On the Automated Verification of Web Applications with Embedded SQL

Shachar Itzhaky[1], Tomer Kotek[*2], Noam Rinetzky[3], Mooly Sagiv[4], Orr Tamir[5], Helmut Veith[†6], and Florian Zuleger[7]

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.

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 $\mathrm{FO}^2_{\mathrm{BD}}$, 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 $\mathrm{FO}^2_{\mathrm{BD}}$ 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 $\mathrm{FO}^2_{\mathrm{BD}}$ reasoning tool.

Main contributions

1. We define SmpSQL, an SQL fragment which is contained in $\mathrm{FO}^2_{\mathrm{BD}}$.
2. We define a simple imperative script language SmpSL with embedded SmpSQL statements.
3. We give a construction for weakest preconditions in $\mathrm{FO}^2_{\mathrm{BD}}$ for SmpSL.
4. We implemented the weakest precondition computation for SmpSL.
5. We implemented a decision procedure for $\mathrm{FO}^2_{\mathrm{BD}}$. 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^2 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 $NS = NewsletterSubscription$ with attributes *nwl*, *user*, *subscribed* and *code*. The database is a structure whose universe is partitioned into three sets: $\mathbf{dom}^{\mathrm{U}}$, $\mathbf{bool}^{\mathrm{B}}$, and $\mathbf{codes}^{\mathrm{B}}$. The attributes *nwl* and *user* range over the finite set $\mathbf{dom}^{\mathrm{U}}$, the attribute *subscribed* ranges over $\mathbf{bool}^{\mathrm{B}} = \{true, false\}$, and the attribute *code* ranges over the fixed finite set $\mathbf{codes}^{\mathrm{B}}$. The superscripts in $\mathbf{dom}^{\mathrm{U}}$, $\mathbf{bool}^{\mathrm{B}}$, and $\mathbf{codes}^{\mathrm{B}}$ serve to indicate that the domain $\mathbf{dom}^{\mathrm{U}}$ is unbounded, while the Boolean domain and the domain of codes are bounded (i.e. of fixed finite size). When $s = true$, $(n, u, s, c) \in NS$ 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 `subscribe`, `unsubscribe`, and `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 `subscribe` is called with inputs n and u (e.g. by a web interface operated by an admin or by the user). `subscribe` stores the tuple $(n, u, false, new\_code)$ 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 `confirm` with input new_code, which subscribes u to n by replacing the tuple $(n, u, false, new\_code)$ of NS to with $(n, u, true, \mathtt{nil})$. For `unsubscribe` the process is similar, and crucially, `unsubscribe` uses the same `confirm` function. `confirm` decides between subscribe and unsubscribe according to whether n is currently subscribed to u. The `CHOOSE` command selects one row non-deterministically. The database preserves the invariant

$$Inv \quad = \quad \forall_{\mathbf{d}} x, y. \forall_{\mathbf{b}} s_1, s_2. \forall_{\mathbf{c}} c_1, c_2. \left((s_1 = s_2 \wedge c_1 = c_2) \vee \bigvee_{i=1,2} \neg NS(x, y, s_i, c_i) \right) \quad (1)$$

```
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 $\text{pre}_{\texttt{f}}$ and $\text{post}_{\texttt{f}}$ for each of the three functions `f`. The relation names **d**, **b**, and **c** are interpreted as the sets $\textbf{dom}^{\mathrm{U}}$, $\textbf{bool}^{\mathrm{B}}$, and $\textbf{codes}^{\mathrm{B}}$, respectively. Proving correctness amounts to proving the correctness of each of the *Hoare triples* $\{\text{pre}_{\texttt{f}} \wedge \mathit{Inv}\}$ `f` $\{\text{post}_{\texttt{f}} \wedge \mathit{Inv}\}$. Each Hoare triple specifies a contract: after every execution of `f`, the condition $\text{post}_{\texttt{f}} \wedge \mathit{Inv}$ should be satisfied if $\text{pre}_{\texttt{f}} \wedge \mathit{Inv}$ was satisfied before executing `f`. $\text{pre}_{\texttt{subscribe}}$ and $\text{pre}_{\texttt{unsubscribe}}$ express that *new_code* is an unused non-nil code and that NS_{gh} is equal to *NS*. NS_{gh} 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. NS_{gh} does not occur in the functions and is not modified. $\text{post}_{\texttt{subscribe}}$ and $\text{post}_{\texttt{unsubscribe}}$ express that *NS* is obtained from NS_{gh} 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*. $\text{pre}_{\texttt{confirm}}$ introduces a Boolean ghost variable sub_{gh} whose value is true (respectively false) if *cd* was generated as a new code in `subscribe` (respectively `unsubscribe`). sub_{gh} does not occur in `confirm`. $\text{post}_{\texttt{confirm}}$ express that, when sub_{gh} is true, *NS* is obtained from *NS* by toggling the value of the column *subscribed* from false to