On the Automated Verification of Web Applications with Embedded SQL

Shachar Itzhaky¹, Tomer Kotek∗², Noam Rinetzky³, Mooly Sagiv⁴, Orr Tamir⁵, Helmut Veith†⁶, and Florian Zuleger⁷

1 Massachusetts Institute of Technology, Cambridge, MA, USA
2 Vienna University of Technology, Vienna, Austria
3 Tel Aviv University, Tel Aviv, Israel
4 Tel Aviv University, Tel Aviv, Israel
5 Tel Aviv University, Tel Aviv, Israel
6 Vienna University of Technology, Vienna, Austria
7 Vienna University of Technology, Vienna, Austria

Abstract
A large number of web applications is based on a relational database together with a program, typically a script, that enables the user to interact with the database through embedded SQL queries and commands. In this paper, we introduce a method for formal automated verification of such systems which connects database theory to mainstream program analysis. We identify a fragment of SQL which captures the behavior of the queries in our case studies, is algorithmically decidable, and facilitates the construction of weakest preconditions. Thus, we can integrate the analysis of SQL queries into a program analysis tool chain. To this end, we implement a new decision procedure for the SQL fragment that we introduce. We demonstrate practical applicability of our results with three case studies, a web administrator, a simple firewall, and a conference management system.

1998 ACM Subject Classification D.3.2 Database Management Languages, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases SQL, Scripting language, Web services, Program verification, Two-variable fragment of First Order logic, Decidability, Reasoning

Digital Object Identifier 10.4230/LIPIcs.ICDT.2017.16

1 Introduction
Web applications are often written in a scripting language such as PHP and store their data in a relational database which they access using SQL queries and data-manipulating commands [37]. This combination facilitates fast development of web applications, which exploit the reliability and efficiency of the underlying database engine and use the flexibility of the script language to interact with the user. While the database engine is typically a mature software product with few if any severe errors, the script with the embedded SQL statements does not meet the same standards of quality.

* Tomer Kotek, Helmut Veith, and Florian Zuleger were partially supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF). This work was done in part while Tomer Kotek was visiting the Simons Institute for the Theory of Computing.
† The tragic death of Helmut Veith prevented him from approving the final version. All faults and inaccuracies belong to his co-authors.

© Shachar Itzhaky, Tomer Kotek, Noam Rinetzky, Mooly Sagiv, Orr Tamir, Helmut Veith, and Florian Zuleger; licensed under Creative Commons License CC-BY 4.0

20th International Conference on Database Theory (ICDT 2017).
Editors: Michael Benedikt and Giorgio Orsi; Article No. 16; pp. 16:1–16:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
On the Automated Verification of Web Applications with Embedded SQL

With a few exceptions (such as [15, 19]) the systematic analysis of programs with embedded-SQL statements has been a blind spot in both the database and the computer-aided verification community. The verification community has mostly studied the analysis of programs which fall into two classes: programs with (i) numeric variables and complex control structure, (ii) complex pointer structures and objects; however, the modeling of data and their relationships has not received the same attention. Research in the database community on the other hand has traditionally focused on correct design of databases rather than correct use of databases.

Our long-term research vision is to transfer and extend the techniques from the verification and program analysis community to the realm of programs with embedded SQL. Since the seminal papers of Hoare, the first step for developing a program analysis is a precise mathematical framework for defining programming semantics and correctness. In this paper we develop a Hoare logic for a practically useful but simple fragment of SQL, called SmpSQL, and a simple scripting language, called SmpSL, which has access to SmpSQL statements. Specifically, we describe a decidable logic for formulating specifications and develop a weakest precondition calculus for SmpSL programs; thus our Hoare logic allows to automatically discharge verification conditions. When analyzing SmpSL programs, we treat SQL as a black box library whose semantics is given by database theory. Thus we achieve verification results relative to the correctness of the underlying database engine.

We recall from Codd’s theorem [13] that the core of SQL is equivalent in expressive power to first-order logic FO. Thus, it follows from Trakhtenbrot’s theorem [35] that it is undecidable whether an SQL query guarantees a given post condition. We have therefore chosen our SQL fragment SmpSQL such that it captures an interesting class of SQL commands, but corresponds to a decidable fragment of first-order logic, namely FO_{BD}^2, the restriction of first-order logic in which all variables aside from two range over fixed finite domains called bounded domains. The decidability of the finite satisfiability problem of FO_{BD}^2 follows from that of FO^2, the fragment of first-order logic which uses only two variables. Although the decidability of FO^2 was shown by Mortimer [30] and a complexity-wise tight decision procedure was later described by Grädel, Kolaitis and Vardi [21], we provide the first efficient implementation of finite satisfiability of FO^2.

We illustrate our methodology on the example of a simple web administration tool based on [22]. The PANDA web administrator is a simple public domain web administration tool written in PHP. We describe in Section 2 how the core mailing-list administration functionality falls into the scope of SmpSL. We formulate a specification consisting of a database invariant and pre- and postconditions. Our framework allows us to automatically check the correctness of such specifications using our own FO_{BD}^2 reasoning tool.

Main contributions

1. We define SmpSQL, an SQL fragment which is contained in FO_{BD}^2.
2. We define a simple imperative script language SmpSL with embedded SmpSQL statements.
3. We give a construction for weakest preconditions in FO_{BD}^2 for SmpSL.
4. We implemented the weakest precondition computation for SmpSL.
5. We implemented a decision procedure for FO_{BD}^2. The procedure is based on the decidability and NEXPTIME completeness result for FO^2 by [21], but we use a more involved algorithm which reduces the problem to a SAT solver and is optimized for performance.
We evaluate our methodology on three applications: a web administrator, a simple firewall, and a conference management system. We compared our tool with Z3 [14], currently the most advanced general-purpose SMT solver with (limited) support for quantifiers. In general, our tool performs better than Z3 in several examples for checking the validity of verification conditions of SmpSL programs. However, our tool and Z3 have complementary advantages: Z3 does well for unsatisfiable instances while our tool performs better on satisfiable instances. We performed large experiments with custom-made blown up versions of the web administrator and the firewall examples, which suggest that our tool scales well. Moreover, we tested the scalability of our approach by comparing of our underlying FO solver with three solvers on a set of benchmarks we assembled inspired by combinatorial problems. The solvers we tested against are Z3, the SMT solver CVC4 [3], and the model checker Nitpick [7]. Our solver outperformed each of these solvers on some of the benchmarks.

2 Running Example

We introduce our approach on the example of a simple web service. The example is a translation from PHP with embedded SQL commands into SmpSL of code excerpts from the Panda web-administrator. The web service provides several services implemented in dedicated functions for subscribing a user to a newsletter, deleting a newsletter, making a user an admin of a newsletter, sending emails to all subscribed users of a newsletter, etc. We illustrate our verification methodology by exposing an error in the Panda web-administrator.

The verification methodology we envision in this paper consists of (1) maintaining database invariants and (2) verifying a contract specification for each function of the web service.

