Symbolic Query Exploration*

Margus Veanes[1], Pavel Grigorenko[2]**, Peli de Halleux[1], and Nikolai Tillmann[1]

[1] Microsoft Research, Redmond, WA, USA
{margus,jhalleux,nikolait}@microsoft.com
[2] Institute of Cybernetics, Tallinn University of Technology, Tallinn, Estonia
pavelg@cs.ioc.ee

Abstract. We study the problem of generating a database and parameters for a given parameterized SQL query satisfying a given test condition. We introduce a formal background theory that includes arithmetic, tuples, and sets, and translate the generation problem into a satisfiability or model generation problem modulo the background theory. We use the satisfiability modulo theories (SMT) solver Z3 in the concrete implementation. We describe an application of model generation in the context of the database unit testing framework of Visual Studio.

1 Introduction

The original motivation behind this work comes from *unit testing* of relational databases. A typical unit test, first populates the database with concrete *test tables*, then evaluates a given *test query* with respect to the tables, and finally checks if the result of the evaluation satisfies a given *test condition*. Typical test conditions are, checking if the result is empty, nonempty, has a certain number of rows, or contains a specific value.

In general, a test query may also be *parameterized*, i.e., involve variables in place of some concrete values, in which case the parameter variables first need to be instantiated with concrete values in a separate step prior to evaluating the query. A test query uses domain specific knowledge about the particular database schema and acts like a usage scenario, much like code in a traditional unit test. A test condition validates the result and acts like a coverage criterion. The task of coming up with concrete test tables and parameters for the test query satisfying the test condition is, on the other hand, a combinatorial problem that is both error-prone and tedious.

We propose a technique that can be used to automate the above data generation problem for a class of SQL queries. For this we introduce a formal background theory $\mathcal{T}^{\Sigma}$ that is rich enough to capture the semantics for the class of queries under consideration, and is tailored for automatic analysis with state of the art SMT solvers. A given query q and a test condition φ are translated into a formula ψ in $\mathcal{T}^{\Sigma}$. The translation is such that, if the formula ψ is satisfiable

* Microsoft Research Technical Report MSR-TR-2009-65

** This work was done during an internship at Microsoft Research, Redmond.

modulo $\mathcal{T}^{\Sigma}$, i.e., ψ has a model S in $\mathcal{T}^{\Sigma}$, then the values of the variables in S, are mapped back to concrete test tables and input parameters for q. Satisfiability checking combined with finding a concrete model as a witness is usually called *model generation*. We illustrate the use of model generation in the context of the Visual Studio database unit testing framework. In this application, model generation is seen as a black box from the user's perspective. There are other well-known applications of model generation in the context of databases, such as integrity and security constraint checking, where this technique could be useful.

In Section 2 we introduce the background theory $\mathcal{T}^{\Sigma}$. In Section 3 we define a formal translation from a class of SQL queries into $\mathcal{T}^{\Sigma}$. In Section 4 we introduce an analysis approach of formulas in $\mathcal{T}^{\Sigma}$ by using satisfiability modulo theories (SMT). In Section 5 we discuss a concrete application for generating database unit tests in Visual Studio, we look at some concrete examples and provide some benchmarks. In Section 6 we discuss future work. Section 7 is about related work.

2 Background $\mathcal{T}^{\Sigma}$

We use a fixed state background $\mathcal{T}^{\Sigma}$ that includes arithmetic, Booleans, tuples, and *finite* sets. The universe is multi-sorted, with all values having a fixed *sort*. The sorts $\mathbb{Z}$, $\mathbb{R}$, and $\mathbb{B}$ are used for integers, reals, and Booleans, respectively; $\mathbb{Z}$ are $\mathbb{R}$ are called *numeric* sorts. The sorts $\mathbb{Z}$, $\mathbb{R}$ and $\mathbb{B}$ are *basic*, so is the *tuple sort* $\mathbb{T}(\sigma_0,\ldots,\sigma_k)$, provided that each σ_i is basic. The *set sort* $\mathbb{S}(\sigma)$ is not basic and requires σ to be basic.

