Test-Case Generation for SQL Nested Queries with Existential Conditions

Rafael Caballero[1] * **José Luzón-Martín** [2] **Antonio Tenorio** [3]

[1] rafa@sip.ucm.es [2] jose.11.10.88@gmail.com [3] senrof21@gmail.com
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain

Abstract: This paper presents a test-case generator for SQL queries. Starting with a set of related SQL views that can include existential subqueries in the conditions, the technique finds a database instance that can be used as a test-case for the target view. The proposal reduces the problem of generating the test-cases to a Constraint Satisfaction Problem using finite domain constraints. In particular, we present a new approach for existential conditions that makes possible to find test-cases for a wider set of queries. The soundness and correctness of the technique with respect to a simple operational semantics for SQL queries without aggregates is proven. The theoretical ideas have been implemented in an available prototype.

Keywords: SQL, Test-cases, constraints, finite domains

1 Introduction

This paper presents a technique for producing test-cases that allows the checking of a set of correlated SQL [SQL92] views. Test-cases are particularly useful in the context of databases, because checking if queries which are defined over real-size database instances are valid is a very difficult task. Therefore, the goal is to automatically obtain a small database instance that allows the user to readily check if the defined SQL views work as expected. There are many different possible coverage criteria for test-cases (see [ZHM97, AO08] for a general discussion). In the particular case of SQL [CT04, ST09], it has been shown that good criteria like the Full Predicate Coverage (FPC) can be obtained by transforming the initial SQL query into a set of independent queries, where each one is devoted to checking a particular component of the query and then obtaining a *positive test-case* for each query in this set. A positive test-case is a test case following the simple criterium of being non-empty database instances such that the query/relation studied produces a non-empty result. As proposed in [CGS10], negative test-cases can be defined in terms of positive test-cases, thus, negative test cases can also be obtained with a positive test-case generator. A first step towards a solution was presented in [CGS10], where the problem was reduced to a Constraint Satisfaction Problem (CSP in short). Although self-contained, this paper must be seen as an improvement with respect to this previous work. More specifically, the main contribution is that queries including existential subqueries are allowed. These subqueries are introduced in SQL conditions by means of the reserved word exists and play an important role in

* Work partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502)

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 103-117, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

defining SQL views. In [CGS10] the treatment of these subqueries was very limited (only if the exists occurred as part of a conjunction at the outer level). Moreover, the operational semantics used in that work (the Extended Relation Algebra, ERA in short) does not allow subqueries, a very important feature that could not be included in the theoretical results. We overcome this limitation in the current paper by:

1. Defining a new SQL Operational Semantics (called SOS), presented in Section 2 that allows existential subqueries as part of the conditions in the where clause. We prove the equivalence of SOS and ERA over the set of basic queries without aggregates.
2. Defining a CSP that includes representation of existential subqueries (Section 3), and proving its correctness and completeness.
3. Implementing a new prototype (Section 4) that generates the test-cases using a constraint solver.

The work is concluded in Section 5, which summarizes the results and discusses future work.

2 SQL operational semantics

A *table schema* is of the form $T(A_1,\ldots,A_n)$, with T the table name and A_i attribute names for $i = 1\ldots n$. We will refer to a particular attribute A by using the notation $T.A$. Each attribute A has an associated type (*integer*, *string*, ...) represented by $type(T.A)$. In this paper we only consider *integer* types, although the inclusion of other data types will be considered in future work.

An *instance* of a table schema $T(A_1,\ldots,A_n)$ will be represented as a finite multiset of functions (called rows) $\{|\mu_1,\mu_2,...,\mu_m|\}$ such that $dom(\mu_i) = \{T.A_1,\ldots,T.A_n\}$, and $\mu_i(T.A_j) \in type(T.A_j)$ for every $i = 1,\ldots,m$, $j = 1,\ldots,n$. Observe that we qualify the attribute names in the domain by table names. This is done because in general we will be interested in rows that combine attributes from different tables, usually as result of cartesian products. In the following, it is useful to consider each attribute $T.A_i$ in $dom(\mu)$ as a logic variable, and μ as a logic substitution.

The concatenation of two rows μ_1, μ_2 with disjoint domain is defined as the union of both functions represented as $\mu_1 \odot \mu_2$. Given a row μ and an expression e we use the notation $e\mu$ to represent the value obtained applying the substitution μ to e.

If $dom(\mu) = \{T.A_1,\ldots,T.A_n\}$ and $\nu = \{U.A_1 \mapsto T.A_1,\ldots,U.A_n \mapsto T.A_n\}$ (i.e., ν is a table renaming) we will use the notation μ^U to represent the substitution composition $\nu \circ \mu$. The previous concepts for concatenations and substitutions can be extended to multisets of rows in a natural way. For instance, given the multiset of rows S and the row μ, $S\mu$ represents the application of μ to each member of the multiset. Analogously, given two multisets S_1, S_2, we define

$$S_1 \odot S_2 = \{\nu \circ \mu \mid \nu \in S_1, \mu \in S_2\}$$

which constitutes the natural representation of *cartesian products* in relational databases. Observe that although our representation for rows is different from the representation used usually in relational databases (tuples), it is possible to define an isomorphism $\ll \cdot \gg$ between the two representations. If R is a relation and $\mu \in R$ is defined as $\mu = \{|T_1.A_1 \mapsto a_1,\ldots T_n.A_n \mapsto a_n|\}$,

then $\ll\mu\gg = (a_1,\ldots,a_n)$ where the order in the tuple is the lexicographic order of the attribute names. In order to define the inverse transformation we assume that every relation has a well-defined schema with attribute names $(T_1.A_1,\ldots,T_n.A_n)$ with the attribute name positions in lexicographic order. Obviously, if R is either a table or a view, $T_i = R$ for $i = 1\ldots n$, but in the case of queries we can assume different relation names as prefixes. Then if $t = (a_1,\ldots,a_n)$ is a tuple in R we define $\ll\mu\gg^{-1} = T_1.A_1 \mapsto a_1,\ldots T_n.A_n \mapsto a_n$. It can be checked that the transformation is distributive with respect to $\prod$ and $\odot$, this means that $\ll t_1 \times t_2\gg^{-1} = \ll t_1\gg^{-1} \odot \ll t_2\gg^{-1}$ and $\ll\mu_1 \odot \mu_2\gg = \ll\mu_1\gg \times \ll\mu_2\gg$.

