_{d'}, \rho_{a'}) \models_I cond_i$. In the former case, similar to Eq. (6), we have

$$(\rho_{d'}, \rho_{a'}) \models_I (\neg cond_i \wedge \bigwedge_{j=1}^{|\vec{a}_{i+1}|} a_{i+1}^j == a_i^j) \tag{24}$$

  While in the latter case, tuples satisfying $cond_i$ are removed from the database table. Therefore, $(\rho_{d'}, \rho_{a'}) \models_I \Psi$ is proved.
  - $Q_{sel}^{dsa} \triangleq \langle rs_{i+1} := \texttt{SELECT}(f(\vec{e}_i), r(\vec{h}(\vec{a}_i)),\ \phi', g(\vec{e}_i)), cond_i \rangle$. Let $\mathscr{T}_{dba}[\![Q_{sel}^{dsa}]\!](\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$. According to the Path function, we have $\Psi = \text{Path}(\Phi, Q_{sel}^{dsa}) = \Phi \wedge ((cond_i \wedge rs_{i+1} = \text{F}(\vec{a}_i)) \vee (\neg cond_i \wedge rs_{i+1} = rs_i))$. Following the similar direction as above, we can easily prove $(\rho_{d'}, \rho_{a'}) \models_I \Psi$ by taking the following facts into the consideration:
    (a) The affected application variable rs_{i+1} is not involved in Φ, so $(\rho_{d'}, \rho_{a'}) \models_I \Psi$.
    (b) When $(\rho_d, \rho_a) \models_I \neg cond_i$ then $(\rho_{d'}, \rho_{a'}) \models_I (\neg cond_i \wedge rs_{i+1} = rs_i)$
    (c) When $(\rho_d, \rho_a) \models_I cond_i$ then $(\rho_{d'}, \rho_{a'}) \models_I (cond_i \wedge rs_{i+1} = \text{F}(\vec{a}_i))$. □

Theorem 5. *Given an annotated program {assume ϕ_1; $\mathcal{P}$; assert ϕ_2; }. If the verification condition $\phi_1 \implies wp(\mathcal{P}, \phi_2)$ is valid, then $\mathcal{P}$ satisfies the property ϕ_2 and vice versa.*

Proof. The proof is based on structural induction on $stmt \in \mathcal{P}$. Recall the definition of wp in Fig. 16 and let $\text{wp}(stmt, \psi_2) = \psi_1$. We have to prove that, if $(\rho_d, \rho_a) \models_I \phi_1$ and $\phi_1 \rightarrow \psi_1$ and $\psi_2 \rightarrow \phi_2$ and $\mathcal{T}_{dba}[\![stmt]\!](\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$, then $(\rho_{d'}, \rho_{a'}) \models_I \psi_2$. Let us now consider the different cases:

- Assignment $\triangleq v := e$. Let $(\rho_d, \rho_a) \models_I \phi_1$. According to the definition, $\text{wp}(v := e, \psi_2) = \psi_2[e/v]$. Assume that $\phi_1 \rightarrow \psi_2[e/v]$. Therefore,

$$(\rho_d, \rho_a) \models_I \psi_2[e/v] \tag{25}$$

  According to the transition semantics, assume $\mathcal{T}_{dba}[\![v := e]\!](\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$ such that

$$\rho_a(z) = \begin{cases} m, \ if \ z = v \\ \rho_a(z), \ Otherwise \end{cases}$$

  where $m = \mathcal{T}_{dba}[\![e]\!](\rho_d, \rho_a)$. Since, in $(\rho_{d'}, \rho_{a'})$ all occurrences of x are replaced by m, according to Eq. (25), we have $(\rho_{d'}, \rho_{a'}) \models_I \Psi_2$. Assuming $\psi_2 \rightarrow \phi_2$, this case is proved.
- $Q_{upd} \triangleq \langle \vec{a} := \texttt{UPDATE}(\vec{e}), cond \rangle$. Let $(\rho_d, \rho_a) \models_I \phi_1$. According to the definition, $\text{wp}(\langle \vec{a} := \texttt{UPDATE}(\vec{e}), cond \rangle, \psi_2) = ((\psi_2 \wedge \neg cond) \vee (\psi_2[\vec{e}/\vec{a}] \wedge cond))$. Assume that $\phi_1 \rightarrow ((\psi_2 \wedge \neg cond) \vee (\psi_2[\vec{e}/\vec{a}] \wedge cond))$. Therefore,

$$(\rho_d, \rho_a) \models_I ((\psi_2 \wedge \neg cond) \vee (\psi_2[\vec{e}/\vec{a}] \wedge cond)) \tag{26}$$