true in the NS_{gh} row whose confirm code is *cd*; when sub_{gh} is false, *NS* is obtained from NS_{gh} 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: $c_{\mathtt{s}}$ by `subscribe` and $c_{\mathtt{u}}$ by `unsubscribe`. At this point, NS contains a single row for the newsletter n and user u namely $(n, u, \mathit{false}, c_{\mathtt{u}})$. The user receives two confirmation emails containing the codes $c_{\mathtt{s}}$ and $c_{\mathtt{u}}$. Clicking on the confirmation URL for $c_{\mathtt{s}}$ (i.e. running $\mathtt{confirm}(c_{\mathtt{s}})$) has no effect since $c_{\mathtt{s}}$ does not occur in the database. However, clicking on the confirmation URL for $c_{\mathtt{s}}$ 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* $\mathrm{wp}[\![\cdot]\!]$ which allows us to automatically derive the weakest precondition for a post-condition with regard to a SmpSL program. For our example functions `f`, $\mathrm{wp}[\![\cdot]\!]$ allows us to automatically derive $\mathrm{wp}[\![\mathtt{f}]\!]\mathrm{post}_{\mathtt{f}}$. The basic property of the weakest precondition is that $\mathrm{post}_{\mathtt{f}}$ holds after `f` has executed iff $\mathrm{wp}[\![\mathtt{f}]\!]\mathrm{post}_{\mathtt{f}}$ held immediately at the start of the execution. It then remains to show that the pre-condition $\mathrm{pre}_{\mathtt{f}}$ implies $\mathrm{wp}[\![\mathtt{f}]\!]\mathrm{post}_{\mathtt{f}}$. This amounts to checking the validity of the verification conditions $VC_{\mathtt{f}} = \mathrm{pre}_{\mathtt{f}} \to \mathrm{wp}[\![\mathtt{f}]\!]\mathrm{post}_{\mathtt{f}}$.

Our reasoner for FO^2 sentences is the back-end for our verification tool. The specification in this example is all in $\mathrm{FO}^2_{\mathrm{BD}}$. The weakest precondition of a SmpSL program applied to a $\mathrm{FO}^2_{\mathrm{BD}}$ sentence gives again a $\mathrm{FO}^2_{\mathrm{BD}}$sentence. Hence $VC_{\mathtt{f}}$ are all in $\mathrm{FO}^2_{\mathrm{BD}}$. Automatically deciding the validity of $\mathrm{FO}^2_{\mathrm{BD}}$ sentences using our FO^2 decision procedure is described in Section 4. Recall that $\mathbf{codes}^{\mathrm{B}}$ is of fixed finite size. Here $|\mathbf{codes}^{\mathrm{B}}| = 3$ 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 $\mathbf{codes}^{\mathrm{B}}$ 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 (s1 = false)` $\cdots$ with `if (sub`$_{\mathtt{gh}}$ `= false)` $\cdots$. Since s_1 is no longer used, the CHOOSE 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 $c_{\mathtt{s}}$ and $sub_{gh} = \mathit{true}$ leaves the database unchanged, while the call to `confirm` with $c_{\mathtt{u}}$ and $sub_{gh} = \mathit{false}$ deletes the row $(n, u, \mathit{false}, c_{\mathtt{u}})$.

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 $\mathrm{FO}^2_{\mathrm{BD}}$ 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 $\mathbf{dom}_1^{\mathrm{B}}, \ldots, \mathbf{dom}_s^{\mathrm{B}}$ called the *bounded domains* and an infinite

$$
\begin{array}{lcl}
\mathrm{pre}_{\mathtt{g}} & = & NS = NS_{gh} \wedge \text{good-code}(new\_code) \\
\text{good-code}(c') & = & \mathbf{c}(c') \wedge (c' \neq nil) \wedge \forall_{\mathbf{d}} x, y.\, \forall_{\mathbf{b}} s.\, \neg NS(x, y, s, c') \\
\mathrm{post}_{\mathtt{g}} & = & \forall_{\mathbf{d}} x, y.\, \forall_{\mathbf{b}} s.\, \forall_{\mathbf{c}} c.\, NS(x, y, s, c) \leftrightarrow (\varphi_{\mathtt{g},1} \vee \varphi_{\mathtt{g},2}) \\
\\
\varphi_{\mathtt{subscribe},1} & = & NS_{gh}(x, y, s, c) \\
\varphi_{\mathtt{subscribe},2} & = & (n = x) \wedge (u = y) \wedge (s = false) \wedge (c = new\_code) \\
 & & \wedge \neg \exists_{\mathbf{b}} s'.\, \exists_{\mathbf{c}} c'.\, NS_{gh}(n, u, s', c') \\
\\
\varphi_{\mathtt{unsubscribe},1} & = & \neg\left((n = x) \wedge (u = y)\right) \wedge NS_{gh}(x, y, s, c) \\
\varphi_{\mathtt{unsubscribe},2} & = & (n = x) \wedge (u = y) \wedge (c = new\_code) \wedge \exists_{\mathbf{c}} c'.\, NS_{gh}(n, u, s, c') \\
\\
\mathrm{pre}_{\mathtt{confirm}} & = & NS = NS_{gh} \wedge \mathbf{b}(sub_{gh}) \\
\mathrm{post}_{\mathtt{confirm}} & = & \bigwedge_{tt \in \mathbf{b}} sub_{gh} = tt \to (\forall_{\mathbf{d}} x, y.\, \forall_{\mathbf{b}} s.\, \forall_{\mathbf{c}} c.\, NS(x, y, s, c) \leftrightarrow \psi_{tt}) \\
\\
\psi_{false} & = & cd \neq c \wedge NS_{gh}(x, y, s, c) \\
\psi_{true} & = & cd \neq c \wedge NS_{gh}(x, y, s, c) \vee (c = nil \wedge s = true \wedge NS_{gh}(x, y, false, cd))
\end{array}
$$

Figure 2 Running Example: Pre- and post-conditions. `g` is either `subscribe` or `unsubscribe`.