The universe of values of sort σ is denoted by $\mathcal{U}^{\sigma}$. Universes of distinct sorts are disjoint.[3] For each sort σ, there is a specific $\mathit{Default}^{\sigma}$ in $\mathcal{U}^{\sigma}$. In particular, $\mathit{Default}^{\mathbb{B}} = \mathit{false}$, $\mathit{Default}^{\mathbb{Z}} = 0$, $\mathit{Default}^{\mathbb{R}} = 0$, $\mathit{Default}^{\mathbb{S}(\sigma)} = \emptyset$, and for a tuple sort the *Default* tuple is the tuple of *Default*'s of the respective element sorts. There is a function $\mathit{AsReal} : \mathcal{U}^{\mathbb{Z}} \rightarrow \mathcal{U}^{\mathbb{R}}$ that maps integers to corresponding reals.

We refer to a sort σ together with a semantic constraint on $\mathcal{U}^{\sigma}$ as a *type*. In particular, the type $\mathbb{Z}^{+}$ refers to the positive integers, i.e., the constraint is $\forall x^{\mathbb{Z}^{+}}(x > 0)$. An *enum* or *k-enum* type refers to integers 0 through $k-1$ for some $k > 0$.

2.1 Expressions

We use an expression language that we also refer to as $\mathcal{T}^{\Sigma}$. Well-formed expressions or terms of $\mathcal{T}^{\Sigma}$ are shown in Figure 1. A term t of sort σ is written t^{σ}; x^{σ} is a variable of basic sort σ; X^{σ} is a variable where σ is a set sort. We adopt the convention that upper case letters are used for set variables. Boolean terms are also called *formulas*. We always assume that terms are well-sorted but omit the sorts when they are clear from the context. The set of *free variables* of a term t is denoted by $FV(t)$, these are all the variables that have an occurrence in t that is not

[3] We could assume that for distinct set sorts σ_1 and σ_2 the empty set is shared, but we may also assume, as we do here, that there is distinct empty set for each set sort. Either assumption is fine, because all expressions in $\mathcal{T}^{\Sigma}$ are well-sorted.

$$\begin{array}{ll}
T^{\sigma} & ::= x^{\sigma} \mid \mathit{Default}^{\sigma} \mid \mathit{Ite}(T^{\mathbb{B}}, T^{\sigma}, T^{\sigma}) \mid \mathit{TheElementOf}(T^{\mathbb{S}(\sigma)}) \mid \\
 & \quad \pi_i(T^{\mathbb{T}(\sigma_0,\ldots,\sigma_i=\sigma,\ldots)}) \\
T^{\mathbb{T}(\sigma_0,\ldots,\sigma_k)} & ::= \langle T^{\sigma_0}, \ldots, T^{\sigma_k} \rangle \\
T^{\mathbb{Z}} & ::= k \mid T^{\mathbb{Z}} + T^{\mathbb{Z}} \mid k * T^{\mathbb{Z}} \mid \Sigma_i(T^{\mathbb{S}(\mathbb{T}(\sigma_0,\ldots,\sigma_i=\mathbb{Z},\ldots))}) \\
T^{\mathbb{R}} & ::= r \mid T^{\mathbb{R}} + T^{\mathbb{R}} \mid k * T^{\mathbb{R}} \mid \Sigma_i(T^{\mathbb{S}(\mathbb{T}(\sigma_0,\ldots,\sigma_i=\mathbb{R},\ldots))}) \mid \mathit{AsReal}(T^{\mathbb{Z}}) \\
T^{\mathbb{B}} & ::= \mathit{true} \mid \mathit{false} \mid \neg T^{\mathbb{B}} \mid T^{\mathbb{B}} \wedge T^{\mathbb{B}} \mid T^{\mathbb{B}} \vee T^{\mathbb{B}} \\
 & \quad T^{\sigma} = T^{\sigma} \mid T^{\mathbb{S}(\sigma)} \subseteq T^{\mathbb{S}(\sigma)} \mid T^{\sigma} \in T^{\mathbb{S}(\sigma)} \mid T^{\mathbb{Z}} \le T^{\mathbb{Z}} \mid T^{\mathbb{R}} \le T^{\mathbb{R}} \\
T^{\mathbb{S}(\sigma)} & ::= X^{\mathbb{S}(\sigma)} \mid \{T^{\sigma} \mid_{\bar{x}} T^{\mathbb{B}}\} \mid T^{\mathbb{S}(\sigma)} \cup T^{\mathbb{S}(\sigma)} \mid T^{\mathbb{S}(\sigma)} \cap T^{\mathbb{S}(\sigma)} \mid T^{\mathbb{S}(\sigma)} \setminus T^{\mathbb{S}(\sigma)} \\
F & ::= T^{\mathbb{B}} \mid \exists x\, F \mid \exists X\, F
\end{array}$$