  According to the semantics function, let us assume $\mathcal{T}_{dba}[\![Q_{upd}]\!](\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$, where

  (i) Values of $\vec{a}$ will be updated by $\mathcal{T}_{dba}[\![\vec{e}]\!](\rho_d, \rho_a)$ for the tuples satisfying '*cond*'. That is, like assignment statement, we have

$$(\rho_{d'}, \rho_{a'}) \models_I \Psi_2 \ when \ (\rho_d, \rho_a) \models_I \Psi_2[\vec{e}/\vec{a}] \wedge cond \tag{27}$$

  (ii) Values of $\vec{a}$ will remain unchanged for those tuples which do not satisfy '*cond*'. That is,

$$(\rho_{d'}, \rho_{a'}) \models_I \Psi_2 \ when \ (\rho_d, \rho_a) \models_I \Psi_2 \wedge \neg cond \tag{28}$$

  Combining Eqs. (26)–(28), the lemma is proved for Q_{upd} assuming $\Psi_2 \rightarrow \phi_2$.
- $Q_{ins} \triangleq \langle \vec{a} := \texttt{INSERT}(\vec{e}), false \rangle$. According to definition of wp, $\text{wp}(\langle \vec{a} := \texttt{INSERT}(\vec{e}), false \rangle, \psi_2) = (\psi_2[\vec{e}/\vec{a}] \vee \psi_2)$. Let $(\rho_d, \rho_a) \models_I \phi_1$ and $\phi_1 \rightarrow (\psi_2[\vec{e}/\vec{a}] \vee \psi_2)$. Then,

$$(\rho_d, \rho_a) \models_I (\psi_2[\vec{e}/\vec{a}] \vee \psi_2) \tag{29}$$

  Now given $\mathcal{T}_{dba}[\![Q_{ins}]\!](\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$, we have two cases:

  (i) Values of existing tuples will remain unchanged. Therefore,

$$(\rho_{d'}, \rho_{a'}) \models_I \Psi_2 \ when \ (\rho_d, \rho_a) \models_I \Psi_2 \tag{30}$$

  (ii) For the newly inserted tuple, we have

$$(\rho_{d'}, \rho_{a'}) \models_I \Psi_2 \ when \ (\rho_d, \rho_a) \models_I \Psi_2[\vec{e}/\vec{a}] \tag{31}$$

  Assuming $\Psi_2 \rightarrow \phi_2$, and combining Eqs. (29), (30), and (31), the lemma is proved for Q_{ins}.
- $Q_{del} \triangleq \langle \vec{a} := \texttt{DELETE}(), cond \rangle$. Let $(\rho_d, \rho_a) \models_I \phi_1$. According to the definition of wp, $\text{wp}(\langle \vec{a} := \texttt{DELETE}(), cond \rangle, \psi_2) = \psi_2 \wedge \neg cond$. Assume $(\rho_d, \rho_a) \models_I \phi_1$ and $\phi_1 \models_I (\psi_2 \wedge \neg cond)$. Therefore,

$$(\rho_d, \rho_a) \models (\psi_2 \wedge \neg cond) \tag{32}$$

  Let $\mathcal{T}_{dba}[\![Q_{del}]\!](\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'})$. Since the tuples which remain intact in the database after the DELETE action, do not satisfy '*cond*', therefore, $(\rho_{d'}, \rho_{a'}) \models_I \Psi_2$ when $(\rho_d, \rho_a) \models (\psi_2 \wedge \neg cond)$. Assuming $\Psi_2 \rightarrow \phi_2$, the lemma is proved for Q_{del}.
- $Q_{sel} \triangleq \langle rs := \texttt{SELECT}(f(\vec{e}), r(\vec{h}(\vec{x})), \phi, g(\vec{e})), cond \rangle$. Since the SELECT statement does not affect any database attributes, this case can be proved easily as ϕ_1 and ϕ_2 only involve database attributes. □

References

Abdulla, P.A., Aiswarya, C., Atig, M.F., Montali, M., Rezine, O., 2016. Recency-bounded verification of dynamic database-driven systems. In: Proceedings of the 35th ACM SIGMOD. pp. 195–210.