The database contains several tables including $NS = \text{NewsletterSubscription}$ with attributes $\text{nwl}$, $\text{user}$, $\text{subscribed}$ and $\text{code}$. The database is a structure whose universe is partitioned into three sets: $\text{dom}^U$, $\text{bool}^B$, and $\text{codes}^B$. The attributes $\text{nwl}$ and $\text{user}$ range over the finite set $\text{dom}^U$, the attribute $\text{subscribed}$ ranges over $\text{bool}^B = \{\text{true}, \text{false}\}$, and the attribute $\text{code}$ ranges over the fixed finite set $\text{codes}^B$. The superscripts in $\text{dom}^U$, $\text{bool}^B$, and $\text{codes}^B$ serve to indicate that the domain $\text{dom}^U$ is unbounded, while the Boolean domain and the domain of codes are bounded (i.e. of fixed finite size). When $s = \text{true}$, $(n, u, s, c) \in NS$ signifies that the user $u$ is subscribed to the newsletter $n$. The process of being (un)subscribed from/to a newsletter requires an intermediary confirmation step in which the confirm code $c$ plays a role.

Figure 1 provides the functions $\text{subscribe}$, $\text{unsubscribe}$, and $\text{confirm}$ translated manually into SmpSL. The comments in quotations // "..." originate from the PHP source code. The intended use of these functions is as follows: To subscribe a user $u$ to a newsletter $n$, the function $\text{subscribe}$ is called with inputs $n$ and $u$ (e.g. by a web interface operated by an admin or by the user). $\text{subscribe}$ stores the tuple $\langle n, u, false, new\_code \rangle$ in $NS$, where $new\_code$ is a confirmation code which does not occur in the database, and an email containing a confirmation URL is sent to the user $u$. Visiting the URL triggers a call to $\text{confirm}$ with input $new\_code$, which subscribes $u$ to $n$ by replacing the tuple $\langle n, u, false, new\_code \rangle$ of $NS$ to with $\langle n, u, true, nil \rangle$. For $\text{unsubscribe}$ the process is similar, and crucially, $\text{unsubscribe}$ uses the same $\text{confirm}$ function. $\text{confirm}$ decides between subscribe and unsubscribe according to whether $n$ is currently subscribed to $u$. The $\text{CHOOSE}$ command selects one row non-deterministically. The database preserves the invariant

$$Inv = \forall x, y, \forall s_1, s_2, \forall c_1, c_2. \left( s_1 = s_2 \land c_1 = c_2 \right) \lor \bigvee_{i=1,2} \neg NS(x, y, s_i, c_i)$$

(1)
subscribe(n,u):
  A = SELECT * FROM NS WHERE user = u AND nwl = n;
  if (A != empty) exit; // "This address is already registered to this newsletter."
  INSERT (n,u,false,new_code) INTO NS;
  // Send confirmation email to u

unsubscribe(n,u):
  A = SELECT * FROM NS WHERE user = u AND nwl = n;
  if (A = empty) exit; // "This address is not registered to this newsletter."
  UPDATE NS SET code = new_code WHERE user = u AND nwl = n
  // Send confirmation email to u

confirm(cd):
  A = SELECT subscribe FROM NS WHERE code = cd;
  if (A = empty) exit; //"No such code"
  s1 = CHOOSE A;
  if (s1 = false)
    UPDATE NS SET subscribed = true, code = nil WHERE code = cd
  else DELETE FROM NS WHERE code = cd;

Figure 1 Running Example: SmpSL code.

Inv says that the pair (n, u) of newsletter and user is a key of the relation NS. The subscripts of the quantifiers denote the domains over which the quantified variables range. In our verification methodology we add invariants as additional conjuncts to the pre- and post-conditions of every function. In this way invariants strengthen the pre-conditions and can be used to prove the post-conditions of the functions. On the other hand, the post-conditions require to re-establish the validity of the invariants.

Figure 2 provides pre- and post-conditions pre₇ and post₇ for each of the three functions f. The relation names d, b, and c are interpreted as the sets domU, boolB, and codesB, respectively. Proving correctness amounts to proving the correctness of each of the Hoare triples \{pre₇ ∧ Inv\} f \{post₇ ∧ Inv\}. Each Hoare triple specifies a contract: after every execution of f, the condition post₇ ∧ Inv should be satisfied if pre₇ ∧ Inv was satisfied before executing f. pre_subscribe and pre_unsubscribe express that new_code is an unused non-nil code and that NSgh is equal to NS. NSgh is a ghost table, used in the post-conditions to relate the state before the execution of the function to the state after the execution. NSgh does not occur in the functions and is not modified. post_subscribe and post_unsubscribe express that NS is obtained from NSgh by inserting or updating a row satisfying user = u AND nwl = n whenever the exit command is not executed. The intended behavior of confirm depends on which function created cd. pre_confirm introduces a Boolean ghost variable subgh whose value is true (respectively false) if cd was generated as a new code in subscribe (respectively unsubscribe). subgh does not occur in confirm. post_confirm express that, when subgh is true, NS is obtained from NS by toggling the value of the column subscribed from false to true in the NSgh row whose confirm code is cd; when subgh is false, NS is obtained from NSgh by deleting the row with confirm code cd.
Let us now describe the error which prevents confirm from satisfying its specification. Consider the following scenario. First, subscribe is called and then unsubscribe, both with the same input n and u. Two confirm codes are created: \(c_u\) by subscribe and \(c_c\) by unsubscribe. At this point, \(NS\) contains a single row for the newsletter \(n\) and user \(u\) namely \((n, u, false, c_u)\). The user receives two confirmation emails containing the codes \(c_u\) and \(c_c\). Clicking on the confirmation URL for \(c_u\) (i.e. running confirm\((c_u)\)) has no effect since \(c_u\) does not occur in the database. However, clicking on the confirmation URL for \(c_c\) results in subscribing \(u\) to \(n\). This is an error, since confirming a code created in unsubscribe should not lead to a subscription.

Our tool automatically checks whether the program satisfies its specification. If not, the programmer or verification engineer may try to refine the specification to adhere more closely to the intended behavior (e.g. by adding an invariant). In this case, the program is in fact incorrect, so no meaningful correct specification can be written for it.

In Section 3.3 we describe a weakest-precondition calculus \(wp[f]\) which allows us to automatically derive the weakest precondition for a post-condition with regard to a SmpSL program. For our example functions \(f\), \(wp[f]\) allows us to automatically derive \(wp[f]post\). The basic property of the weakest precondition is that \(wp[f]post\) held immediately at the start of the execution. It then remains to show that the pre-condition \(pre\) implies \(wp[f]post\). This amounts to checking the validity of the verification conditions \(VC = pre \rightarrow wp[f]post\).

Our reasoner for \(FO^2\) sentences is the back-end for our verification tool. The specification in this example is all in \(FO^2_{BD}\). The weakest precondition of a SmpSL program applied to a \(FO^2_{BD}\) sentence gives again a \(FO^2_{BD}\) sentence. Hence \(VC\) are all in \(FO^2_{BD}\). Automatically deciding the validity of \(FO^2_{BD}\) sentences using our \(FO^2\) decision procedure is described in Section 4. Recall that \(codes^B\) is of fixed finite size. Here \(|codes^B| = 3\) is sufficient to detect the error. Observe that the same confirm code may be reused once it is replaced with \(nil\) in confirm, so the size of the database is unbounded. The size of \(codes^B\) must be chosen manually when applying our automatic tool.