set $\mathbf{dom}^{\mathrm{U}}$ called the *unbounded domain*. The domains are disjoint. We assume three disjoint countably infinite sets: the set of attributes **att**, the set of relation names **relnames**, and the set of variables **SQLvars**. We assume a function $\mathbf{sort} : \mathbf{att} \to \{\mathbf{dom}^{\mathrm{U}}, \mathbf{dom}_1^{\mathrm{B}}, \ldots, \mathbf{dom}_s^{\mathrm{B}}\}$. 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*** $\mathrm{ar}(R)$ 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*** $\mathcal{I}$ of a database schema $\mathbf{R}$ is a many-sorted structure with finite domains $\mathbf{dom}_0 \subseteq \mathbf{dom}^{\mathrm{U}}$ and $\mathbf{dom}_j = \mathbf{dom}_j^{\mathrm{B}}$ for $1 \leq j \leq s$. We denote by $\mathbf{sort}_{\mathcal{I}}$ the function obtained from **sort** by setting $\mathbf{sort}_{\mathcal{I}}(att) = \mathbf{dom}_0$ whenever $\mathbf{sort}(att) = \mathbf{dom}^{\mathrm{U}}$. The relation schema $R = (relname, att_1, \ldots, att_e)$ is interpreted in $\mathcal{I}$ as a relation $R^{\mathcal{I}} \subseteq \mathbf{sort}_{\mathcal{I}}(att_1) \times \cdots \times \mathbf{sort}_{\mathcal{I}}(att_e)$. A ***row*** is a tuple in a relation $R^{\mathcal{I}}$.

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

3.1.2 Queries in SmpSQL

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

$$
\begin{array}{lcl}
\langle \mathtt{Select} \rangle & ::= & \mathtt{SELECT}\ \mathtt{att}_{\mathtt{a}_1}, \ldots, \mathtt{att}_{\mathtt{a}_i}\ \mathtt{FROM}\ \mathtt{R}\ \mathtt{WHERE}\ \langle \mathtt{Condition} \rangle \\
\langle \mathtt{Condition} \rangle & ::= & \mathtt{att}_{\mathtt{b}_1}, \ldots, \mathtt{att}_{\mathtt{b}_j}\ \mathtt{IN}\ \langle \mathtt{Select} \rangle \mid \\
 & & \langle \mathtt{Condition} \rangle\ \mathtt{AND}\ \langle \mathtt{Condition} \rangle \mid \\
 & & \langle \mathtt{Condition} \rangle\ \mathtt{OR}\ \langle \mathtt{Condition} \rangle \mid \\
 & & \mathtt{NOT}\ \langle \mathtt{Condition} \rangle \mid \\
 & & \mathtt{att}_{\mathtt{m}} = \mathtt{p}
\end{array}
$$

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

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 $att_1, \ldots, att_n$. Let $p, p_1, \ldots, p_n$ be variables from **SQLvars**. The syntax of the primitive commands is:

$$\begin{array}{lcl} \langle\mathtt{Insert}\rangle & ::= & \mathtt{INSERT}\ (\mathtt{p_1}, \ldots, \mathtt{p_n})\ \mathtt{INTO}\ \mathtt{R} \\ \langle\mathtt{Update}\rangle & ::= & \mathtt{UPDATE}\ \mathtt{R}\ \mathtt{SET}\ \mathtt{att_m} = \mathtt{p}\ \mathtt{WHERE}\ \langle\mathtt{Condition}\rangle \\ \langle\mathtt{Delete}\rangle & ::= & \mathtt{DELETE}\ \mathtt{FROM}\ \mathtt{R}\ \mathtt{WHERE}\ \langle\mathtt{Condition}\rangle \end{array}$$

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 $\mathbf{sort}(att_k)$ to any attribute 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 **connames**, which is disjoint from $\mathbf{att}, \mathbf{dom}^{\mathrm{U}}, \mathbf{dom}_1^{\mathrm{B}}, \ldots, \mathbf{dom}_s^{\mathrm{B}}, \mathbf{relnames}$ but contains **SQLvars**.