Fig. 1. Well-formed expressions in $\mathcal{T}^{\Sigma}$.

in the scope of a quantifier. In particular, $FV(\{t \mid_x \varphi\}) = (FV(t) \cup FV(\varphi)) \setminus \{x\}$, where $\mid_x$ is the comprehension quantifier. A term without free variables is a *closed* term. We write $t[x_0, \ldots, x_{n-1}]$ for a term t where each x_i may occur free in t. Let θ be the substitution $\{x_i \mapsto t_i\}_{i<n}$ (where x_i and t_i have the same sort)[4]; $t\theta$ denotes the application of θ on t. We write also $t[t_0, \ldots, t_{n-1}]$ for $t\theta$. For example, if $t[x]$ is the term $\mathit{Ite}(\{x \mid_x \varphi\} = \emptyset, x + x, x)$ and $\theta = \{x \mapsto x + y\}$ then $t\theta$ or $t[x+y]$ is the term $\mathit{Ite}(\{x \mid_x \varphi\} = \emptyset, (x+y) + (x+y), x+y)$.

We often omit the variables $\bar{x}$ from the comprehension quantifier $\mid_{\bar{x}}$ when they are clear from the context. We also use additional definitions in terms of $\mathcal{T}^{\Sigma}$ when they are needed. When a definition is obvious (such as $x < y$), we use it without further notice. *We often use the abbreviation $x.i$ for $\pi_i(x)$.*

A term in $\mathcal{T}^{\Sigma}$ of the form $\{x \mid x = t_1 \vee \cdots \vee x = t_n\}$ (where x is not free in any t_i), is abbreviated by $\{t_1, \ldots, t_n\}$ and is not considered as a comprehension term, but as an *explicit set* term.

2.2 Semantics

A *state* S is a mapping of variables to values. Since $\mathcal{T}^{\Sigma}$ is assumed to be the background we omit it from S, and assume that S has an implicit part that includes the interpretation for the function symbols of $\mathcal{T}^{\Sigma}$, for example that $+$ means addition and $\cup$ means set union. By slight abuse of notation, we reuse the function symbols in Figure 1 also to denote their interpretations, e.g., we write π_i also for $\pi_i^{\mathcal{T}^{\Sigma}}$, and let the context determine whether we refer to the symbol or its interpretation in $\mathcal{T}^{\Sigma}$. We write $\mathit{Dom}(S)$ for the domain of S. Given two states S_1 and S_2 we write $S_1 \uplus S_2$ for the union of S_1 and S_2 but where the variables in $\mathit{Dom}(S_1) \cap \mathit{Dom}(S_2)$ have the value in S_2.

[4] We always make the assumption that substitutions are well-sorted in this sense, without further notice.