Ábrahám, E., de Boer, F.S., de Roever, W.-P., Steffen, M., 2005. An assertion-based proof system for multithreaded java. TCS 331 (2–3), 251–290.

Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M., 2016. Deductive software verification–the key book. Lecture Notes in Comput. Sci. 10001.

Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., Ernst, M.D., 2010. Finding bugs in web applications using dynamic test generation and explicit-state model checking. IEEE TSE 36 (4), 474–494.

Aycock, J., Horspool, N., 2000. Simple generation of static single-assignment form. In: Watt, D.A. (Ed.), Compiler Construction. Springer Berlin Heidelberg, Berlin, Germany, pp. 110–125.

Baltopoulos, I.G., Borgström, J., Gordon, A.D., 2011. Maintaining database integrity with refinement types. In: European Conference on Object-Oriented Programming. Springer, Berlin, Heidelberg, pp. 484–509.

Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M., 2005. Boogie: A modular reusable verifier for object-oriented programs. In: International Symposium on Formal Methods for Components and Objects. Springer, pp. 364–387.

Beckert, B., Klebanov, V., 2007. A dynamic logic for deductive verification of concurrent programs. In: In. Conf.on SEFM. IEEE, pp. 141–150.

Benzaken, V., Schaefer, X., 1998. Static management of integrity in object-oriented databases: Design and implementation. In: Int. Conf. on Extending Database Technology. Springer, pp. 309–325.

Bjørner, D., Havelund, K., 2014. 40 years of formal methods. In: International Symposium on Formal Methods. Springer, Cham, pp. 42–61.

Briggs, P., Cooper, K.D., Harvey, T.J., Simpson, L.T., 1998. Practical improvements to the construction and destruction of static single assignment form. Softw. - Pract. Exp. 28 (8), 859–881.

Bruns, D., 2015. Deductive Verification of Concurrent Programs. KIT, Fakultät für Informatik.

Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A., 2018. Quantifier elimination for database driven verification. arXiv preprint arXiv:1806.09686.

Caruccio, L., Polese, G., Tortora, G., 2016. Synchronization of queries and views upon schema evolutions: A survey. TODS 41 (2), 1–41.

Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E., 2005. Beyond assertions: Advanced specification and verification with jml and esc/java2. In: International Symposium on Formal Methods for Components and Objects. Springer, pp. 342–363.

Chays, D., Dan, S., Frankl, P.G., Vokolos, F.I., Weyuker, E.J., 2000. A framework for testing database applications. In: Proceedings of the 2000 ACM SIGSOFT ISSTA. pp. 147–157.

Christiansen, H., Martinenghi, D., 2003. Simplification of database integrity constraints revisited: A transformational approach. In: Int. Symposium on Logic-Based Program Synthesis and Transformation. Springer, pp. 178–197.

Chrysanthis, P.K., Ramamritham, K., 1998. Correctness criteria and concurrency control. Manage. Heterog. Auton. Database Syst..

Chu, S., Murphy, B., Roesch, J., Cheung, A., Suciu, D., 2018. Axiomatic foundations and algorithms for deciding semantic equivalences of sql queries. Proc. VLDB Endow. 11 (11), 1482–1495.

Chu, S., Wang, C., Weitz, K., Cheung, A., 2017. Cosette: An automated prover for sql. In: CIDR.

Clarke, E.M., Grumberg, O., Long, D.E., 1994. Model checking and abstraction. ACM Trans. Program. Lang. Syst. (TOPLAS) 16 (5), 1512–1542.

Clarke, E., Kroening, D., Lerda, F., 2004. A tool for checking ansi-c programs. In: In Proc. of TACAS. Springer, pp. 168–176.

Clarke Jr, E.M., Grumberg, O., Kroening, D., Peled, D., Veith, H., 2018. Model Checking. MIT press.

Cortesi, A., Halder, R., 2013. Abstract interpretation of recursive queries. In: ICDCIT. Springer, pp. 157–170.

Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B., 2012. Frama-c. In: Int. Conf.on SEFM. Springer, pp. 233–247.

da Cruz, D., Frade, M.J., Pinto, J.S., 2012. Verification conditions for single-assignment programs. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing. ACM, pp. 1264–1270.