A simple way to correct the error in confirm is by adding \(sub_{gh}\) as a second argument of confirm and replacing if \((s_1 = false)\) \(\cdots\) with if \((sub_{gh} = false)\) \(\cdots\). Since \(s_1\) is no longer used, the \(CHOICE\) command can be deleted. The value of \(sub_{gh}\) received by confirm is set correctly by subscribe and unsubscribe. With these changes, the error is fixed and confirm satisfies its specification. In the scenario from above, the call to confirm with \(c_u\) and \(sub_{gh} = true\) leaves the database unchanged, while the call to confirm with \(c_u\) and \(sub_{gh} = false\) deletes the row \((n, u, false, c_u)\).

### 3 Verification of SmpSL Programs

Here we introduce our programming language and our verification methodology. We introduce the SQL fragment SmpSQL in Section 3.1 and the scripting language SmpSL in Section 3.2. In Section 3.3 we explain the weakest precondition transformer of SmpSL, and we show how discharging verification conditions of \(FO^2_{BD}\) specification reduces to reasoning in \(FO^2\).

#### 3.1 The SQL fragment SmpSQL

##### 3.1.1 Data model of SmpSQL

The data model of SmpSQL is based on the presentation of the relational model in Chapter 3.1 of [1]. We assume finite sets of \(dom_1^B, \ldots, dom_n^B\) called the bounded domains and an infinite
The relation schema $R$ is a relation name and a finite sequence of attributes. The attributes are the names of the columns of the table. The arity $\text{ar}(R)$ of a relation schema $R$ is the number of its attributes. A database schema is a non-empty finite set of tables.

A database instance $I$ of a database schema $R$ is a many-sorted structure with finite domains $\text{dom}_U$ called the unbounded domain. The domains are disjoint. We assume three disjoint countably infinite sets: the set of attributes $\text{att}$, the set of relation names $\text{relnames}$, and the set of variables $\text{SQLvars}$. We assume a function $\text{sort} : \text{att} \rightarrow \{\text{dom}_U, \text{dom}_B^1, \ldots, \text{dom}_B^B\}$.

Figure 2 Running Example: Pre- and post-conditions. $g$ is either $\text{subscribe}$ or $\text{unsubscribe}$.

set $\text{dom}_U$ called the unbounded domain. The domains are disjoint. We assume three disjoint countably infinite sets: the set of attributes $\text{att}$, the set of relation names $\text{relnames}$, and the set of variables $\text{SQLvars}$. We assume a function $\text{sort} : \text{att} \rightarrow \{\text{dom}_U, \text{dom}_B^1, \ldots, \text{dom}_B^B\}$. A table or a relation schema is a relation name and a finite sequence of attributes. The attributes are the names of the columns of the table. The arity $\text{ar}(R)$ of a relation schema $R$ is the number of its attributes. A database schema is a non-empty finite set of tables.

A database instance $I$ of a database schema $R$ is a many-sorted structure with finite domains $\text{dom}_0 \subseteq \text{dom}_U$ and $\text{dom}_j = \text{dom}_B^j$ for $1 \leq j \leq s$. We denote by $\text{sort}_I$ the function obtained from $\text{sort}$ by setting $\text{sort}_I(\text{att}) = \text{dom}_0$ whenever $\text{sort}(\text{att}) = \text{dom}_U$. The relation schema $R = (\text{relname}, att_1, \ldots, att_s)$ is interpreted in $I$ as a relation $R^I \subseteq \text{sort}_I(\text{att}_1) \times \cdots \times \text{sort}_I(\text{att}_s)$. A row is a tuple in a relation $R^I$.

A database schema $R$ is valid for SmpSQL if for all relation schemas $R$ with attributes $att_1, \ldots, att_s$ in $R$, there are at most two attributes $att_j$ for which $\text{sort}(att_j) = \text{dom}_U$. In the sequel we assume that all database schemas are valid. The SmpSQL commands will be allowed to use variables from $\text{SQLvars}$. We denote members of $\text{SQLvars}$ by $p, p_1, \text{etc}$.

### 3.1.2 Queries in SmpSQL

Given a relation schema $R$ and attributes $att_1, \ldots, att_n$ of $R$, the syntax of $\text{SELECT}$ is:

\[
\langle \text{Select} \rangle ::= \text{SELECT} \ att_1, \ldots, att_n \ \text{FROM} \ R \ \text{WHERE} \ \langle \text{Condition} \rangle \\
\langle \text{Condition} \rangle ::= \ att_1, \ldots, att_n \ \text{IN} \ \langle \text{Select} \rangle | \\
\langle \text{Condition} \rangle \ \text{AND} \ \langle \text{Condition} \rangle | \\
\langle \text{Condition} \rangle \ \text{OR} \ \langle \text{Condition} \rangle | \\
\text{NOT} \ \langle \text{Condition} \rangle | \\
att_n = p
\]

where $p$ is a variable and $1 \leq m, a_1, \ldots, a_i, b_1, \ldots, b_j \leq n$. The semantics of $\langle \text{Select} \rangle$ is the set of tuples from the projection of $R$ on $att_1, \ldots, att_n$, which satisfy $\langle \text{Condition} \rangle$. The
condition \( \text{att}_a = p \) indicates that the set of rows of \( R \) in which the attribute \( \text{att}_m \) has value \( p \) is selected. The condition \( \text{att}_{b_1}, \ldots, \text{att}_{b_i} \in (\text{Select}) \) selects the set of rows of \( R \) in which \( \text{att}_{b_1}, \ldots, \text{att}_{b_i} \) are mapped to one of the tuples queried in the nested query \( (\text{Select}) \).

### 3.1.3 Data-manipulating commands in SmpSQL

SmpSQL supports the three primitive commands INSERT, UPDATE, and DELETE.

Let \( R \) be a relation schema with attributes \( \text{att}_1, \ldots, \text{att}_n \). Let \( p, p_1, \ldots, p_n \) be variables from \( \text{SQLvars} \). The syntax of the primitive commands is:

\[
\begin{align*}
\langle \text{Insert} \rangle & ::= \text{INSERT} (p_1, \ldots, p_n) \text{ INTO } R \\
\langle \text{Update} \rangle & ::= \text{UPDATE } R \text{ SET } \text{att}_m = p \text{ WHERE } (\text{Condition}) \\
\langle \text{Delete} \rangle & ::= \text{DELETE FROM } R \text{ WHERE } (\text{Condition})
\end{align*}
\]

The semantics of INSERT, UPDATE and DELETE is given in the natural way. We allow update commands which set several attributes simultaneously. We assume that the data manipulating commands are used in a domain-correctness fashion, i.e. INSERT and UPDATE may only assign values from \( \text{sort}(\text{att}_k) \) to any attribute \( \text{att}_k \).

### 3.2 The script language SmpSL

#### 3.2.1 Data model of SmpSL

The data model of SmpSL extends that of SmpSQL with constant names and additional relation schemas. We assume a countably infinite set of constant names \( \text{connames} \), which is disjoint from \( \text{att}, \text{dom}_U^1, \ldots, \text{dom}_s^1, \text{renames} \) but contains \( \text{SQLvars} \).

A \text{state schema} is a database schema \( R \) expanded with a tuple of constant names \( \text{const} \). A \text{state} interprets a state schema. It consists of a database instance \( I \) expanded with a tuple of universe elements \( \text{const} \) interpreting \( \text{const} \). In programs, the constant names play the role of local variables, domain constants (e.g. \text{true} and \text{true}) and of inputs to the program\(^1\).