A *state for* a term t is a state S such that $FV(t) \subseteq \mathit{Dom}(S)$. Given a term t and a state S for t, t^S is the *interpretation* or *evaluation of* t *in* S, defined by induction over the structure of t. Given a formula φ and a state S for φ, $S \models \varphi$ means that φ^S is *true*. Besides the standard logical connectives, arithmetical operations and set operations, equations (1–4) below show the semantics for the nonstandard constructions of t in Figure 1.

$$\begin{aligned}
\mathit{Ite}(\varphi, t_1, t_2)^S &= \begin{cases} t_1^S, \text{ if } S \models \varphi; \\ t_2^S, \text{ otherwise.} \end{cases} & (1)\\
\mathit{TheElementOf}(t_1^{\mathbb{S}(\sigma)})^S &= \begin{cases} a, & \text{if } t_1^S = \{a\}; \\ \mathit{Default}^\sigma, & \text{otherwise.} \end{cases} & (2)\\
\{t_0 \mid_{x^\sigma} \varphi\}^S &= \{t_0^{S \uplus \{x \mapsto a\}} : a \in \mathcal{U}^\sigma, S \uplus \{x \mapsto a\} \models \varphi\} & (3)\\
\Sigma_i(t_1)^S &= \sum_{a \in t_1^S} \pi_i(a) & (4)
\end{aligned}$$

The interpretation of a comprehension with several variables is a straightforward generalization of (3). In (3) it is assumed that there are only *finitely* many a such that $S \uplus \{x \mapsto a\} \models \varphi$, otherwise we may assume that $\{t_0 \mid_{x^\sigma} \varphi\}^S$ is $\emptyset$.[5] The use of comprehensions as terms is well-defined since sets are *extensional*: $\forall X\, Y\, (\forall z(z \in X \Leftrightarrow z \in Y) \Leftrightarrow X = Y)$.[6]

A state S for a formula φ such that $S \models \varphi$ is a *model* of φ. A formula φ is *satisfiable* if there exists a model of φ, and φ is *valid* if all states for φ are models of φ.

For a closed term t we talk about *evaluation of* t, without reference to any particular state.

Multiplication. We define $n * m$ with Σ_0, where $n > 0$ is an integer.

$$n * m \overset{\text{def}}{=} \Sigma_0(\{\langle m, x\rangle \mid 0 \le x < n\}) = \sum_{x=0}^{n-1} \pi_0(\langle m, x\rangle) = \sum_{x=0}^{n-1} m \qquad (5)$$

Note that m may be an integer or a real and the sort of m determines the sort of $n * m$. Thus, the projected sum operation Σ_i is very powerful and in the general case leads to undecidability of very restricted fragments of $\mathcal{T}^\Sigma$.

Bags. Bags or *multisets* are represented as graphs of maps with positive integer ranges, i.e., a bag b with elements $\{a_i\}_{i<n}$ each having *multiplicity* $m_i > 0$ in b for $i < n$, is represented as a set of pairs $\{\langle a_i, m_i\rangle\}_{i<n}$, thus having the sort $\mathbb{S}(\mathbb{T}(\sigma, \mathbb{Z}))$ for some basic sort σ called the *domain sort of* b. We let $\mathbb{M}(\sigma)$ be the type $\mathbb{S}(\mathbb{T}(\sigma, \mathbb{Z}^+))$ with the additional *map constraint*:

$$\forall X^{\mathbb{M}(\sigma)}\, \forall x^\sigma\, y^\sigma\, ((x \in X \wedge y \in X \wedge x.0 = y.0) \Rightarrow x.1 = y.1).$$

[5] In our translation from SQL to $\mathcal{T}^\Sigma$, finiteness is guaranteed by construction.

[6] Extensionality of sets is a meta-level property that is not expressible in $\mathcal{T}^\Sigma$.

We use the following definitions for dealing with bags.

$$\begin{aligned}
AsBag(Y^{\mathbb{S}(\sigma)}) &\overset{\text{def}}{=} \{\langle y, 1\rangle \mid y \in Y\}\\
AsSet(X^{\mathbb{M}(\sigma)}) &\overset{\text{def}}{=} \{y.0 \mid y \in X\}\\
\Sigma^{\mathrm{b}}_i(X^{\mathbb{M}(\mathbb{T}(\sigma_0,\ldots,\sigma_i,\ldots))}) &\overset{\text{def}}{=} \Sigma_0(\{\langle x.1 * x.0.i, x.0\rangle \mid x \in X\}) \quad (\sigma_i \text{ is numeric})
\end{aligned}$$

Intuitively $AsSet(X)$ eliminates the duplicates from X. Σ^{b}_i is a generalization of the projected sum over sets to bags. Note that $x.1$ above is always positive (thus, the use of $*$ is well-defined). Note that an expression like $X^{\mathbb{M}(\sigma)} \cup Y^{\mathbb{M}(\sigma)}$ is a well-formed expression in $\mathcal{T}^{\Sigma}$, but it does not preserve the type $\mathbb{M}(\sigma)$.