Definition 1 SQL view and query syntax.
The general form of a view is: create view V($A_1,\ldots,A_n$) as Q, with Q a query and $V.A_1,\ldots V.A_n$ the name of the view attributes. Queries can be:

1. Basic queries Q = select $e_1,\ldots,e_n$ from $R_1\ B_1\ ,\ldots,R_m\ B_m$ where C;

   with R_j tables or views for $j = 1\ldots m$, e_i, $i = 1\ldots n$ expressions involving constants, predefined functions and attributes of the form $B_j.A$, $1 \le j \le m$, and A an attribute of R_j. The condition C must have one of the following forms:

   $$C ::= \mathsf{false} \mid \mathsf{true} \mid e_1 \diamond e_2 \mid C_1 \text{ and } C_2 \mid C_1 \text{ or } C_2 \mid \text{not} C1 \mid \mathsf{exists}(\mathsf{Q})$$

   with e_1, e_2 arithmetic expressions, $\diamond \in \{=, <>, >, <, >=, <=\}$, C_1, C_2 SQL conditions and Q an SQL query.

   In the rest of the paper the logic values *true* and *false* are represented respectively by $\top$ and $\bot$ in order to be distinguished from the SQL reserved words true and false.

2. Unions of the form Q_1 union Q_2, with Q_1 and Q_2 queries.

3. Intersections of the form Q_1 intersects Q_2, with Q_1 and Q_2 queries.

A *database schema* D is a tuple $(\mathscr{T},\mathscr{C},\mathscr{V})$, where $\mathscr{T}$ is a finite set of tables, $\mathscr{C}$ a finite set of database constraints and $\mathscr{V}$ a finite set of views. A *database instance* d of a database schema is a set of table instances, one for each table in $\mathscr{T}$ verifying $\mathscr{C}$ (thus we only consider *valid instances*). To represent the instance of a table T in a database instance d we will use the notation $d(T)$. In this paper *primary key* and *foreign key* constraints are considered. They can be formally defined as follows:

Definition 2 Let D be a database schema and d a database instance. Let T be a table in D such that $d(T) = \{|\mu_1,\ldots,\mu_n|\}$, then:

1. If T has a primary key constraint defined over the columns $C_1,\ldots,C_p$, then we say that d is a valid instance with respect to this constraint if for every $i = 1\ldots n$ the logic formula

   $$\bigwedge_{j=1, j\neq i}^{n} \left(\bigvee_{k=1}^{p} \mu_i(T.C_k) \neq \mu_j(T.C_k)\right)$$

   holds (that is, it is evaluated to $\top$).

2. If T has a foreign key constraint for the columns $C_1,\ldots,C_f$ referring the columns $C'_1,...C'_f$ from table T_2. Then suppose that $d(T_2) = \{|v_1,\ldots,v_{n'}|\}$. Then d is a valid instance with respect to this foreign key if for every μ_i, $i = 1\ldots n$ the following logic formula holds:

$$\bigvee_{j=1,j\neq i}^{n'} (\bigwedge_{k=1}^{f} \mu_i(T.C_k) = v_j(T_2.C'_k))$$

That is, in the case of the primary key we require that every pair of rows must differ in at least one attribute of the primary key. In the case of the foreign key, every row μ_i in T requires the existence of another tuple v_j in T_2 such that they have the same values over the attributes defining the primary case in each case.

A *symbolic database instance* d_s is a database instance whose rows can contain logic variables. We say that d_s is satisfied by a substitution μ when $(d_s\mu)$ is a database instance. μ must substitute all the logic variables in d_s by domain values.

Next we present a small example that is used in the rest of the paper.

Example 1 *A board game for multiple players. Players are included into the following table*

```
create table player {  id int primary key   };
```

The board can contain pieces from different players, but only one piece at each position. The board state is represented by the following table:

```
create table board {
  int x,
  int y,
  int id,
  primary key x,y;
  foreign key(id) references player(id);
  };
```

where x, y *are the position in the board and* id *the player identifier.*

We are interested in detecting the ids *of the players that are still playing (that is, they have at least one piece on the board), and all their pieces are* threatened*: for every piece of the player there is a piece of a different player which is either in the same* x *or in the same* y*. The views defining such pieces are the following:*

```
create view nowPlaying(id) as
  select p.id
  from player p
  where exists (select b.id from board b where b.id=p.id);

create view checked(id) as
  select p.id
  from player p
  where exists (select n.id from nowPlaying n where n.id = p.id) and
        not exists (select b1.id from board b1
```

```
where b1.id = p.id and
      not exists
     (select b2.id from board b2
      where (b2.x - b1.x) * (b2.y-b1.y)=0 and
            (b1.id <> b2.id)));
```

The *syntactic dependency tree* for a view, table or query R, with R in the root of this tree, is defined as:

- If R is a table it has no children.
- If R is a view the only child is the syntactic dependency tree of its associated query.
- If R is a query the children are the syntactic dependency trees of the relations occurring in the from section, plus one child for each subquery occurring in the where section.

Now we are ready to define our SQL operational semantics (SOS for short).