#### 3.2.2 SmpSL programs

The syntax of SmpSL is given by

\[
\begin{align*}
\langle \text{Program} \rangle & ::= (\text{Command}) \mid (\text{Program}) : (\text{Command}) \\
\langle \text{Command} \rangle & ::= (\text{Insert}) \mid (\text{Update}) \mid (\text{Delete}) \mid R = (\text{Select}) \mid \overline{d} = \text{CHOOSE } R \mid \\
& \quad \text{if } (\text{cond}) (\text{Program}) \mid \text{if } (\text{cond}) \text{exit} \mid \\
& \quad \text{if } (\text{cond}) (\text{Program}) \text{ else } (\text{Program})
\end{align*}
\]

Every data-manipulating command \( C \) of SmpSQL is a SmpSL command. The semantics of \( C \) in SmpSQL is the same as in SmpSQL, with the caveat that the variables receive their values from their interpretations (as constant names) in the state, and \( C \) is only legal if all the variables of \( C \) indeed appear in the state schema as constant names.

The command \( R = (\text{Select}) \) assigns the result of a SmpSQL query to a relation schema \( R \in R \) whose arity and attribute sorts match the select query. Executing the command in a state \( (I, \overline{\text{const}}) \) sets \( R^2 \) to the relation selected by \( S \), leaving the interpretation of all other

\(^1\) We deviate from [1] in the treatment of constants in that we do not assume that constant names are always interpreted as \textit{distinct} members of \( \text{dom}_U^1 \). This is so since several program variables or inputs can have the same value.
Given a relation schema $R \in \mathcal{R}$ with attributes $att_1, \ldots, att_n$ and a tuple $\vec{d} = (d_1, \ldots, d_n)$ of constant names from $\text{const}$, $\vec{d} = \text{CHOOSE} \ R$ is a SmpSL command. If $R^\emptyset$ is empty, the command has no effect. If $R^\emptyset$ is not empty, executing this command sets $(d_1^1, \ldots, d_n^1)$ to the value of a non-deterministically selected row from $R^\emptyset$.

The branching commands have the natural semantics. Two types of branching conditions $\text{cond}$ are allowed: $(R = \emptyset)$ and $(R \neq \emptyset)$, which check whether $R^\emptyset$ is the empty set, and $(c_1 = c_2)$ and $(c_1 \neq c_2)$, which check whether $c_1^2 = c_2^2$.

### 3.3 Verification of SmpSL programs

#### 3.3.1 SQL and FO

It is well-established that a core part of SQL is captured by FO by Codd’s classical theorem relating the expressive power of relational algebra to relational calculus. While SQL goes beyond FO in several aspects, such as aggregation, grouping, and arithmetic operations (see [27]), these aspects are not allowed in SmpSQL. Hence, FO is especially suited for reasoning about SmpSQL and SmpSL.

The notions of state schema and state fit naturally in the syntax and semantics of FO. In the sequel, a **vocabulary** is a tuple of relation names and constant names. Every state schema $R$ is a vocabulary. A state $(I, \text{const})^R$ interpreting a state schema $R$ and a tuple of constant names $\text{const}$ is an $(R, \text{const})$-structure.

#### 3.3.2 Hoare verification of SmpSL programs and weakest precondition

Hoare logic is a standard program verification methodology [23]. Let $P$ be a SmpSL program and let $\varphi_\text{pre}$ and $\varphi_\text{post}$ be FO-sentences. A **Hoare triple** is of the form $(\varphi_\text{pre}) \ P \ (\varphi_\text{post})$. A Hoare triple is a contract relating the state before the program is run with the state afterward. The goal of the verification process is to prove that the contract is correct.

Our method of proving that a Hoare triple is valid reduces the problem to that of finite satisfiability of a FO-sentence. We compute the weakest precondition $wp[P] \varphi_\text{post}$ with respect to the program $P$. The weakest precondition transformer was introduced in Dijkstra’s classic paper [17], c.f. [24]. Let $A_P$ denote the state after executing $P$ on the initial state $A$. The main property of the weakest precondition is: $A_P \models \varphi_\text{post}$ iff $A \models wp[P] \varphi_\text{post}$.

Using $\text{wp}[\cdot]$ we can rephrase the problem of whether the Hoare triple $(\varphi_\text{pre}) \ P \ (\varphi_\text{post})$ is valid in terms of FO reasoning on finite structures: Is the FO-sentence $\varphi_\text{pre} \rightarrow wp[P] \varphi_\text{post}$ a tautology? Equivalently, is the FO-sentence $\varphi_\text{pre} \land \neg wp[P] \varphi_\text{post}$ unsatisfiable? Section 3.3.3 discusses the resulting FO reasoning task.

We describe the computation of the weakest preconditions inductively for SmpSQL and SmpSL. The weakest precondition for SmpSQL is given in Fig. 3, and for SmpSL in Fig. 4. For SmpSQL conditions, $[\cdot]^{R^\emptyset}$ is a formula with $n$ free first-order variables $v_1, \ldots, v_n$ for a conditional expression in the context of relation schema $R$ of arity $n$. $\text{SELECT} \cdots \text{FROM} \ R \cdots$ is also a formula with free variables $v_1, \ldots, v_n$ describing the rows selected by the SELECT query. The rules $wp[s][Q]$ transform a (closed) formula $Q$, which is a postcondition of the command $s$, into a (closed) formula expressing the weakest precondition. The notation $\psi[t/v]$ indicates substitution of all free occurrences of the variable $v$ in $\psi$ by the term $t$.

The notation $\psi[\theta(\alpha_1, \ldots, \alpha_n)/R(\alpha_1, \ldots, \alpha_n)]$ indicates that any atomic sub-formula of $\psi$ of the form $R(\alpha_1, \ldots, \alpha_n)$ (for any $\alpha_1, \ldots, \alpha_n$) is replaced by $\theta(\alpha_1, \ldots, \alpha_n)$ (with the same constant names unchanged. The variables in the query receive their values from their interpretations in the state, and for the command to be legal, all variables in the query must appear in the state schema as constant names.

Using $wp[\cdot]$, we can rephrase the problem of whether the Hoare triple $(\varphi_\text{pre}) \ P \ (\varphi_\text{post})$ is valid in terms of FO reasoning on finite structures: Is the FO-sentence $\varphi_\text{pre} \rightarrow wp[P] \varphi_\text{post}$ a tautology? Equivalently, is the FO-sentence $\varphi_\text{pre} \land \neg wp[P] \varphi_\text{post}$ unsatisfiable? Section 3.3.3 discusses the resulting FO reasoning task.

We describe the computation of the weakest preconditions inductively for SmpSQL and SmpSL. The weakest precondition for SmpSQL is given in Fig. 3, and for SmpSL in Fig. 4. For SmpSQL conditions, $[\cdot]^{R^\emptyset}$ is a formula with $n$ free first-order variables $v_1, \ldots, v_n$ for a conditional expression in the context of relation schema $R$ of arity $n$. $\text{SELECT} \cdots \text{FROM} \ R \cdots$ is also a formula with free variables $v_1, \ldots, v_n$ describing the rows selected by the SELECT query. The rules $wp[s][Q]$ transform a (closed) formula $Q$, which is a postcondition of the command $s$, into a (closed) formula expressing the weakest precondition. The notation $\psi[t/v]$ indicates substitution of all free occurrences of the variable $v$ in $\psi$ by the term $t$.