De Moura, L., Bjørner, N., 2008. Z3: an efficient smt solver. In: Int. conf. on TACAS. Springer, pp. 337–340.
Deutsch, A., Hull, R., Vianu, V., 2014. Automatic verification of database-centric systems. ACM SIGMOD Rec. 43 (3), 5–17.
Dijkstra, E.W., Dijkstra, E.W., Dijkstra, E.W., Informaticien, E.-U., Dijkstra, E.W., 1976. A Discipline of Programming, Vol. 1. prentice-hall Englewood Cliffs.
Elmasri, R., Navathe, S.B., 2011. Database Systems, Vol. 9. Pearson Education Boston, MA.
Emmi, M., Majumdar, R., Sen, K., 2007. Dynamic test input generation for database applications. In: Proc of ISSTA. pp. 151–162.
Feng, Y., Martins, R., Van Geffen, J., Dillig, I., Chaudhuri, S., 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In: Proc. of PLDI. pp. 422–436.
Filliâtre, J.-C., 2011. Deductive Software Verification. Springer.
Filliâtre, J.-C., Paskevich, A., 2013. Why3—where programs meet provers. In: European Symposium on Programming. Springer, pp. 125–128.
Flanagan, C., Saxe, J.B., 2001. Avoiding exponential explosion: Generating compact verification conditions. ACM SIGPLAN Not. 36 (3), 193–205.
Fokkink, W., 2013. Introduction to Process Algebra. springer science & Business Media.
Frade, M.J., Pinto, J.S., 2011. Verification conditions for source-level imperative programs. Comp. Sci. Rev. 5 (3), 252–277.
Furia, C.A., Meyer, B., Velder, S., 2014. Loop invariants: Analysis, classification, and examples. ACM Comput. Surv. 46 (3), 34.
Gligoric, M., Majumdar, R., 2013. Model checking database applications. In: Proc. of TACAS. Springer, pp. 549–564.
Hähnle, R., Huisman, M., 2017. 24 challenges in deductive software verification. In: ARCADE@ CADE. pp. 37–41.
Hähnle, R., Huisman, M., 2019. Deductive software verification: from pen-and-paper proofs to industrial tools. In: Computing and Software Science. Springer, pp. 345–373.
Halder, R., Cortesi, A., 2012. Abstract interpretation of database query languages. CLSS 38 (2), 123–157.
Hoare, C.A.R., 1969. An axiomatic basis for computer programming. Commun. ACM 12 (10), 576–580.
Itzhaky, S., Kotek, T., Rinetzky, N., Sagiv, M., Tamir, O., Veith, H., Zuleger, F., 2017. On the automated verification of web applications with embedded sql. In: Proc. of ICDT. p. 1.
Jana, A., Alam, M.I., Halder, R., 2018a. A symbolic model checker for database programs. In: ICSOFT. pp. 381–388.
Jana, A., Halder, R., Kalahasti, A., Ganni, S., Cortesi, A., 2018b. Extending abstract interpretation to dependency analysis of database applications. IEEE TSE 46 (5), 463–494.
Jhala, R., Majumdar, R., 2009. Software model checking. ACM Comput. Surv. 41 (4), 1–54.
Kroening, D., Weissenbacher, G., 2010. Verification and falsification of programs with loops using predicate abstraction. Form. Asp. Comput. 22 (2), 105–128.
Leino, K., Rustan, M., 2005. Efficient weakest preconditions. Inform. Process. Lett. 93 (6), 281–288.
Lourenço, C.B., Lamraoui, S.-M., Nakajima, S., Pinto, J.S., 2015. Studying verification conditions for imperative programs. Electron. Commun. EASST 72.
Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R., 2010. Toward a verified relational database management system. ACM SIGPLAN Not. 45 (1), 237–248.
Parr, T., 2013. The Definitive ANTLR 4 Reference. Pragmatic Bookshelf.
PL/SQL Project, 2020a. Github pl/sql project. https://github.com/mdjdrn1/CourierCompanyDatabase. [Online accessed 20-May-2020].
PL/SQL Project, 2020b. Github pl/sql project. https://github.com/gamunu/OrganizedIn. [Online accessed 20-May-2020].
PL/SQL Project, 2020c. Github pl/sql project. https://github.com/kyleruss/cablecity-db. [Online accessed 20-May-2020].
PL/SQL Project, 2020d. Github pl/sql project. https://github.com/Lewan110/hotel-room-reservation-database. [Online accessed 20-May-2020].
PL/SQL Project, 2020e. Github pl/sql project. https://github.com/abhijit-bhandarkar/Retail-Business-Management-System. [Online accessed 20-May-2020].
PL/SQL Project, 2020f. Github pl/sql project. https://github.com/nafis00141/Computer-Store-Management-System. [Online accessed 20-May-2020].
PL/SQL Project, 2020g. Github pl/sql project. https://github.com/NabidAlam/Bank-Management-Project-PL-SQL. [Online accessed 20-May-2020].
PL/SQL Project, 2020h. Github pl/sql project. https://github.com/VivekBhat/Course-Registration-System-DBMS. [Online accessed 20-May-2020].
Reed, J., Sinclair, J., Guigand, F., 1999. Deductive reasoning versus model checking: Two formal approaches for system development. In: Proceedings of the 1st International Conference on Integrated Formal Methods. Springer, York, UK, pp. 429–430.
Souri, A., Norouzi, M., 2015. A new probable decision making approach for verification of probabilistic real-time systems. In: 2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS). IEEE, pp. 44–47.
Souri, A., Rahmani, A.M., Navimipour, N.J., Rezaei, R., 2019a. A hybrid formal verification approach for qos-aware multi-cloud service composition. Cluster Comput. 1–18.
Souri, A., Rahmani, A.M., Navimipour, N.J., Rezaei, R., 2019b. A symbolic model checking approach in formal verification of distributed systems. Hum.-Cent. Comput. Inf. Sci. 9 (1), 4.
Ullman, J.D., 2020. Database constraints. http://infolab.stanford.edu/~ullman/fcdb/jw-notes06/constraints.html. [Online accessed 20-May-2020].
Uribe, T.E., 2000. Combinations of model checking and theorem proving. In: Kirchner, H., Ringeissen, C. (Eds.), Frontiers of Combining Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 151–170.
Vianu, V., 2009. Automatic verification of database-driven systems: a new frontier. In: Proceedings of the 12th International Conference on Database Theory. pp. 1–13.
Visser, J., 2008. Coupled transformation of schemas, documents, queries, and constraints. Electron. Notes TCS 200 (3), 3–23.
Wang, Y., Dillig, I., Lahiri, S.K., Cook, W.R., 2017. Verifying equivalence of database-driven applications. PACMPL 2 (POPL), 1–29.
Wang, Y., Dong, J., Shah, R., Dillig, I., 2019. Synthesizing database programs for schema refactoring. In: Proc. of PLDI. pp. 286–300.
Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., Su, Z., 2008. Dynamic test input generation for web applications. In: Proc. of ISSTA. pp. 249–260.
Weiß, B., 2011. Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction. KIT Scientific Publishing.
Winskel, G., 1993. The Formal Semantics of Programming Languages: an Introduction. MIT press.