Definition 3 Let $\mathscr{D}$ a relational database schema and d be an instance of $\mathscr{D}$. Then the SQL operational semantics, $SOS(d)$ is defined as follows:

1. For any table T defined in $\mathscr{D}$, $\langle T,d\rangle = d(T)$.
2. For any simple query Q

   select $e_1, \ldots, e_n$ from $R_1\ B_1\ , \ldots, R_m\ B_m$ where C; define:

   $$\langle Q,d\rangle = \{s_Q(\mu) \mid \nu_1 \in \langle R_1,d\rangle, \ldots, \nu_m \in \langle R_m,d\rangle, \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}, \langle C\mu,d\rangle\}$$

   where $s_Q(\mu) = \{Q.A_1 \mapsto (e_1\mu), \ldots, Q.A_n \mapsto (e_n\mu)\}$, and $\langle C,d\rangle$ is defined as follows

   - If $C \equiv$ false then $\langle C,d\rangle = \bot$
   - If $C \equiv$ true then $\langle C,d\rangle = \top$
   - If $C \equiv e$, with e an arithmetic expression involving constants then $\langle C,d\rangle = e$
   - If $C \equiv e_1 \diamond e_2$, with $\diamond$ a relational operator, then $\langle C,d\rangle = (\langle e_1,d\rangle \diamond \langle e_2,d\rangle)$.
   - If $C \equiv C_1$ and C_2 then $\langle C,d\rangle = \langle C_1,d\rangle \wedge \langle C_2,d\rangle$
   - If $C \equiv C_1$ or C_2 then $\langle C,d\rangle = \langle C_1,d\rangle \vee \langle C_2,d\rangle$
   - If $C \equiv$ not C_1 then $\langle C,d\rangle = \neg\langle C,d\rangle$
   - If $C \equiv$ exists Q then $\langle C,d\rangle = \langle Q,d\rangle \neq \emptyset$
3. If Q is a view with attributes $A_1,\ldots,A_n$ defined by a query of the form Q_1 union Q_2, then $\langle Q,d\rangle = \langle Q1,d\rangle \cup \langle Q2,d\rangle$ with $\cup$ the union multiset operator.
4. Analogously, if Q is of the form Q_1 intersection Q_2 then, $\langle \mathscr{R},d\rangle = \langle Q1,d\rangle \cap \langle Q2,d\rangle)$, with $\cap$ the intersection multiset operator.
5. For any view V with definition create view V($E_1, \ldots, E_n$) as Q, $\langle V,d\rangle = \langle Q,d\rangle\{V.E_1 \mapsto Q.A_1,\ldots,V.E_n \mapsto Q.A_n\}$.

The following observations can be useful for understanding SOS. Queries are treated as anonymous views, and the name Q is always given as table name to the rows in the result. Analogously A_i is always the name given to the i-th expression in the select clause. This is useful for instance for ensuring that unions and intersections provide homogeneous rows, and also for the renaming applied by views in the last item. The expression $(e_1 \diamond e_2)\mu$ in the second case represents the logic value $\top$ or $\bot$ obtained after replacing each attribute by its value in μ and evaluating the relational expression in usual logic. In well-defined SQL queries all the attributes in the expressions correspond exactly to one attribute of a relation defined in the query. Well-defined existential subqueries must verify that after replacing the external attributes by values they become well-defined queries. This property is exploited in the definition used in the existential condition $C(\mu) = (\langle Q\mu, d\rangle \neq \emptyset)$, which must be read as: first replace in Q all the external attributes by their values, then obtain their semantics, and finally check whether the result is different from the empty multiset.

Using this definition we can define a *positive test-case* (PTC) for a view V as a non-empty database instance d such that $\langle V\rangle \neq \emptyset$. In previous papers we have used the Extended Relational Algebra (ERA from now on) [GUW08] as suitable operational semantics for SQL. However ERA does not allow existential subqueries, a key point of this work that is solved in SOS. Defining ERA and the transformation of SQL into ERA is beyond the scope of this paper, and we only mention some of the basic operations:

- Unions and intersections. The union of R and S, is a multiset $R \cup S$ in which the row μ occurs $n+m$ times. The intersection of R and S, $R \cap S$, is a multiset in which the row μ occurs $min(n,m)$ times.

- Projection. The expression $\pi_{e_1 \mapsto A_1,\ldots,e_n \mapsto A_n}(R)$ produces a new relation producing for each row $\mu \in R$ a new row $\{A_1 \mapsto e_1 \ll \mu \gg, \ldots, A_n \mapsto e_n \ll \mu \gg\}$. That is, we substitute in each e_i the attribute names by their values in μ and then evaluate e_i. The resulting multiset has the same number of rows as R.

- Selection. Denoted by $\sigma_C(R)$, where C is the condition that must be satisfied for all rows in the result. The selection operator on multisets applies the selection condition to each row occurring in the multiset independently.

- Cartesian products. Denoted as $R \times S$, each row in the first relation is paired with each row in the second relation.

- Renaming. The expression $\rho_S(R)$ changes the name of the relation R to S.

A sound requirement is that both ERA and SOS must define the same semantics over the common SQL fragment. This is important because ERA is assumed as the standard SQL semantics.