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

3.2.2 SmpSL programs

The syntax of SmpSL is given by

$$\begin{array}{lcl} \langle\mathtt{Program}\rangle & ::= & \langle\mathtt{Command}\rangle \mid \langle\mathtt{Program}\rangle\,;\langle\mathtt{Command}\rangle \\ \langle\mathtt{Command}\rangle & ::= & \langle\mathtt{Insert}\rangle \mid \langle\mathtt{Update}\rangle \mid \langle\mathtt{Delete}\rangle \mid \mathtt{R} = \langle\mathtt{Select}\rangle \mid \bar{\mathtt{d}} = \mathtt{CHOOSE}\ \ \mathtt{R} \mid \\ & & \mathtt{if}\ (\mathtt{cond})\ \langle\mathtt{Program}\rangle \mid \mathtt{if}\ (\mathtt{cond})\,\mathtt{exit} \mid \\ & & \mathtt{if}\ (\mathtt{cond})\ \langle\mathtt{Program}\rangle\ \mathtt{else}\ \langle\mathtt{Program}\rangle \end{array}$$

Every data-manipulating command C of SmpSQL is a SmpSL command. The semantics of C in SmpSL 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 $\mathtt{R} = \langle\mathtt{Select}\rangle$ assigns the result of a SmpSQL query to a relation schema $R \in \mathbf{R}$ whose arity and attribute sorts match the select query. Executing the command in a state $(\mathcal{I}, \overline{const}^{\mathcal{I}})$ sets $R^{\mathcal{I}}$ 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 *distinct* members of $\mathbf{dom}^{\mathrm{U}}$. This is so since several program variables or inputs can have the same value.

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

The branching commands have the natural semantics. Two types of branching conditions `cond` are allowed: $(R = \mathtt{empty})$ and $(R \mathrel{!}= \mathtt{empty})$, which check whether $R^{\mathcal{I}}$ is the empty set, and $(c_1 = c_2)$ and $(c_1! = c_2)$, which check whether $c_1^{\mathcal{I}} = c_2^{\mathcal{I}}$.

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 $\mathbf{R}$ is a vocabulary. A state $(\mathcal{I}, \overline{const}^{\mathcal{I}})$ interpreting a state schema $\mathbf{R}$ and a tuple of constant names $\overline{const}$ is an $\langle \mathbf{R}, \overline{const} \rangle$-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 φ_{pre} and φ_{post} be FO-sentences. A ***Hoare triple*** is of the form $\{\varphi_{pre}\}P\{\varphi_{post}\}$. 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*** $\mathrm{wp}[\![P]\!]\varphi_{post}$ of φ_{post} with respect to the program P. The weakest precondition transformer was introduced in Dijkstra's classic paper [17], c.f. [24]. Let $\mathcal{A}_P$ denote the state after executing P on the initial state $\mathcal{A}$. The main property of the weakest precondition is: $\mathcal{A}_P \models \varphi_{post}$ iff $\mathcal{A} \models \mathrm{wp}[\![P]\!]\varphi_{post}$. Using $\mathrm{wp}[\![\cdot]\!]$ we can rephrase the problem of whether the Hoare triple $\{\varphi_{pre}\}P\{\varphi_{post}\}$ is valid in terms of FO reasoning on finite structures: *Is the FO-sentence* $\varphi_{pre} \to \mathrm{wp}[\![P]\!]\varphi_{post}$ *a tautology?* Equivalently, *is the FO-sentence* $\varphi_{pre} \wedge \neg\mathrm{wp}[\![P]\!]\varphi_{post}$ *unsatisfiable?* Section 3.3.3 discusses the resulting FO reasoning task.

We describe the computation of the weakest precondition inductively for SmpSQL and SmpSL. The weakest precondition for SmpSQL is given in Fig. 3, and for SmpSL in Fig. 4. For SmpSQL conditions, $[\![\cdot]\!]^R$ 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. $[\![\mathtt{SELECT}\ \cdots\ \mathtt{FROM\ R} \cdots]\!]$ is also a formula with free variables $v_1, \ldots, v_n$ describing the rows selected by the SELECT query. The rules $\mathrm{wp}[\![s]\!]Q$ 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 ψ by the term t.

The notation $\psi\big[\theta(\alpha_1, \ldots, \alpha_n)/R(\alpha_1, \ldots, \alpha_n)\big]$ indicates that any atomic sub-formula of ψ 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

$$
\begin{array}{lcl}
[\![\mathtt{att_i} = \mathtt{c}]\!]^R & \widehat{=} & v_i = c \\
[\![\mathtt{att_{b_1}}, \ldots, \mathtt{att_{b_j}}\ \mathtt{IN}\ \mathtt{S_1}]\!]^R & \widehat{=} & [\![S_1]\!][v_{b_k}/v_k : 1 \leq k \leq j] \\
\\
[\![\mathtt{cond_1}\ \mathtt{AND}\ \mathtt{cond_2}]\!]^R & \widehat{=} & [\![cond_1]\!]^R \wedge [\![cond_2]\!]^R \\