Md. Imran Alam received an M.Tech degree in Computer Science and Engineering from the Indian Institute of Technology Patna, India, in 2015. He is currently pursuing a Ph.D. degree from the Department of Computer Science and Engineering, Indian Institute of Technology Patna, India. His research interest includes Databases, Formal Methods, Program Analysis and Verification, Model Checking, Information Flow Security Analysis, Blockchain, and Machine learning.

Raju Halder received the doctoral degree from the Universitá Ca' Foscari Venezia, Italy, in 2012. He is an assistant professor in the Department of Computer Science and Engineering, IIT Patna, India. Before joining IIT Patna, he served as a post doctoral researcher at Macquarie University, Australia. He worked with the Robotics team at HASLab (University of Minho), Portugal, in 2016. Prior to his Ph.D., he also worked as an associate system engineer in IBM India Pvt. Ltd. during 2007–2008. His areas of research interests include formal methods, blockchain technology, program analysis and verification, data privacy and security.

Jorge Sousa Pinto is an Associate Professor with Habilitation at the Department of Informatics of the University of Minho and a researcher at HASLab/INESC TEC. He obtained his degree of Docteur de L'Ecole Polytechnique (Paris) in 2001 and has in the past worked on linear logic and functional programming. More recently his work focused on deductive software verification. He has published around 60 papers in international conferences and journals, and is one of the authors of the textbook "Rigorous Software Development: an Introduction to Program Verification".