Theorem 1 *Let D be a database schema and d a database instance. Let R be a relation in D or a query defined over relations in D. Assume that the relations in the syntactic dependency tree of R do include neither aggregates not existential subqueries. Then* $\eta \in \langle R, d\rangle$ *with cardinality k iff* $\ll \eta \gg \in ERA(R,d)$ *with cardinality k.*

Proof. Using induction on the depth of the syntactic dependency tree for R, and distinguishing cases depending on the form of R:
- R is a query of the form select $e_1, \ldots, e_n$ from $R_1\ B_1\ , \ldots, R_m\ B_m$ where C;, and C does not contain subqueries. Then according to Definition 3

$$\langle Q,d\rangle = \{s_Q(\mu) \mid \nu_1 \in \langle R_1,d\rangle,\ldots,\nu_m \in \langle R_m,d\rangle, \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}, \langle C\mu,d\rangle\} \tag{1}$$

And it is easy to check that in the case of conditions without subqueries $\langle C\mu,d\rangle = C'\mu$ where C' is obtained from C by replacing true by $\top$, false by $\bot$, and by $\wedge$, or by $\vee$ and not by $\neg$. Analogously in ERA the query can expressed as:

$$ERA(Q,d) = \prod_{e_1 \mapsto Q.A_1,\ldots,e_n \mapsto Q.A_1} \sigma_{C'}(\rho_{B_1}(ERA(R_1,d)) \times \cdots \times \rho_{B_m}(ERA(R_m,d))) \tag{2}$$

with C' defined as above. Now assume that $\eta \in \langle Q,d\rangle$ with cardinality k. This means that there exists $\mu_1,\ldots,\mu_k$ such that $\eta = s_Q(\mu_i)$ for $i = 1\ldots k$. According to 1 this implies that

$$\mu_i = {\nu_1^i}^{B_1} \odot \cdots \odot {\nu_m^i}^{B_m}, \text{ with } \nu_j^i \in \langle R_j,d\rangle \text{ for } j = 1\ldots m \tag{3}$$

Then applying the induction hypothesis to 3 we have that $\lessdot \nu_j^i \gtrdot \in ERA(R_j,d)$ with the same cardinality for $i = 1\ldots n, j = 1\ldots m$, which means that $\rho {\nu_j^i}^{B_j} \in \rho_{B_j}(ERA(R_j,d))$ for $i = 1\ldots n, j = 1\ldots m$, and therefore $\rho\mu_i \in (\rho_{B_1}(ERA(R_1,d)) \times \cdots \times \rho_{B_m}(ERA(R_m,d)))$. From 1 we have that $\langle C\mu,d\rangle = C'\mu_i$ holds, which means that $\rho\mu_i \in \sigma_{C'}(\rho_{B_1}(ERA(R_1,d)) \times \cdots \times \rho_{B_m}(ERA(R_m,d)))$. Finally, taking into account that $\eta = s_Q(\mu_i)$ for $i = 1\ldots k$, and considering that $\lessdot \eta \gtrdot = \lessdot s_Q(\mu_i) \gtrdot$ = $\{Q.A_1 \mapsto (e_1 \lessdot \mu_i \gtrdot), \ldots, Q.A_n \mapsto (e_n \lessdot \mu_i \gtrdot)\}$ = $\prod_{e_1 \mapsto Q.A_1,\ldots,e_n \mapsto Q.A_1}(\lessdot \mu_i \gtrdot)$ we have from 2 that $\lessdot \eta \gtrdot \in ERA(Q,d)$.

□

3 Generating Constraints

In our tool, initially the user chooses a view V and the number of rows of the initial *symbolic database* instance (that is the number of rows of the tables involved in the computation of V). Each attribute value in each row corresponds to a fresh logic variable with its associated domain integrity constraints. The tool tries to obtain a positive test-case by binding the logic variables in the symbolic database. Notice that the number of rows directly affects the result since for some queries does not exists positive test-cases with an specific size. In the future we plan to decide automatically the more convenient size by inspecting the relations definition. Observe also that currently only integer domains are supported. Extending the proposal to other domains such as strings does not seem difficult to achieve, but is beyond the scope of this work.

For instance, in Example 1, suppose that the user decides to look for a positive test-case with two rows in each table. Then, initially the prototype builds the following symbolic instance:

player
id
player.id.0
player.id.1

board		
id	x	y
board.id.0	board.x.0	board.y.0
board.id.1	board.x.1	board.y.1

Notice that player.id.0, player.id.1, board.id.0, board.id.1, board.x.0, board.x.1, board.y.0, board.y.1 represent logic variables. The next definition is employed by the tool to establish the constraints on these variables.

Definition 4 Let D be a database schema and d a database instance. We define $\theta(R,d)$ for every relation R in D as a multiset of pairs (ψ,u) with ψ a first order formula, and u a row. This multiset is defined as follows:

1. For every table T in D such that $d(T) = \{|\mu_1,\ldots,\mu_n|\}$: where $\mu_i = (T.C_1 \mapsto X_{i1},\ldots,T.C_m \mapsto X_{im})$ then:
   - If the definition of T has neither primary key nor foreign key constraints: $\theta(T,d) = \{|(true,\mu_1),\ldots,(true,\mu_n)|\}$.
   - If the definition of T contains primary or foreign key constraints:
     - if T has a primary key constraint for the columns $C_1,\ldots,C_p$: Let T' be the table T without that primary key constraint. Assume that $\theta(T',d) = \{|(\psi_1,\mu_1),\ \ldots,\ (\psi_n,\mu_n)|\}$, then:

$$\theta(T,d) = \{|((\psi_i \wedge (\bigwedge_{j=1,j\neq i}^{n} (\bigvee_{k=1}^{p} \mu_i(T.C_k) \neq \mu_j(T.C_k)))),\mu_i)|i \in 1,\ldots,n|\}$$

     - if T has a foreign key constraint for the columns $T.C_1,\ldots,T.C_f$ referring the columns $T2.C'_1,...T2.C'_f$ from table $T2$: Let T' be the T without that foreign key constraint. Assume that $\theta(T',d) = \{|(\psi_1,\mu_1),\ldots,(\psi_n,\mu_n)|\}$, and that $d(T_2) = \{|\nu_1,\ldots,\nu_{n'}|\}$. Then:

$$\theta(T,d) = \{|((\psi_i \wedge (\bigvee_{j=1,j\neq i}^{n'} (\bigwedge_{k=1}^{f} \mu_i(T.C_k) = \nu_j(T2.C'_k)))),\mu_i)|i \in 1,\ldots,n|\}$$