[\![\mathtt{cond_1}\ \mathtt{OR}\ \mathtt{cond_2}]\!]^R & \widehat{=} & [\![cond_1]\!]^R \vee [\![cond_2]\!]^R \\
[\![\mathtt{NOT}\ \mathtt{cond_1}]\!]^R & \widehat{=} & \neg[\![cond_1]\!]^R
\end{array}
$$

$$
[\![\mathtt{SELECT}\ \mathtt{att_{a_1}}, \ldots, \mathtt{att_{a_i}}\ \mathtt{FROM}\ \mathtt{R}\ \mathtt{WHERE}\ \mathtt{cond}]\!] \ \widehat{=}
$$
$$
\quad (\exists v_{a_{i+1}}, \ldots, v_{a_n} R(\bar{v}) \wedge [\![cond]\!]^R)[v_\ell/v_{a_\ell} : 1 \leq \ell \leq i]
$$
where $\{a_1, \ldots, a_n\} = \{1, \ldots, n\}$

$$
\mathrm{wp}[\![\mathtt{INSERT}\ (\mathtt{c_1}, \ldots, \mathtt{c_n})\ \mathtt{INTO}\ \mathtt{R}]\!]Q \quad \widehat{=} \quad Q\big[R(\bar{\alpha}) \vee \textstyle\bigwedge_{i=1}^n \alpha_i = c_i \ / \ R(\bar{\alpha})\big]
$$

$$
\mathrm{wp}[\![\mathtt{DELETE}\ \mathtt{FROM}\ \mathtt{R}\ \mathtt{WHERE}\ \mathtt{cond}]\!]Q \quad \widehat{=} \quad Q\big[R(\bar{\alpha}) \wedge \neg[\![cond]\!]^R[\alpha_i/v_i : 1 \leq i \leq n] \ / \ R(\bar{\alpha})\big]
$$

$$
\mathrm{wp}[\![\mathtt{UPDATE}\ \mathtt{R}\ \mathtt{SET}\ \mathtt{att_j} = \mathtt{c}\ \mathtt{WHERE}\ \mathtt{cond}]\!]Q \quad \widehat{=}
$$
$$
\quad Q\big[R(\bar{\alpha}) \wedge \neg[\![cond]\!]^R[\alpha_i/v_i : 1 \leq i \leq n] \vee
$$
$$
\qquad \exists v_j R(\overline{\alpha^j}) \wedge [\![cond]\!]^R[\alpha_i^j/v_i : 1 \leq i \leq n] \wedge \alpha_j = c / R(\bar{\alpha})\big]
$$

Figure 3 Rules for weakest precondition for SmpSQL basic commands. We denote by R a relation schema with attributes $\langle att_1, \ldots, att_n\rangle$. We write α_i^j for α_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 $\overline{\alpha^j} = (\alpha_1^j, \ldots, \alpha_n^j)$. Note that each of the last three rows $Q\big[expr(\alpha)\big/R(\bar{\alpha})\big]$ substitutes every occurrence of R with an updated expression $expr$.

$\alpha_1, \ldots, \alpha_n)$. 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 α_i. The α_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 $\mathrm{FO}^2_{\mathrm{BD}}$ 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 $\mathrm{FO}^2_{\mathrm{BD}}$, which extends the classical *two-variable fragment* FO^2. The logic FO^2 is the set of all FO formulas which use only variables the variables x and y. The vocabularies of FO^2-sentences are not allowed function names, only relation and constant names. Note FO^2 cannot express that a relation name is interpreted as a function. FO^2 contains the equality symbol $=$. $\mathrm{FO}^2_{\mathrm{BD}}$ extends 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.

$\mathrm{FO}^2_{\mathrm{BD}}$ is the language of our invariants and pre- and postconditions, see Eq. (1) and Fig. 2 in Section 2. An important property of $\mathrm{FO}^2_{\mathrm{BD}}$ 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 $\mathbf{dom}^{\mathrm{U}}$. We reduce the task of reasoning over $\mathrm{FO}^2_{\mathrm{BD}}$ to reasoning over FO^2.

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

$$
\begin{array}{lll}
[\![\texttt{c}_1 = \texttt{c}_2]\!] & \widehat{=} & c_1 = c_2 \\
[\![\texttt{c}_1 \texttt{ != } \texttt{c}_2]\!] & \widehat{=} & c_1 \neq c_2 \\
\\
[\![\texttt{R} \ != \texttt{empty}]\!] & \widehat{=} & \exists v_1, \ldots, v_n R(v_1, \ldots, v_n) \\
[\![\texttt{R}= \texttt{empty}]\!] & \widehat{=} & \neg\exists v_1, \ldots, v_n R(v_1, \ldots, v_n)
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{wp}[\![\texttt{R} = \texttt{SELECT} \ \cdots]\!]Q & \widehat{=} & Q\big[[\![\texttt{SELECT} \ \cdots]\!](\alpha_1, \ldots, \alpha_n) \big/ R(\alpha_1, \ldots, \alpha_n)\big] \\
\mathrm{wp}[\![(\texttt{d}_1, \ldots, \texttt{d}_n) = \texttt{CHOOSE R}]\!]Q & \widehat{=} & \forall u_1, \ldots, u_n \big(R(u_1, \ldots, u_n) \to Q[u_i/d_i : 1 \leq i \leq n]\big) \\
\mathrm{wp}[\![\texttt{if cond s}_1 \texttt{ else s}_2]\!]Q & \widehat{=} & (\neg[\![cond]\!] \wedge \mathrm{wp}[\![s_2]\!]Q) \vee ([\![cond]\!] \wedge \mathrm{wp}[\![s_1]\!]Q)
\end{array}
$$

Figure 4 Rules for weakest precondition construction for SmpSL basic commands. The weakest precondition of `if cond exit;` $\texttt{s}_2$ is the same as that of `if !cond` $\texttt{s}_2$.