Example 1. Let $q[X^{\mathbb{M}(\mathbb{T}(\mathbb{Z},\mathbb{Z},\mathbb{Z}))}]$ be the following expression where $\varphi[x]$ is the formula $x < 4$.

$$q[X] = \{\langle x.0.0, \Sigma^{\mathrm{b}}_1(\{y \mid y \in X \wedge x.0.0 = y.0.0 \wedge \varphi[y.0.2]\})\rangle \mid x \in X \wedge \varphi[x.0.2]\}$$

Let $t = \{\langle\langle 0,2,1\rangle, 2\rangle, \langle\langle 1,2,3\rangle, 1\rangle, \langle\langle 1,2,4\rangle, 1\rangle\}$. Consider the evaluation of $q[t]$.

$$\begin{aligned}
q[t] &= \{\langle x.0.0, \Sigma^{\mathrm{b}}_1(\{y \mid y \in t \wedge x.0.0 = y.0.0 \wedge \varphi[y.0.2]\})\rangle \mid x \in t \wedge \varphi[x.0.2]\}\\
&= \{\langle 0, \Sigma^{\mathrm{b}}_1(\{y \mid y \in t \wedge 0 = y.0.0 \wedge \varphi[y.0.2]\})\rangle,\\
&\qquad \langle 1, \Sigma^{\mathrm{b}}_1(\{y \mid y \in t \wedge 1 = y.0.0 \wedge \varphi[y.0.2]\})\rangle\}\\
&= \{\langle 0, \textstyle\sum_{a \in \{\langle\langle 0,2,1\rangle,2\rangle\}} \pi_1(a) * \pi_1(\pi_0(a))\rangle,\\
&\qquad \langle 1, \textstyle\sum_{a \in \{\langle\langle 1,2,3\rangle,1\rangle\}} \pi_1(a) * \pi_1(\pi_0(a))\rangle\}\\
&= \{\langle 0, 4\rangle, \langle 1, 2\rangle\}
\end{aligned}$$

3 From SQL to $\mathcal{T}^{\Sigma}$

In this section we show how we translate a class of SQL queries into $\mathcal{T}^{\Sigma}$. We name the translation $\mathbf{Q} : \text{SQL} \to \mathcal{T}^{\Sigma}$. This section is less formal than Section 2. We omit full details of $\mathbf{Q}$ and illustrate it through examples and templates, that should be adequate for understanding how the general case works. Moreover, we restrict our focus to queries without side-effects and consider a subset of `SELECT` statements. We illustrate parts of the concrete grammar with simplified grammar fragments extracted from [1]. Queries that may cause deletion or addition of rows in the database are outside the scope of this paper. Also, queries that use `ORDER BY` are not handled here. In Section 6 we briefly discuss an extension of our approach for analyzing queries with side-effects, as ongoing and future work. In the general case, tables and results of queries are represented as bags whose domain sort is a tuple.

3.1 Data types

Typical databases use additional data types besides numbers and Booleans. In particular, *strings* are used in virtually every database. So how do we support them? There are two approaches to deal with this. One is to encode the data types in $\mathcal{T}^\Sigma$. The other one is to extend $\mathcal{T}^\Sigma$ with the corresponding sorts and background theories. In this paper we take the first approach. The main advantage is that we have a smaller core that we need to deal with in the context of analysis, that is discussed in Section 4. The main disadvantage is that the overhead of the encoding may be more expensive than using a built-in theory.