2. For every view $V =$ create view V($E_1,\ldots,E_n$) as Q,

$$\theta(V,d) = \theta(Q,d)\{V.E_1 \mapsto Q.A_1,\ldots,V.E_n \mapsto A_n\}$$

3. If Q is a basic query of the form:

$$\text{select } e_1,\ldots,e_n \text{ from } R_1\ B_1,\ldots,R_m\ B_m \text{ where } C_w;$$

   Then:

$$\theta(Q,d) = \{|(\psi_1 \wedge \cdots \wedge \psi_m \wedge \varphi(C_w\mu,d), s_Q(\mu)) \mid (\psi_1,\nu_1) \in \theta(R_1,d),\ldots,(\psi_m,\nu_m) \in \theta(R_m,d), \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}|\}$$

   where

   - $s_Q(\mu) = \{Q.A_1 \mapsto (e_1\mu),\ldots,Q.A_n \mapsto (e_n\mu)\}$,
   - The first order formula $\varphi(C,d)$ is defined as

- If $C \equiv$ false then $\varphi(C,d) = \bot$
- If $C \equiv$ true then $\varphi(C,d) = \top$
- If $C \equiv e$, with e an arithmetic expression involving constants then $\varphi(C,d) = e$
- If $C \equiv e_1 \diamond e_2$, with $\diamond$ a relational operator, then $\varphi(C,d) = (\varphi(e_1,d) \diamond \varphi(e_2,d))$.
- If $C \equiv C_1$ and C_2 then $\varphi(C,d) = \varphi(C_1,d) \wedge \varphi(C_2,d)$
- If $C \equiv C_1$ or C_2 then $\varphi(C,d) = \varphi(C_1,d) \vee \varphi(C_2,d)$
- If $C \equiv$ not C_1 then $\varphi(C,d) = \neg\varphi(C,d)$
- If $C \equiv$ exists Q then suppose that $\theta(Q,d) = \{|(\psi_1,\mu_1),\ldots(\psi_p,\mu_p)|\}$. Then $\varphi(C,d) = (\vee_{j=1}^{p} \psi_j)$.

4. For set queries:

   - $\theta(V_1$ union $V_2,d) = \theta(V_1,d) \cup \theta(V_2,d)$ with $\cup$ the multiset union.
   - $(\psi,\mu) \in \theta(V_1$ intersection $V_2,d)$ with cardinality k iff $(\psi_1,\mu) \in \theta(V_1,d)$ with cardinality k_1, $(\psi_2,\mu) \in \theta(V_2,d)$ with cardinality k_2, $k = min(k_1,k_2)$ and $\psi = \psi_1 \wedge \psi_2$.

Observe that the notation $s_Q(x)$ with Q a query is a shorthand for the row μ with domain $\{E_1,\ldots,E_n\}$ such that $(E_i)x = (e_i)x$, with $i = 1\ldots n$, with select e_1 E_1, $\ldots$, e_n E_n the select clause of Q. If E_i's are omitted in the query, it is assumed that $E_i = e_i$.

The following result and its corollary represent the main result of this paper, stating the soundness and completeness of our proposal:

Theorem 2 *Let D be a database schema and d a valid database instance. Let R be either a relation in D or a query defined using relations in D. Then $\eta \in \langle R,d\rangle$ with cardinality k iff $(true,\eta) \in \theta(R,d)$ with cardinality k.*

Proof. The result is proven by using complete induction on the depth of the syntactic dependence tree defining R.
- If R is a table. Suppose that $d(R) = \{|\mu_1,\ldots,\mu_n|\}$. We distinguish cases depending on the database constraints in the definition of R.

we prove the result using induction on the number of constraints m in the definition of table R.
- If $m = 0$, that is, R is defined with neither primary key nor foreign key constraints. In SOS by Definition 3, $\langle T,d\rangle = d(T) = \{|\mu_1,\ldots,\mu_n|\}$. From Definition 4, $\theta(T,d) = \{|(true,\mu_1),\ldots,(true,\mu_n)|\}$. Then every element η occurs in $\langle R,d\rangle$ with the same cardinality a $(true,\eta)$ in $\theta(R,d)$.
- If $m > 0$ R is defined using at least one primary or foreign key constraints. Select any constraint. Suppose that the selected constraint is a primary key defined by columns $C_1,\ldots,C_p$. Then if $\mu_i \in \langle R,d\rangle$, it appears only once due to the primary key constraints. Let us see that $\mu_i \in \langle R,d\rangle$ iff $(true,\mu_i) \in \theta(R,d)$ (with cardinality 1). From Definition 2.1, $\mu_i \in \langle R,d\rangle$, with d a valid instance implies that

$$\bigwedge_{j=1,j\neq i}^{n} \left(\bigvee_{k=1}^{p} \mu_i(R.C_k) \neq \mu_j(R.C_k)\right) \tag{4}$$

Defined T' as R without this constraint, as explained in Definition 4.1. Then d is also a valid instance for T', and thus $d(T') = d(R)$ which means $\langle T',d\rangle = \langle (R,d)\rangle$. Assume that $\theta(T',d) =$

$\{|(\psi_1,\mu_1),\ldots,(\psi_n,\mu_n)|\}$, and from Definition 4.1

$$\theta(T,d) = \{|(\psi_i \wedge (\bigwedge_{j=1,j\neq i}^{n} (\bigvee_{k=1}^{p} \mu_i(R.C_k) \neq \mu_j(R.C_k)))),\mu_i)|i \in 1,\ldots,n|\} \tag{5}$$

$\mu_i \in \langle R,d\rangle$ iff (from $\langle T',d\rangle = \langle (R,d)\rangle$) $\mu_i \in \langle T',d\rangle$ iff $(true,\mu_i) \in \theta(T',d)$ (by induction hypothesis) iff $\psi_i = true$. Moreover, $\mu_i \in \langle R,d\rangle$ iff 4 is $\top$ (because d is valid) iff (true,μ_i) $\in \theta(R,d)$ (replacing ψ_i and 4 by $\top$ in 5). The result is analogous if the selected constraint is a foreign key.