Proof (sketch). By Section 3.3.2, $\{\varphi_{pre}\}P\{\varphi_{post}\}$ is valid iff $\theta = \neg(\varphi_{pre} \wedge \neg\mathrm{wp}[\![P]\!]\varphi_{post})$ is satisfiable by a finite structure. We assume for simplicity that in all the tables, the sort of the first and second attributes att_1 and att_2 is $\mathbf{dom}^{\mathrm{U}}$. By Figs. 3 and 4, the only variables ranging over the unbounded domain are v_1 and v_2. Let θ' be the $\mathrm{FO}^2_{\mathrm{BD}}$ sentence obtained from θ by substituting v_1 and v_2 with x and y respectively, and restricting the range of the quantifiers appropriately: for a command manipulating or querying a table R with attributes $att_1, \ldots, att_n$ in Figs. 3 and 4, each quantifier $\forall v_k$ or $\exists v_k$ is replaced with $\forall_{\mathbf{sort}(att_k)} v_k$ or $\exists_{\mathbf{sort}(att_k)} v_k$. We compute an FO^2 sentence θ'' which is equivalent to θ' by hardcoding the bounded domains. Every table T with an attribute att with $\mathbf{sort}(att) = \mathbf{dom}^{\mathrm{B}}_j$ of size d is replaced with d tables $T_1, \ldots, T_d$ without the attribute att. This change is reflected in θ'', e.g. existential quantification is replaced with disjunction. By the decidability of finite satisfiability of FO^2-sentences, we get that The problem of deciding whether $\{\varphi_{pre}\}P\{\varphi_{post}\}$ is valid is reduced to the decidability of validity for FO^2-sentences. ◀

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 $\mathrm{FO}^2(\tau)$ sentence ϕ over vocabulary τ is satisfiable by any τ-structure – finite or infinite – it is satisfiable by a finite τ-structure whose cardinality is bounded by a computable function of ϕ. Grädel, Kolaitis and Vardi [21] computed an asymptotically-tight exponential bound $bnd(\phi)$, and based on it gave a NEXPTIME algorithm. The algorithm non-deterministically guesses $t \leq bnd(\phi)$ and a τ-structure $\mathcal{A}$ with universe of size t, then checks whether $\mathcal{A}$ satisfies ϕ, 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 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 ψ on a structure whose universe cardinality is bounded by 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

$$\forall x \forall y \left(\alpha(x,y) \wedge \bigwedge_{i=1}^{m} F_i(x,y) \rightarrow \beta_i(x,y) \right) \wedge \bigwedge_{i=1}^{m} \forall x \exists y\, F_i(x,y) \tag{2}$$

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

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

(a) *ψ is in SSNF;*
(b) *The set of cardinalities of the models of ϕ is equal to the corresponding set for ψ; and*
(c) *The size of ψ is linear in the size of ϕ.*

Proposition 2 follows from the discussion before Proposition 3.1 in [21], by applying an additional normalization step converting SNF sentences to SSNF sentences.[2][3]

4.2.2 The CNF formula

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

It is convenient to assume ψ does not contain constants. If ψ 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_ψ encodes a structure with the universe $\{1, \ldots, bnd(\psi)\}$. Later, we will add clauses to guarantee that this structure satisfies ψ. For every unary relation name U in ψ and $\ell_1 \in \{1, \ldots, bnd(\psi)\}$, let v_{U,ℓ_1} be a propositional variable. For every binary relation name R in ψ and $\ell_1, \ell_2 \in \{1, \ldots, bnd(\psi)\}$, let v_{R,ℓ_1,ℓ_2} be a propositional variable. The variables v_{U,ℓ_1} and v_{R,ℓ_1,ℓ_2} encode the interpretations of the unary and binary relation names U and R in the straightforward way (defined precisely below). Let V_ψ be the set of all variables v_{U,ℓ_1} and v_{R,ℓ_1,ℓ_2}.

Given an assignment S to the variables of V_ψ we define the unique structure $\mathcal{A}_{\mathrm{S}}$ as follows:

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

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

Before defining B_ψ precisely we can already state the crucial property of B_ψ:

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

The formula B_ψ is the conjunction of B^{eq}, $B^{\forall\exists}$, and $B^{\forall\forall}$, 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 ψ 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

$$B^{\mathrm{eq}} = \bigwedge_{1\leq \ell_1 \neq \ell_2 \leq m} (\neg v_{=,\ell_1,\ell_2}) \wedge \bigwedge_{1\leq \ell \leq m} v_{=,\ell,\ell}$$

B^{eq} enforces that the equality symbol is interpreted correctly as the equality relation on universe elements.

The $\forall\exists$-conjuncts. For every conjunct $\forall x \exists y\, F_i(x,y)$ and $1 \leq \ell_1 \leq bnd(\psi)$, let $B^{\forall\exists}_{i,\ell_1}$ be the clause $\bigvee_{\ell_2=1}^{bnd(\psi)} v_{F_i,\ell_1,\ell_2}$. This clause says that there is at least one universe element ℓ_2 such that $\mathcal{A}_{\mathrm{S}} \models F(\ell_1,\ell_2)$. Let

$$B^{\forall\exists} = \bigwedge_{1\leq i \leq m} \bigwedge_{1\leq \ell_1 \leq bnd(\psi)} B^{\forall\exists}_{i,\ell_1}$$

For every truth-value assignment S to V_ψ, $\mathcal{A}_{\mathrm{S}}$ satisfies $\bigwedge_{i=1}^{m} \forall x \exists y\, F_i(x,y)$ iff S satisfies $B^{\forall\exists}$.