The notation $\psi[\theta(\alpha_1, \ldots, \alpha_n)/R(\alpha_1, \ldots, \alpha_n)]$ indicates that any atomic sub-formula of $\psi$ of the form $R(\alpha_1, \ldots, \alpha_n)$ (for any $\alpha_1, \ldots, \alpha_n$) is replaced by $\theta(\alpha_1, \ldots, \alpha_n)$ (with the same constant names unchanged. The variables in the query receive their values from their interpretations in the state, and for the command to be legal, all variables in the query must appear in the state schema as constant names.

Given a relation schema $R \in \mathcal{R}$ with attributes $att_1, \ldots, att_n$ and a tuple $\vec{d} = (d_1, \ldots, d_n)$ of constant names from $\text{const}$, $\vec{d} = \text{CHOOSE} \ R$ is a SmpSL command. If $R^\emptyset$ is empty, the command has no effect. If $R^\emptyset$ is not empty, executing this command sets $(d_1^1, \ldots, d_n^1)$ to the value of a non-deterministically selected row from $R^\emptyset$.

The branching commands have the natural semantics. Two types of branching conditions $\text{cond}$ are allowed: $(R = \emptyset)$ and $(R \neq \emptyset)$, which check whether $R^\emptyset$ is the empty set, and $(c_1 = c_2)$ and $(c_1 \neq c_2)$, which check whether $c_1^2 = c_2^2$.
As discussed in Section 3.3.2, using the weakest precondition, the problem of verifying Hoare variables

\[
\phi \preceq \psi
\]

Theorem 1. Let \( \{ \varphi_{\text{pre}} \} P \{ \varphi_{\text{post}} \} \) be a Hoare triple such that both \( \varphi_{\text{pre}} \) and \( \varphi_{\text{post}} \) belong to \( \text{FO}^2_{\text{BD}} \). The problem of deciding whether \( \{ \varphi_{\text{pre}} \} P \{ \varphi_{\text{post}} \} \) is valid is decidable.

**Figure 3** Rules for weakest precondition for SmpSQL basic commands. We denote by \( R \) a relation schema with attributes \( \langle \text{att}_1, \ldots, \text{att}_n \rangle \). We write \( \alpha_i \) for \( \alpha_i \) if \( i \neq j \), and for \( v_i \) if \( i = j \). We denote \( \bar{v} = (v_1, \ldots, v_n) \), \( \bar{\alpha} = (\alpha_1, \ldots, \alpha_n) \), and \( \bar{\alpha}^j = (\alpha_1^j, \ldots, \alpha_n^j) \). Note that each of the last three rows \( Q[\text{expr}(\alpha)/R(\bar{\alpha})] \) substitutes every occurrence of \( R \) with an updated expression \( \text{expr} \).

\[\begin{align*}
\text{wp[INSERT} \ (c_1, \ldots, c_k) \ \text{INTO R]Q} & \equiv \ Q[R(\bar{\alpha}) \lor \bigwedge_{i=1}^n \alpha_i = c_i \ / \ R(\bar{\alpha})] \\
\text{wp[DELETE} \ \text{FROM R WHERE cond]Q} & \equiv \ Q[R(\bar{\alpha}) \land \neg[\text{cond}] R[\alpha_i/v_i : 1 \leq i \leq n] / R(\bar{\alpha})] \\
\text{wp[UPDATE R SET att} _j = c \ \text{WHERE cond]Q} & \equiv \ Q[R(\bar{\alpha}) \land \neg[\text{cond}] R[\alpha_i/v_i : 1 \leq i \leq n] \lor \exists v_j R(\bar{\alpha}^j) \land [\text{cond}] R[\alpha_i^j/v_i : 1 \leq i \leq n] \land \alpha_j = c / R(\bar{\alpha})]
\end{align*}\]

\(\alpha_1, \ldots, \alpha_n\). The formula \( \theta(v_1, \ldots, v_n) \) has \( n \) free variables, and \( \theta(\alpha_1, \ldots, \alpha_n) \) is obtained by substituting each \( v_i \) into \( \alpha_i \). The \( \alpha_i \) may be variables or constant names.

The weakest precondition of a SmpSL program is obtained by applying the weakest precondition of its commands.

### 3.3.3 The specification logic \( \text{FO}^2_{\text{BD}} \) and decidability of verification

As discussed in Section 3.3.2, using the weakest precondition, the problem of verifying Hoare triples can be reduced to the problem of checking satisfiability of a FO-sentence by a finite structure. While this problem is not decidable in general by Trakhtenbrot’s theorem, it is decidable for a fragment of FO we denote \( \text{FO}^2_{\text{BD}} \), which extends the classical two-variable fragment \( \text{FO}^2 \). The logic \( \text{FO}^2 \) is the set of all FO formulas which use only variables the

\[\begin{align*}
\text{wp[INSERT} \ (c_1, \ldots, c_k) \ \text{INTO R]Q} & \equiv \ Q[R(\bar{\alpha}) \lor \bigwedge_{i=1}^n \alpha_i = c_i \ / \ R(\bar{\alpha})] \\
\text{wp[DELETE} \ \text{FROM R WHERE cond]Q} & \equiv \ Q[R(\bar{\alpha}) \land \neg[\text{cond}] R[\alpha_i/v_i : 1 \leq i \leq n] / R(\bar{\alpha})] \\
\text{wp[UPDATE R SET att} _j = c \ \text{WHERE cond]Q} & \equiv \ Q[R(\bar{\alpha}) \land \neg[\text{cond}] R[\alpha_i/v_i : 1 \leq i \leq n] \lor \exists v_j R(\bar{\alpha}^j) \land [\text{cond}] R[\alpha_i^j/v_i : 1 \leq i \leq n] \land \alpha_j = c / R(\bar{\alpha})]
\end{align*}\]

\(\alpha_1, \ldots, \alpha_n\). The formula \( \theta(v_1, \ldots, v_n) \) has \( n \) free variables, and \( \theta(\alpha_1, \ldots, \alpha_n) \) is obtained by substituting each \( v_i \) into \( \alpha_i \). The \( \alpha_i \) may be variables or constant names.

The weakest precondition of a SmpSL program is obtained by applying the weakest precondition of its commands.

As discussed in Section 3.3.2, using the weakest precondition, the problem of verifying Hoare triples can be reduced to the problem of checking satisfiability of a FO-sentence by a finite structure. While this problem is not decidable in general by Trakhtenbrot’s theorem, it is decidable for a fragment of FO we denote \( \text{FO}^2_{\text{BD}} \), which extends the classical two-variable fragment \( \text{FO}^2 \). The logic \( \text{FO}^2 \) is the set of all FO formulas which use only variables the variables \( x \) and \( y \). The vocabularies of \( \text{FO}^2 \)-sentences are not allowed function names, only relation and constant names. Note \( \text{FO}^2 \) cannot express that a relation name is interpreted as a function. \( \text{FO}^2 \) contains the equality symbol \( = \). \( \text{FO}^2_{\text{BD}} \) extends \( \text{FO}^2 \) by allowing quantification on an unbounded number of variables, under the restriction that all variables besides from \( x \) and \( y \) range over the bounded domains only.