- If R is a view then $\theta(V,d) = \theta(Q,d)\{V.E_1 \mapsto Q.A_1,\ldots,V.E_n \mapsto Q.A_n\}$ with $E_1,\ldots,E_n$ the attribute names defined by the view (Def. 4), and $\langle V,d\rangle = \langle Q,d\rangle\{V.E_1 \mapsto Q.A_1,\ldots,V.E_n \mapsto Q.A_n\}$ (Def. 3) and the result is straightforward from the induction hypothesis applied to Q.
- If R is a simple query of the form:

  select $e_1,\ldots,e_n$ from $R_1\ B_1\ ,\ldots,R_m\ B_m$ where C;

  According to Definition 3

$$\langle Q,d\rangle = \{s_Q(\mu) \mid \nu_1 \in \langle R_1,d\rangle,\ldots,\nu_m \in \langle R_m,d\rangle, \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}, \langle C\mu,d\rangle\} \tag{6}$$

and by Definition 4

$$\begin{aligned}\theta(Q,d) = \quad & \{|(\psi_1 \wedge \cdots \wedge \psi_m \wedge \varphi(C\mu,d), s_Q(\mu)) \mid \\ & (\psi_1,\nu_1) \in \theta(R_1,d),\ldots,(\psi_m,\nu_m) \in \theta(R_m,d), \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}|\}\end{aligned} \tag{7}$$

The first step is to check that

$$\varphi(C\mu,d) \text{ iff } \langle C\mu,d\rangle \tag{8}$$

The result is straightforward from the respective definitions (3 and 4) except in the case of existential subqueries. In this case Definition 3 indicates that an existential subquery Q' is transformed into $\langle Q',d\rangle \neq \emptyset$, which is true iff there is some $\eta \in \langle Q',d\rangle$. By the inductive hypothesis this means that $(true,\eta) \in \theta(Q',d)$. Thus, in the definition 4 the multiset $\theta(Q',d) = \{|(\psi_1,\mu_1),\ldots(\psi_p,\mu_p)|\}$ contains at least one tuple (ψ_i,μ_i) with $\psi_i = true$ for some $1 \leq i \leq p$, which implies that $\varphi(C\mu,d) = (\vee_{j=1}^{p} \psi_j)$ is true, proving 8.

Then $(true,\eta) \in \theta(Q,d)$ with cardinality k iff there are exactly k values such that for $i = 1\ldots k$

$$\mu_i = {\nu_1^i}^{B_1} \odot \cdots \odot {\nu_m^i}^{B_m}, \text{ for some } (\psi_1^i,\nu_1^i) \in \theta(R_1,d),\ldots,(\psi_m^i,\nu_m^i) \in \theta(R_m,d) \tag{9}$$

$$\eta = s_Q(\mu_i) \tag{10}$$

$$\psi_1^i = true,\ldots,\psi_m^i = true \tag{11}$$

$$\varphi(C\mu,d) = true \text{ and thus by Def. 3 } \langle C\mu,d\rangle = true \tag{12}$$

From 11 and considering 9, applying the induction hypothesis:

$$\nu_1^i \in \langle R_1,d\rangle,\ldots,\nu_m^i \in \langle R_m,d\rangle, i = 1\ldots k \tag{13}$$

and thus $s_Q(\mu_i) \in \langle Q,d\rangle$ in 6 for $i = 1 \ldots m$. That is (considering 10), $\eta \in \langle Q,d\rangle$ with multiplicity k.

- If R is a query of the form: Q_1 union Q_2 then $\langle Q,d\rangle = \langle Q1,d\rangle \cup \langle Q2,d\rangle$ (Def. 3), $\theta(V_1 \text{ union } V_2,d) = \theta(V_1,d) \cup \theta(V_2,d)$ (Def. 4), and the result is an easy consequence of the induction hypothesis.
- If R is a query of the form: Q_1 intersection Q_2 the result is similar to the previous case and it is omitted for the sake of space. □

Observe that in [CGS10] the proof was restricted to queries without subqueries due to the limitations of the Extended Relational Algebra [GUW08] used as operational semantics.

The following corollary contains the idea for generating constraints that will yield the PTCs:

Corollary 1 *Let D be a database schema and d_s a symbolic database instance. Let R be a relation in D such that $\theta(R) = \{|(\psi_1,\mu_1),\ldots(\psi_n,\mu_n)|\}$, and η a substitution satisfying d_s. Then $d_s\eta$ is a PTC for R iff $(\bigvee_{i=1}^{n} \psi_i)\eta = true$.*

Proof. Straightforward from Theorem 2: $(\bigvee_{i=1}^{n} \psi_i)\eta = true$ iff there is some ψ_i with $1 \leq i \leq n$ such that $\psi_i\eta = true$ iff $(\mu_i\eta) \in \langle R\rangle$ iff $\langle R\rangle \neq \emptyset$. □

4 Prototype

In this section, we comment on some aspects of our implementation and show a system session for our running example. The SQL Test Case Generator (*STCG*) is implemented in the programming language C++.

The input of STCG consists of a table or view name R, and a SQL file containing the definition of R and of all the relations in its syntactic dependence tree. The goal is to obtain a positive test-case for R.

STCG can be seen as a pipeline, which helps to maintain and extend the implementation. In a first phase the prototype generates a representation in C++ of the CSP. This phase consists of two steps:

1. **SQL parser**: STCG includes a SQL parser to pre-process the input file. The result is a C++ representation of the database schema. This parser has been produced using *GNU Bison* [GNU], a general-purpose parser generator, and *Flex* [CF04], a tool for generating scanners.