The $\forall\forall$-conjunct. Let $\forall x \forall y\, \alpha'$ be the unique $\forall\forall$-conjunct of ψ. For every $1 \leq \ell_1, \ell_2 \leq bnd(\psi)$, let α''_{ℓ_1,ℓ_2} denote the propositional formula obtained from the quantifier-free FO^2 formula α' by substituting every atom a with the corresponding propositional variable for ℓ_1 and ℓ_2 as follows:

$$\begin{array}{lllllllll} U(x) & \mapsto & v_{U,\ell_1}, & R(y,y) & \mapsto & v_{R,\ell_2,\ell_2}, & R(x,x) & \mapsto & v_{R,\ell_1,\ell_1} \\ U(y) & \mapsto & v_{U,\ell_2}, & R(x,y) & \mapsto & v_{R,\ell_1,\ell_2}, & R(y,x) & \mapsto & v_{R,\ell_2,\ell_1} \end{array}$$

Let $B^{\forall\forall}_{\ell_1,\ell_2}$ be the Tseitin transformation of α''_{ℓ_1,ℓ_2} to CNF [36], see also [6, Chapter 2]. The Tseitin transformation introduces a linear number of new variables of the form $u^{\gamma}_{\ell_1,\ell_2}$, one for each sub-formula γ of α''_{ℓ_1,ℓ_2}. The transformation guarantees that, for every assignment S of V_ψ, S satisfies α''_{ℓ_1,ℓ_2} iff S can be expanded to satisfy $B^{\forall\forall}_{\ell_1,\ell_2}$. Let

$$B^{\forall\forall} = \bigwedge_{1\leq \ell_1,\ell_2 \leq bnd(\psi)} B^{\forall\forall}_{\ell_1,\ell_2}(\ell_1,\ell_2)$$

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

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.

incorrect	FO^2-solver	Z3
web-subscribe	0.910s	TO
web-unsubscribe	0.741s	OM
firewall	0.876s	OM
conf-bid	0.451s	0.015s
conf-assign	0.369s	0.013s
conf-display	0.992s	0.016s

correct	FO^2-solver	Z3
web-subscribe	1.04s	0.02s
web-unsubscribe	1.46s	0.02s
firewall	18.50s	0.03s
conf-bid	TO	0.22s
conf-assign	1.196s	0.2s
conf-display	TO	0.16s

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.[4]

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 FO^2-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 FO^2-solver.

Artificial examples. In addition, we constructed a set of artificial benchmarks comprising of several families of FO^2-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 FO^2-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 FO^2-solver. We shall conclude that the FO^2-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 FO^2-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 FO^2-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 FO^2:
`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

Table 2 Running time comparison on inflated examples.

		FO^2-solver			Z3		
	multiplier	1	10	100	1	10	100
incorrect	subscribe	0.910s	3.84s	785.2s	TO	TO	TO
	unsubscribe	0.741s	1.70s	209.2s	TO	TO	TO
	firewall	0.876s	3.75s	455.7s	OM	OM	OM
correct	subscribe	1.04s	TO	TO	0.02s	0.03s	0.10s
	unsubscribe	1.46s	TO	TO	0.02s	0.03s	0.09s
	firewall	18.50s	TO	TO	0.03s	0.03s	0.11s

Table 3 Running time comparison on artificial benchmarks.

	size	status	Z3	CVC4	Nitpick	FO^2-solver
2col	3	unsat	0m0.037s	0m0.076s	TO	TO
	4	sat	TO	TO	0m7.038s	0m5.433s
	5	unsat	0m0.702s	0m0.477s	TO	TO
	6	sat	TO	TO	0m8.973s	0m9.323s
	10	sat	TO	TO	0m37.944s	0m19.580s
	11	unsat	1m32.664s	0m30.912s	TO	TO
	14	sat	TO	TO	2m13.661s	TO
	40	sat	TO	TO	TO	TO
alternating-paths	2	sat	0m0.049s	TO	0m11.144s	0m1.105s
	100	sat	TO	TO	TO	0m9.671s
alternating-simple-paths	3	sat	TO	TO	TO	0m6.754s
	4	sat	TO	TO	TO	0m10.128s
	7	sat	TO	TO	TO	TO
	10	sat	TO	TO	TO	TO
exponential	3	sat	TO	TO	0m12.255s	0m1.847s
	4	sat	TO	TO	0m15.358s	11m6.482s
one-var-alternating-sat	300	sat	0m0.037s	0m0.497s	0m11.605s	0m9.720s
one-var-alternating-unsat	5	unsat	0m0.026s	0m0.073s	0m22.537s	0m54.198s
one-var-nested-exists-sat	300	sat	0m0.031s	0m0.045s	0m7.132s	0m0.562s
one-var-nested-forall-sat	500	sat	0m0.033s	TO	0m7.183s	0m7.318s
path-unsat	2	unsat	0m0.033s	0m0.044s	TO	1m37.099s
	3	unsat	0m0.030s	0m0.062s	TO	1m35.451s
	6	unsat	0m0.037s	0m0.891s	TO	1m39.209s

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 (∃FO) to formalize local changes to the database and allow the verification of LTL properties whose atoms are given by ∃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 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 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 FO^2_{BD} is decidable. We implemented a solver for the 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 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 FO^2 solver on unsatisfiable inputs.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

2 Shqiponja Ahmetaj, Diego Calvanese, Magdalena Ortiz, and Mantas Simkus. Managing change in graph-structured data using description logics. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 966–973, 2014. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8238`.