Strings. There are several ways how strings can be encoded in $\mathcal{T}^\Sigma$. Suppose that in a given column, all strings have a maximum length k; a possible encoding of a k-string is as a k-tuple of integers, where each character a is encoded as an integer $c(a)$ in the range $[1, 255]$. A further constraint associated with this encoding is that it has the form $\langle c(a_0), \ldots, c(a_l), 0, \ldots, 0\rangle$ for a string $a_0 \cdots a_l$ for $l < k$, and the empty string is the *Default* of the tuple sort. Operations over k-strings, such as extracting a substring, can then be defined in terms of tuple operations.

Commonly, a collection of strings D are used as enums in a given column (for example names of persons), and the only string operations that are relevant are equality and lexicographic ordering $\leq_{\mathrm{lex}}$ over strings in D. In this case one can define a bijection $f_D : D \to [0, |D| - 1]$ such that, for all $a, b \in D$, $a \leq_{\mathrm{lex}} b$ iff $f_D(a) \leq f_D(b)$, and encode strings in D as $|D|$-enums.

3.2 Nullable values

We encode nullable values with tuples. Given a basic sort σ, let $?\sigma$ be the sort $\mathbb{T}(\sigma, \mathbb{B})$ with the constraint $\forall x^{?\sigma}\, (x.1 = \mathit{false} \Rightarrow x.0 = \mathit{Default}^\sigma)$ and $\mathit{null}^{?\sigma} \stackrel{\mathrm{def}}{=} \mathit{Default}^{\mathbb{T}(\sigma,\mathbb{B})}$. Operations that are defined for σ are lifted to $?\sigma$. For example, for a numeric sort σ,

$$x^{?\sigma} + y^{?\sigma} \stackrel{\mathrm{def}}{=} \mathit{Ite}(x.1 \wedge y.1, \langle x.0 + y.0, \mathit{true}\rangle, \mathit{null}^{?\sigma}).$$

The projected sum operation is lifted analogously. The sorts $\mathbb{T}(\sigma, \mathbb{B})$ are not used to represent any other data types besides $?\sigma$. This encoding introduces an overhead for the symbolic analysis and is avoided unless the corresponding value type is declared nullable.

3.3 Query expressions

We consider top level query expressions that have the form *query_expr* according to the (simplified) grammar:

query_expr `::=` *select* `|` `(`*query_expr* *set_operation* *query_expr*`)`
set_operation `::=` `UNION | EXCEPT | INTERSECT`
select `::=` `SELECT [DISTINCT]` *select_list*
`FROM` *table_src* `[WHERE` *condition*`]` `[`*group_by_having*`]`

Set operations such as `UNION` remove duplicate rows from the arguments and the resulting query. In particular, the translation for `UNION` is:

$$\mathbf{Q}(\texttt{q1 UNION q2}) \stackrel{\mathrm{def}}{=} \mathit{AsBag}(\mathit{AsSet}(\mathbf{Q}(\texttt{q1})) \cup \mathit{AsSet}(\mathbf{Q}(\texttt{q2}))).$$

The other set operations have a similar translation.

3.4 Select clauses

A select clause refers to a particular selection of the columns from a given table by using a *select_list*. In the following translation we translate a *select_list* `l` into a sequence of projection indices $(l_0, \dots, l_n)$ on the table on which the selection is applied.

$$\mathbf{Q}(\texttt{SELECT l FROM t}) \overset{\text{def}}{=} \{\langle\langle x.0.l_0, \dots, x.0.l_n\rangle, M(x)\rangle \mid x \in \mathbf{Q}(\texttt{t})\} \qquad (6)$$

$$\text{where } M(x) = \Sigma_0(\{\langle y.1, y\rangle \mid y \in \mathbf{Q}(\texttt{t}) \wedge \bigwedge_{i=0}^{n} y.0.l_i = x.0.l_i\})$$

Note that multiplicities of the resulting tuples are computed separately, which is needed to preserve the type of the result as a bag. For example, the following is *not* a valid translation, unless `l` is `*`.

$$\{\langle\langle x.0.l_0, \dots, x.0.l_n\rangle, x.1\rangle \mid x \in \mathbf{Q}(\texttt{t})\} \quad \text{(this is not a bag in general!)}$$