2. **Formula Generator**: The core of STCG. Takes as input the output of the previous step, and following Definition 4 described in section 3, generates the first order logic formula representing the constraints associated to the table or view specified.

Figure 1 shows a diagram of this first phase. Next, we want to satisfy this formula by binding the logic variables, thus obtaining a positive test case if possible.

In the second phase (see Figure 2) the logic formula is translated into a format accepted by some external constraint solver. We have used *G12/CPX* as solver, which is distributed with *G12 MiniZinc distribution*, however, any solver that implements FlatZinc can be used for this purpose. *MiniZinc* [NSB+07] is a medium-level constraint modeling language. We translate our

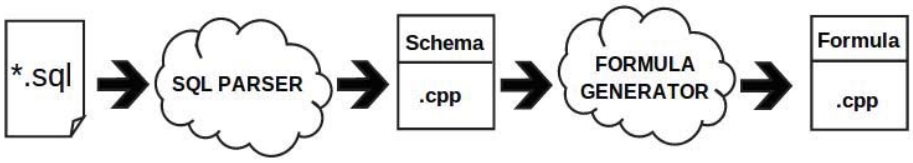

Figure 1: Pipeline of STCG.

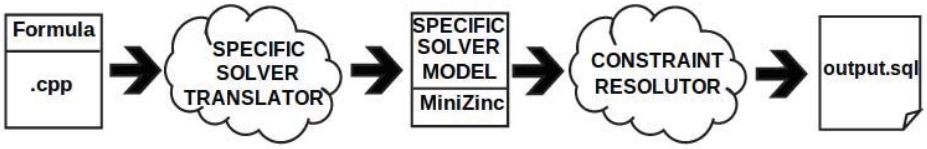

Figure 2: Pipeline of STCG extended with an input builder for a solver.

logic formula into a *MiniZinc model*, so it can be solved, returning, if possible, the positive test-cases. Other constraint programing languages can be added by implementing new translators.

Below we present a system session for the example at hand. Consider that the input SQL file corresponds to Example 1.

Following the pipeline, the SQL parser takes this file as input and generates an internal C++ representation of the database schema (it can be obtained at the command line using the "-v" option of STCG). Suppose that the user intends to generate a test case for view *checked* indicating that tables *player* and *board* must have two rows at most. Thus, the *Formula Generator* module takes the database schema as input and generates the following formula (where symbolic variables are of the form *table name.column name.row*):

```
#(checked)=
 {(((((board.id.0 ==  player.id.0) OR (board.id.1 == player.id.0))
 AND ...
 (not (((((board.x.0 - board.x.1) * (board.y.0 - board.y.1)) == 0)
 AND (board.id.1 != board.id.0)) OR ...)))))),
 {checked.id -> player.id.0}), ...}
```

This corresponds to $\theta(checked, board)$ without considering primary and foreign keys (displayed below). For the sake of space we display only part of the first formula and the first row $\{\mathsf{checked.id} \rightarrow \mathsf{player.id.0}\}$. This row indicates that all the pieces of player.id.0 are checked if the formula holds. Firstly, the formula indicates that player.id.0 must have a piece on either the position corresponding to board.id.0 or to board.id.1. The other part of the formula displayed indicates that there is a check between positions contained in board.x.0,board.y.0, and board.x.1,board.y.1 for two different players.

Primary key constraint of table *player* is shown below. For simplicity, only logic formula for the first row $\{$player.id $\rightarrow$ player.id.0$\}$ is displayed. That formula indicates that this table cannot have two players with the same id attribute, that is player.id.0 != player.id.1.

```
#(player)={(player.id.0 != player.id.1),
{player.id -> player.id.0}), ...}
```

Next, we display primary key and foreign key constraints of *board* for the first row $\{$board.x $\rightarrow$ board.x.0, board.y $\rightarrow$ board.y.0, board.id $\rightarrow$ board.id.0$\}$. That formula indicates that id attribute of each row of this table must reference one of id attributes of the table *player* (foreign key), that is board.id.0 == player.id.0 $\wedge$ board.id.1 == player.id.0, and two pieces cannot be in the same game board position, that is (board.x.0 != board.x.1) $\vee$ (board.y.0 != board.y.1) (primary key).

```
#(board)=
{(((board.id.0 == player.id.0) AND (board.id.1 == player.id.0))
AND ((board.x.0 != board.x.1) OR (board.y.0 != board.y.1)),
{board.x-> board.x.0,board.y-> board.y.0,board.id-> board.id.0}),...}
```

Actually, STCG generates only one formula, with primary key and foreign key constraints included, but we show them separated for simplicity.

The next step of the pipeline is to solve this constraints if possible. As mentioned above, we use the MiniZinc constraint modeling language for this purpose. We introduce a new step to the pipeline, that will translate our *Formula* to a *MiniZinc model*. This model consists of four sections: variable declarations, constraint specification, solver invocation and output formatting.