\( \text{FO}^2_{\text{BD}} \) is the language of our invariants and pre- and postconditions, see Eq. (1) and Fig. 2 in Section 2. An important property of \( \text{FO}^2_{\text{BD}} \) is that it is essentially closed under taking weakest precondition according to Figs. 3 and 4 since all relation schemas in a (valid) database schema have at most 2 attributes whose sort is \( \text{dom}^U \). We reduce the task of reasoning over \( \text{FO}^2_{\text{BD}} \) to reasoning over \( \text{FO}^2 \).

**Theorem 1.** Let \( \{ \varphi_{\text{pre}} \} P \{ \varphi_{\text{post}} \} \) be a Hoare triple such that both \( \varphi_{\text{pre}} \) and \( \varphi_{\text{post}} \) belong to \( \text{FO}^2_{\text{BD}} \). The problem of deciding whether \( \{ \varphi_{\text{pre}} \} P \{ \varphi_{\text{post}} \} \) is valid is decidable.
On the Automated Verification of Web Applications with Embedded SQL

An early precursor for the use of a SAT solver for finite satisfiability is [28].

By Section 3.3.2, the bounded model property of FO\(^2\) is reduced to the decidability of validity for FO\(^2\). We compute an asymptotically-tight exponential bound \(\text{bnd}(\phi)\), and based on it gave a NEXPTIME algorithm. The algorithm non-deterministically guesses \(t \leq \text{bnd}(\phi)\) and a \(\tau\)-structure \(\mathcal{A}\) with universe of size \(t\), then checks whether \(\mathcal{A}\) satisfies \(\phi\), and answers accordingly.

4 FO\(^2\) Reasoning

4.1 The bounded model property of FO\(^2\)

Section 4 is devoted to our algorithm for FO\(^2\) finite satisfiability. The main ingredient for this algorithm is the bounded model property, which guarantees that if an FO\(^2\) sentence is satisfiable by any \(\tau\)-structure — finite or infinite — it is satisfiable by a finite \(\tau\)-structure whose cardinality is bounded by a computable function of \(\phi\). Grädel, Kolaitis and Vardi [21] computed an asymptotically-tight exponential bound \(\text{bnd}(\phi)\), and based on it gave a NEXPTIME algorithm. The algorithm non-deterministically guesses \(t \leq \text{bnd}(\phi)\) and a \(\tau\)-structure \(\mathcal{A}\) with universe of size \(t\), then checks whether \(\mathcal{A}\) satisfies \(\phi\), and answers accordingly.

4.2 Finite satisfiability using a SAT solver

Our algorithm for FO\(^2\) finite satisfiability reduces the problem of finding a satisfying model of cardinality bounded by \(\text{bnd}\) to the satisfiability of a propositional Boolean formula in Conjunctive Normal Form CNF, which is then solved using a SAT solver. The bound in [21] is given for formulas in Scott Normal Form (SNF) only. We use a refinement of SNF we call Skolemized Scott Normal Form (SSNF). The CNF formula we generate encodes the semantics of the sentence \(\psi\) on a structure whose universe cardinality is bounded by \(\text{bnd}\). An early precursor for the use of a SAT solver for finite satisfiability is [28].
4.2.1 Skolemized Scott Normal Form

An FO$^2$-sentence is in Skolemized Scott Normal Form if it is of the form

$$\forall x \forall y \left( \alpha(x,y) \land \bigwedge_{i=1}^{m} F_i(x,y) \rightarrow \beta_i(x,y) \right) \land \bigwedge_{i=1}^{m} \forall x \exists y F_i(x,y)$$

(2)

where $\alpha$ and $\beta_i$, $i = 1, \ldots, m$, are quantifier-free formulas which do not contain any $F_j$, $j \neq i$. Note that $F_i$ are relation names.

**Proposition 2.** Let $\tau$ be a vocabulary and $\phi$ be a FO$^2(\tau)$-sentence. There are polynomial-time computable vocabulary $\sigma \supseteq \tau$ and FO$^2(\sigma)$-sentence $\psi$ such that

(a) $\psi$ is in SSNF;
(b) The set of cardinalities of the models of $\phi$ is equal to the corresponding set for $\psi$; and
(c) The size of $\psi$ is linear in the size of $\phi$.

Proposition 2 follows from the discussion before Proposition 3.1 in [21], by applying an additional normalization step converting SNF sentences to SSNF sentences.$^{23}$

4.2.2 The CNF formula

Given the sentence $\psi$ in SSNF from Eq. (2) and a bound $\text{bnd}(\psi)$, we build a CNF propositional Boolean formula $C_\psi$ which is satisfiable iff $\psi$ is satisfiable. The formula $C_\psi$ will serve as the input to the SAT solver. First we construct a related CNF formula $B_\psi$. The crucial property of $B_\psi$ is that it is satisfiable iff $\psi$ is satisfiable by a model of cardinality exactly $\text{bnd}(\psi)$.

It is convenient to assume $\psi$ does not contain constants. If $\psi$ did contain constants $c$, they could be replaced by unary relations $U_c$ of size 1.

We start by introducing the variables and clauses which guarantee that $B_\psi$ encodes a structure with the universe $\{1, \ldots, \text{bnd}(\psi)\}$. Later, we will add clauses to guarantee that this structure satisfies $\psi$. For every unary relation name $U$ in $\psi$ and $\ell_1 \in \{1, \ldots, \text{bnd}(\psi)\}$, let $v_{U,\ell_1}$ be a propositional variable. For every binary relation name $R$ in $\psi$ and $\ell_1, \ell_2 \in \{1, \ldots, \text{bnd}(\psi)\}$, let $v_{R,\ell_1,\ell_2}$ be a propositional variable. The variables $v_{U,\ell_1}$ and $v_{R,\ell_1,\ell_2}$ encode the interpretations of the unary and binary relation names $U$ and $R$ in the straightforward way (defined precisely below). Let $V_\psi$ be the set of all variables $v_{U,\ell_1}$ and $v_{R,\ell_1,\ell_2}$.

Given an assignment $S$ to the variables of $V_\psi$ we define the unique structure $A_S$ as follows:

1. The universe $A_S$ of $A_S$ is $\{1, \ldots, \text{bnd}(\psi)\}$;
2. An unary relation name $U$ is interpreted as the set $\{\ell_1 \in A_S \mid S(v_{U,\ell_1}) = \text{True}\}$;
3. A binary relation name $R$ is interpreted as the set $\{(\ell_1, \ell_2) \in A_S^2 \mid S(v_{R,\ell_1,\ell_2}) = \text{True}\}$.

For every structure $A$ with universe $\{1, \ldots, \text{bnd}(\psi)\}$, there is $S$ such that $A = A_S$.

Before defining $B_\psi$ precisely we can already state the crucial property of $B_\psi$:

**Proposition 3.** $\psi$ is satisfiable by a structure with universe $\{1, \ldots, \text{bnd}(\psi)\}$ iff $B_\psi$ is satisfiable.

The formula $B_\psi$ is the conjunction of $B^{\text{un}}$, $B^{\text{v2}}$, and $B^{\text{v}\forall}$, described in the following.

---

2 The word Skolemized is used in reference to the standard Skolemization process of eliminating existential quantifiers by introducing fresh function names called Skolem functions. In our case, since function names are not allowed in our fragment, we introduce the relation names $F_i$, to which we refer as Skolem relations. Moreover, we cannot eliminate the existential quantifiers entirely, but only simplify the formulas in their scope to the atoms $F_i(x,y)$.