3 Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=2032305.2032319`.

4 Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. In Aarti Gupta and Daniel Kroening, editors, *Satisfiability Modulo Theories (SMT 2010)*, 2010. `http://www.SMT-LIB.org`.

5 Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT race 2010. FMV report series technical report 10/1, Johannes Kepler University, Linz, Austria, 2010.

6 Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

7 Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *International Conference on Interactive Theorem Proving*, pages 131–146. Springer, 2010.

8 Diego Calvanese, Tomer Kotek, Mantas Šimkus, Helmut Veith, and Florian Zuleger. Shape and content. In *Integrated Formal Methods*, pages 3–17. Springer, 2014.

9 Diego Calvanese, Magdalena Ortiz, and Mantas Simkus. Verification of evolving graph-structured data under expressive path constraints. In *19th International Conference on Database Theory, ICDT 2016, Bordeaux, France, March 15-18, 2016*, pages 15:1–15:19, 2016. `doi:10.4230/LIPIcs.ICDT.2016.15`.

10 Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *Knowledge and Data Engineering, IEEE Transactions on*, 1(1):146–166, 1989.

11 Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

12 Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pages 154–169, 2000. `doi:10.1007/10722167_15`.

13 Edgar F Codd. *Relational completeness of data base sublanguages*. IBM Corporation, 1972.

14 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

15 Alin Deutsch, Richard Hull, and Victor Vianu. Automatic verification of database-centric systems. *SIGMOD Record*, 43(3):5–17, 2014. `doi:10.1145/2694428.2694430`.

16 Alin Deutsch, Monica Marcus, Liying Sui, Victor Vianu, and Dayou Zhou. A verifier for interactive, data-driven web applications. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD'05, pages 539–550, New York, NY, USA, 2005. ACM. `doi:10.1145/1066157.1066219`.

17 Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

18 Michael Felderer, Matthias Büchler, Martin Johns, Achim D Brucker, Ruth Breu, and Alexander Pretschner. Security testing: A survey. *Advances in Computers*, 2015.

19 Mary F. Fernández, Daniela Florescu, Alon Y. Levy, and Dan Suciu. Declarative specification of web sites with Strudel. *VLDB J.*, 9(1):38–55, 2000. doi:10.1007/s007780050082.

20 Daniela Florescu, Valerie Issarny, Patrick Valduriez, and Khaled Yagoub. Weave: A data-intensive web site management system. In *In Proc. of the Conf. on Extending Database Technology (EDBT*, 2000.

21 Erich Grädel, Phokion G Kolaitis, and Moshe Y Vardi. On the decision problem for two-variable first-order logic. *Bulletin of symbolic logic*, 3(01):53–69, 1997.

22 Alessandro Grassi and Marco Nenciarini. Panda - the php-based email administrator. http://panda-admin.sourceforge.net/index.php?mode=home, 2007–2015.

23 Charles A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

24 Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):21, 2009.

25 Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *Computer Aided Verification*, pages 1–35. Springer, 2013.

26 S. Kumaran, P. Nandi, T. Heath, K. Bhaskaran, and R. Das. Adoc-oriented programming. In *In Symp. on Applications and the Internet (SAINT)*, 2003.

27 Leonid Libkin. Expressive power of SQL. *Theoretical Computer Science*, 296(3):379–404, 2003.

28 William Mccune. A davis-putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, Argonne National Laboratory, 1994.

29 Moodle. http://sourceforge.net/projects/moodle/, 2001–2015.

30 Michael Mortimer. On languages with two variables. *Mathematical Logic Quarterly*, 21(1):135–140, 1975.

31 A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42, 2003.

32 Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

33 Arend Rensink. Canonical graph shapes. In David Schmidt, editor, *Programming Languages and Systems*, volume 2986 of *Lecture Notes in Computer Science*, pages 401–415. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-24725-8_28.

34 Alexandre Riazanov and Andrei Voronkov. The design and implementation of Vampire. *AI communications*, 15(2, 3):91–110, 2002.

35 Boris Trakhtenbrot. The impossibility of an algorithm for the decidability problem on finite classes. In *Proceedings of the USSR Academy of Sciences*, volume 70, pages 569–572, 1950.

36 Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

37 Hugh E Williams and David Lane. *Web database applications with PHP and MySQL*. O'Reilly Media, Inc., 2004.