The model is saved in a file that can be later executed by MiniZinc, returning the positive test-case if possible. To obtain meaningful results we restrict the domain of symbolic variables to 1..5 in this session. In the future, our tool will support domain constraints of symbolic variables. The result returned by MiniZinc is:

```
INSERT INTO board(id,x,y) VALUES (1,1,1);
INSERT INTO board(id,x,y) VALUES (2,1,5);
INSERT INTO player(id) VALUES(1);
INSERT INTO player(id) VALUES(2);
```

This result is a positive test-case, written as a set of SQL insert statements that populates the tables of the database with the following data:

player
id
1
2

board		
id	x	y
1	1	1
2	1	5

Thus, the board has player 1 in position (1,1) and player 2 in position (1,5) and both players are checked as defined in the example 1. We can check that this instance is a positive test-case by executing the SQL query select * from checked, which returns the following non empty result:

id
1
2

5 Conclusions and Future Work

We have presented a technique for generating positive test-cases for sets of correlated SQL views. Our setting considers that a database instance constitutes a test-case for a view if the view computes at least one tuple with respect to the instance. Our proposal is to reduce the problem of obtaining a test-case to a Constraint Satisfaction Problem over finite domains. The generated constraints take into account primary and foreign key database constraints if they are defined in an SQL table definition statement. The work improves the proposal of [CGS10] by introducing a complete treatment of existential subqueries. In order to prove the correctness of the technique, a new operational semantics for SQL views without aggregates is introduced. The theoretical ideas have been implemented in a working prototype called STCG, available at:

https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/STCG

The prototype takes a set of views and the name of a particular view V, and generates a positive test-case for V. It uses MiniZinc [NSB+07] as constraint modeling language. The output is given in the form of a list of SQL insert commands that create the database instance.

Although the framework presented is useful for many types of queries, including those defined by set operations and those queries including existential subqueries, it is only a first step since there are many useful SQL features still not supported by our tool. As immediate future work, we plan the integration of aggregate subqueries. The theoretical problem was already addressed in [CGS10], and the implementation seems straightforward. It would also be interesting to extend the SQL operational semantics to aggregates in order to prove the soundness of the proposal. Another limitation that must be solved is the inclusion of null values, which is currently not supported. However, the main limitation of the tool, and thus the main goal for our future work, is to extend the data types allowed for SQL columns. Currently the type of table attributes is limited to integer, and at least varchar (that is, string types) are required in order to obtain a really applicable tool.

From the theoretical point of view it would be interesting to define the introduction of existential nested subqueries directly into ERA, thus replacing SOS by an enriched version of ERA. Another theoretical improvement would be to determine formally the minimum number of rows necessary for ensuring that a positive test-case exists. Currently, it is the user who indicates the number of rows for each table, but we think that the size could be determined by automatically examining the SQL code.

Bibliography

[AO08] P. Ammann, J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[CF04] J. Coelho, M. Florido. CLP(Flex): Constraint Logic Programming Applied to XML Processing. In *Proceedings of the CoopIS/DOA/ODBASE*. Pp. 1098–1112. Springer LNCS 3291, Heidelberg, Germany, 2004.

[CGS10] R. Caballero, Y. García-Ruiz, F. Sáenz-Pérez. Applying Constraint Logic Programming to SQL Test Case Generation. In Blume et al. (eds.), *Functional and Logic Pro-*

gramming. Lecture Notes in Computer Science 6009, pp. 191–206. Springer Berlin / Heidelberg, 2010.

[CT04] M. J. S. Cabal, J. Tuya. Using an SQL coverage measurement for testing database applications. In Taylor and Dwyer (eds.), *SIGSOFT FSE*. Pp. 253–262. ACM, 2004.

[GNU] GNU. Bison - GNU parser generator.

[GUW08] H. Garcia-Molina, J. D. Ullman, J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[NSB+07] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, G. Tack. MiniZinc: Towards a Standard CP Modelling Language. In Bessiere (ed.). Lecture Notes in Computer Science 4741, pp. 529–543. Springer, 2007.

[SQL92] SQL, ISO/IEC 9075:1992, third edition. 1992.

[ST09] M. Surez-Cabal, J. Tuya. Structural Coverage Criteria for Testing SQL Queries. *Journal of Universal Computer Science* 15(3):584–619, 2009.

[ZHM97] H. Zhu, P. A. V. Hall, J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys* 29:366–427, 1997.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Debugging Fuzzy XPath Queries

Jesús M. Almendros-Jiménez[1], Alejandro Luna[2] and Ginés Moreno[3]

[1] jalmen@ual.es
Dpto. de Lenguajes y Computación
Universidad de Almería
04120 Almería (Spain)
[2] Alejandro.Luna@alu.uclm.es
[3] Gines.Moreno@uclm.es
Dept. of Computing Systems
University of Castilla-La Mancha
02071 Albacete (Spain)

Abstract: In this paper we report a preliminary work about XPath debugging. We will describe how we can manipulate an XPath expression in order to obtain a set of alternative XPath expressions that match to a given XML document. For each alternative XPath expression we will give a chance degree that represents the degree in which the expression deviates from the initial expression. Thus, our work is focused on providing the programmer a repertoire of paths that (s)he can use to retrieve answers. The approach has been implemented and tested.

Keywords: XPath; Fuzzy (Multi-adjoint) Logic Programming; Debugging

1 Introduction

The eXtensible Markup Language (XML) is widely used in many areas of computer software to represent machine readable data. XML provides a very simple language to represent the structure of data, using tags to label pieces of textual content, and a tree structure to describe the hierarchical content. XML emerged as a solution to data exchange between applications where tags permit to locate the content. XML documents are mainly used in databases. The XPath language [BBC+07] was designed as a query language for XML in which the path of the tree is used to describe the query. XPath expressions can be adorned with boolean conditions on nodes and leaves to restrict the number of answers of the query. XPath is the basis of a more powerful query language (called XQuery) designed to join multiple XML documents and to give format to the answer.

In spite of the simplicity of the XPath language, the programmer usually makes mistakes when (s)he describes the path in which the data are allocated. Tipically, (s)he omits some of the tags of the path, s(he) adds more than necessary, and (s)he also uses similar but wrong tag names. When the query does not match to the tree structure of the XML tree, the answer is empty. However, we can also find the case in which the query matches to the XML tree but the answer does not satisfy the programmer. Due to the inherent flexibility of XML documents, the same tag can occur at several positions, and the programmer could find answers that do not correspond to her(is) expectations. In other words, (s)he finds a correct path, but a wrong answer. We can also

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 119-133, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.