3 The linear size of $\psi$ uses our relation symbols have arity at most 2 to get rid of a log factor in [21].
The equality symbol. The equality symbol requires special attention. Let

\[ B_{eq} = \bigwedge_{1 \leq \ell \neq \ell' \leq m} (\neg v_{=,\ell,\ell'}) \land \bigwedge_{1 \leq \ell \leq m} v_{=,\ell,\ell} \]

\( B_{eq} \) enforces that the equality symbol is interpreted correctly as the equality relation on universe elements.

The \( \forall\exists \)-conjectures. For every conjunct \( \forall x \exists y F_i(x, y) \) and \( 1 \leq \ell_1 \leq \text{bnd}(\psi) \), let \( B_{\psi,1}^{\forall\exists} \) be the clause \( \bigwedge_{\ell_2=1}^{\text{bnd}(\psi)} v_{F_i,\ell_1,\ell_2} \). This clause says that there is at least one universe element \( \ell_2 \) such that \( A_S \models F(\ell_1, \ell_2) \). Let

\[ B_{\psi,1}^{\forall\exists} = \bigwedge_{1 \leq i \leq m} \bigwedge_{1 \leq \ell_1 \leq \text{bnd}(\psi)} B_{\psi,1}^{\forall\exists,\ell_1} \]

For every truth-value assignment \( S \) to \( \psi \), \( A_S \) satisfies \( \bigwedge_{i=1}^{m} \forall x \exists y F_i(x, y) \) iff \( S \) satisfies \( B_{\psi,1}^{\forall\exists} \).

The \( \forall\forall \)-conjunct. Let \( \forall x \forall y \alpha' \) be the unique \( \forall\forall \)-conjunct of \( \psi \). For every \( 1 \leq \ell_1, \ell_2 \leq \text{bnd}(\psi) \), let \( \alpha_{\ell_1,\ell_2}' \) denote the propositional formula obtained from the quantifier-free FO\(^2\) formula \( \alpha' \) by substituting every atom \( a \) with the corresponding propositional variable for \( \ell_1 \) and \( \ell_2 \) as follows:

\[
\begin{align*}
U(x) & \mapsto v_{U,\ell_1}, & R(y, y) & \mapsto v_{R,\ell_2,\ell_2}, & R(x, x) & \mapsto v_{R,\ell_1,\ell_1} \\
U(y) & \mapsto v_{U,\ell_2}, & R(x, y) & \mapsto v_{R,\ell_1,\ell_2}, & R(y, x) & \mapsto v_{R,\ell_2,\ell_1}
\end{align*}
\]

Let \( B_{\psi,1}^{\forall\forall,\ell_1,\ell_2} \) be the Tseitin transformation of \( \alpha_{\ell_1,\ell_2}' \) to CNF [36], see also [6, Chapter 2]. The Tseitin transformation introduces a linear number of new variables of the form \( v_{\gamma,\ell_1,\ell_2} \), one for each sub-formula \( \gamma \) of \( \alpha_{\ell_1,\ell_2}' \). The transformation guarantees that, for every assignment \( S \) of \( \psi \), \( S \) satisfies \( \alpha_{\ell_1,\ell_2}' \) iff \( S \) can be expanded to satisfy \( B_{\psi,1}^{\forall\forall,\ell_1,\ell_2} \). Let

\[ B_{\psi}^{\forall\forall} = \bigwedge_{1 \leq \ell_1, \ell_2 \leq \text{bnd}(\psi)} B_{\psi,1}^{\forall\forall,\ell_1,\ell_2} \]

Note that [21] guarantees only that \( \text{bnd}(\psi) \) is an upper bound on the cardinality of a satisfying model. Therefore, we build a formula \( C_\psi \) based on \( B_{\psi} \) such that \( C_\psi \) is satisfiable iff \( \psi \) is satisfiable by a structure of cardinality at most \( \text{bnd}(\psi) \). The algorithm for finite satisfiability of a FO\(^2\) sentence \( \phi \) consists of computing the SSNF \( \psi \) of \( \phi \) and returning the result of a satisfiability check using a SAT solver on \( C_\psi \). Both the number of variables and the number of clauses in \( C_{\text{Un}i(\psi)} \) are quadratic in \( \text{bnd}(\psi) \).

## 5 Experimental Results

### 5.1 Details of our tools

The verification condition generator described in Section 3.3.2 is implemented in Java, JFlex and CUP. It is employed to parse the schema, precondition and postcondition and the SmpSL programs. The tool checks that the pre and post conditions are specified in FO\(^2\) and that the scheme is well defined. The SMT-LIB v2 [4] standard language is used as the output format of the verification condition generator. We compare the behavior of our FO\(^2\)-solver with Z3 on the verification condition generator output. The validity of the verification condition can be checked by providing its negation to the SAT solver. If the SAT solver exhibits a satisfying
### Table 1 Running time comparison for example benchmarks.

<table>
<thead>
<tr>
<th></th>
<th>FO$^2$-solver</th>
<th>Z3</th>
<th></th>
<th>FO$^2$-solver</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td>web-subscribe</td>
<td>0.910s</td>
<td>TO</td>
<td>web-subscribe</td>
<td>1.04s</td>
<td>0.02s</td>
</tr>
<tr>
<td>web-unsubscribe</td>
<td>0.741s</td>
<td>OM</td>
<td>web-unsubscribe</td>
<td>1.46s</td>
<td>0.02s</td>
</tr>
<tr>
<td>firewall</td>
<td>0.876s</td>
<td>OM</td>
<td>firewall</td>
<td>18.50s</td>
<td>0.03s</td>
</tr>
<tr>
<td>conf-bid</td>
<td>0.451s</td>
<td>0.015s</td>
<td>conf-bid</td>
<td>TO</td>
<td>0.22s</td>
</tr>
<tr>
<td>conf-assign</td>
<td>0.369s</td>
<td>0.013s</td>
<td>conf-assign</td>
<td>1.196s</td>
<td>0.2s</td>
</tr>
<tr>
<td>conf-display</td>
<td>0.992s</td>
<td>0.016s</td>
<td>conf-display</td>
<td>TO</td>
<td>0.16s</td>
</tr>
</tbody>
</table>

assignment then that serves as counterexample for the correctness of the program. If no satisfying assignment exists, then the generated verification condition is valid, and therefore the program satisfies the assertions. The FO$^2$-solver described in Section 4 is implemented in python and uses pyparsing to parse the SMT-LIB v2 [4] file. The FO$^2$-solver assumes a FO$^2$-sentence as input and uses Lingeling [5] SAT solver as a base Solver.

#### 5.2 Example applications

We tried our approach with a few programs inspired by real-life applications. The first case study is a simplified version of the newsletter functionality included in the PANDA web administrator, that was already discussed and is shown in Fig. 1. The second is an excerpt from a firewall that updates a table of which device is allowed to send packets to which other device. The third is a conference management system with a database of papers, and transactions to manage the review process: reviewers first bid on papers from the pool of submissions, with a policy that a users cannot bid for papers with which they are conflicted. The chair then assigns reviewers to papers by selecting a subset of the bids. At any time, users can ask to display the list of papers, with some details, but the system may hide some confidential information, in particular, users should not be able to see the status of papers before the program is made public. We show how our system detects an information flow bug in which the user might learn that some papers were accepted prematurely by examining the session assignments. This bug is based on a bug we observed in a real system. Each example comes with two specifications, one correct and the other incorrect.

The running time in seconds for all of our examples is reported in Table 1. Timeout is set to 60 minutes and denoted as TO. If the solver reaches out of memory we mark it as OM. On the set of correct examples, Z3 terminates within milliseconds, while FO$^2$-solver takes a few seconds and times out on some of them. On the set of incorrect examples, Z3 fails to answer while our solver performs well. Note that correct examples correspond to unsatisfiable FO$^2$-sentences, while incorrect examples correspond to satisfiable FO$^2$-sentences.

#### 5.3 Examining scalability

**Inflated examples.** In order to evaluate scalability to large examples we inflated our base examples. For instance, while the subscribe example from Table 1 consisted of the subscription

---

4 We omit the confirmation step due to a missing feature in the implementation of the weakest precondition, however the final version of the tool will support the code from Table 1.
of one new email to a mailing-list, Table 2 presents analogous examples based on combining the verification conditions arising from subscribing multiple emails to the mailing-list. The column multiplier details the number of individual subscriptions based on which the formula is constructed. The unsubscribe and firewall example programs are inflated similarly.

We have tested both our FO²-solver and Z3 on large examples and the results reported in Table 2. The high-level of the results is similar to the case of the small examples. On the incorrect examples set Z3 continues to fail mostly due to running out of memory, though it succeeds on the subscribe example. On the correct examples set Z3 continues to outperform the FO²-solver.

Artificial examples. In addition, we constructed a set of artificial benchmarks comprising of several families of FO²-sentences. Each family is parameterized by a number that controls the size of the sentences (roughly corresponding to the number of quantifiers in the sentence). These problems are inspired by combinatorial problems such as graph coloring and paths. We ran experiments using the FO²-solver and three publicly available solvers: Z3, CVC4 (which are SMT solvers), and Nitpick (a model checker). The results are collected in Table 3.

Scalability of FO²-solver. We shall conclude that the FO²-solver, despite being a proof-of-concept prototype implemented in Python with minimal optimizations, handles satisfiable sentences well and also scales well for them. It struggles on unsatisfiable sentences and does not scale well. SMT solvers usually find unsatisfiability proofs much faster, esp. when quantifiers are involved, because they do not have to instantiate all clauses and can terminate as soon as a core set of contradicting clauses is found. This suggests that in practice we may choose to run both FO²-solver and Z3 in parallel and answer according the first result obtained. We also intend to explore how to improve the performance of our solver in the case of incorrect examples. By construction, whenever FO²-solver finds a satisfying model, its size is at most 4 times that of the minimal model. (The constant 4 can be decreased or increased.)

Tools and benchmarks online.
1. FO²Solver:
   http://forsyte.at/people/kotek/fo2-solver/
2. SmpSL Verification Conditions Generator:
   http://forsyte.at/people/kotek/smpsl-verification-conditions-generator/
3. Benchmarks for FO²:
   http://forsyte.at/people/kotek/two-variable-fragment-benchmarks/

6 Discussion

Related work. Verification of database-centric software systems has received increasing attention in the last decade, see for example the recent survey [15]. Below, we explain how our approach differs from the works surveyed in [15]. [15] assumes the services accessing the database to be provided a priori in terms of a local contract given by a pre- and post-condition (see also [31, 26]). The focus of verification then is on the verification of global temporal properties of the system, assuming the local contracts. While the services may be
automatically synthesized in some cases, e.g. [19, 20, 15, 16], they are often implemented manually (e.g. using a scripting language) and the validity of their contracts needs to be verified. This is the verification problem we target in this paper: we show how to prove the correctness of a single service with regard to its pre- and postcondition. The approaches have orthogonal strengths: The works surveyed in [15] use (modulo reductions) the existential fragment of first-order logic ($\exists$FO) to formalize local changes to the database and allow the verification of LTL properties whose atoms are given by $\exists$FO-formulae. In contrast, our approach is limited to the verification of local pre- and post-conditions and system invariants. On the other hand we allow universal quantification in our specifications. It is an interesting direction for future work how to extend our approach to more general temporal properties (e.g. as considered in [15]).

Several papers use variations of FO$^2$ to study verification of programs that manipulate relational information. [8] presents a verification methodology based on FO$^2$, a description logic and a separation logic for analyzing the shapes and content of in-memory data structures. [33] develops a logic similar to FO$^2$ to reason about shapes. In both [8] and [33], the focus is
on analysis of dynamically-allocated memory, and databases are not studied. Furthermore, no tools based on these works are available. Our work draws inspiration from [2], which discusses the verification of evolving graph databases based on a description logic related to $\text{FO}^2$ and a dedicated action language. Our work and [2] exhibit some similarity on a technical level but have a different focus: [2] advocates the use of description logic, while we consider the use of a scripting language with embedded SQL to be advantageous because it does not require to learn new syntax (the identification of an appropriate language and SQL fragment is one of the contributions of this paper); establishing the precise technical relationship between our framework and [2] seems possible but requires additional work to be carried out. Further, the verification method suggested by [2] was not implemented. To our knowledge no description logic solver implements reasoning tasks for the description logic counterpart of $\text{FO}^2$ studied in [2], not even solvers for expressive description logics such as SROIQ. The authors of [2] extended their work to description logics with path constrains in [9].

Verification of script programs with embedded queries has revolved around security, see [18]. However, it seems no other work has been done on such programs.

**Conclusion and future work.** We developed a verification methodology for script programs with access to a relational database via SQL. We isolated a simple but useful fragment SmpSQL of SQL and developed a simple script programming language SmpSL on top of it. We have shown that verifying the correctness of SmpSL programs with respect to specifications in $\text{FO}^2_{BD}$ is decidable. We implemented a solver for the $\text{FO}^2$ finite satisfiability problem, and, based on it, a verification tool for SmpSL programs. Our experimental results are very promising and suggest that our approach has great potential to evolve into a mainstream method for the verification of script programs with embedded SQL statements.

While we believe that many of the SQL statements that appear in real-life programs fall into our fragment SmpSQL it is evident that future tools need to consider all of database usage in real-world programs. In future work, we will explore the extension of SmpSL and SmpSQL. Our next goal is to be able to verify large, real-life script programs such as Moodle [29], whose programming language and SQL statements use e.g. some arithmetic or simple inner joins. To do so, we will adapt our approach from the custom-made syntax of SmpSL to a fragment of PHP. We will both explore decidable logics extending $\text{FO}^2_{BD}$, and investigate verification techniques based on undecidable logics including the use of first-order theorem provers such as Vampire [34, 25] and abstraction techniques which guarantee soundness but may result in spurious errors [12]. For dealing with queries with transitive closure, it is natural to consider fragments of Datalog [10].

A natural extension is to consider global temporal specifications in addition to local contracts. Here the goal is to verify properties of the system which can be expressed in a temporal logic such as Linear Temporal Logic LTL [32, 11]. The approach surveyed in [15], which explore global temporal specifications of services given in terms of local contracts, may be a good basis for studying global temporal specifications in our context.

Another research direction which emerges from the experiments in Section 5 is to explore how to improve the performance of our $\text{FO}^2$ solver on unsatisfiable inputs.

**References**


16:18 On the Automated Verification of Web Applications with Embedded SQL