If the `DISTINCT` keyword is used then duplicate rows are removed.

$$\mathbf{Q}(\texttt{SELECT DISTINCT l FROM t}) \overset{\text{def}}{=} \mathit{AsBag}(\mathit{AsSet}(\mathbf{Q}(\texttt{SELECT l FROM t})))$$

The following property is used in the set conversion:

$$\mathit{AsSet}(\mathbf{Q}(\texttt{SELECT l FROM t})) = \{\langle y.l_0, \dots, y.l_n\rangle \mid y \in \mathit{AsSet}(\mathbf{Q}(\texttt{t}))\} \qquad (7)$$

An optional `WHERE` condition is translated into a formula in $\mathcal{T}^{\Sigma}$ and appears as an additional condition in the above comprehensions.

3.5 Join operations

Join operations are used in `FROM` statements. In general, a `FROM` statement takes an argument *table_src*, that, in simplified form, has the grammar:

table_src `::=` *table_name* `[AS` *alias*`] |` *joined_table*
joined_table `::=` *table_src* *join* *table_src* `ON` *condition*
join `::= [{INNER | {{LEFT | RIGHT | FULL} [OUTER]}}] JOIN`

The condition may use column names of the (aliased) tables and operations on the corresponding data types. We only consider the case of `INNER JOIN`:

$$\mathbf{Q}(\texttt{t1 INNER JOIN t2 ON c}) \overset{\text{def}}{=} \qquad (8)$$
$$\{\langle x_1.0 \times x_2.0, x_1.1 * x_2.1\rangle \mid x_1 \in \mathbf{Q}(\texttt{t1}) \wedge x_2 \in \mathbf{Q}(\texttt{t2}) \wedge \mathbf{Q}(\texttt{c})[x_1.0, x_2.0]\}$$

where $\mathbf{Q}(\texttt{c})[y_1, y_2]$ denotes the translation of the condition `c` to the corresponding formula in $\mathcal{T}^{\Sigma}$, where the column names referring to the tables `t1` and `t2` occur as corresponding tuple projection operations on y_1 and y_2, respectively. The operation × is defined as follows, where x is an m-tuple and y is an n-tuple:

$$x \times y \overset{\text{def}}{=} \langle \pi_0(x), \dots, \pi_{m-1}(x), \pi_0(y), \dots, \pi_{n-1}(y)\rangle$$

The following property holds for the translation:

$$\mathit{AsSet}(\mathbf{Q}(\texttt{t1 INNER JOIN t2 ON c})) = \qquad (9)$$
$$\{y_1 \times y_2 \mid y_1 \in \mathit{AsSet}(\mathbf{Q}(\texttt{t1})) \wedge y_2 \in \mathit{AsSet}(\mathbf{Q}(\texttt{t2})) \wedge \mathbf{Q}(\texttt{c})[y_1, y_2]\}$$

3.6 Grouping and aggregates

A very common construct is the combined use of `GROUP BY` with *aggregate* operations. A *group_by_having* expression has the following (simplified) grammar, where a *group_by_item* for us is a column name.

group_by_having `::=` *group_by* `[HAVING` *condition*`]`
group_by `::= GROUP BY` *group_by_list*
group_by_list `::=` *group_by_item* `[ ,...n ]`

This expression appears in a *select* expression, the grammar of which is shown above, and there is a context condition that the columns in *select_list* that are not included in *group_by_list* must be applied to aggregate operations. The context condition is needed to eliminate duplicate rows produced by the select clause by combining the values in the columns not in the *group_by_list* into a single value for the given column. Here we only consider aggregates in combination with grouping.[7] The aggregate operations we consider are `SUM`, `COUNT`, `MAX`, `MIN`.

Example 2. Assume that `X` is a table with the columns `(A,B,C)` where each column has integer type. Consider the following query `q`.

```
SELECT A, SUM(B) AS D
FROM X
WHERE C < 4
GROUP BY A
```

$\mathbf{Q}(\texttt{q})$ is $\mathit{AsBag}(q[X])$ with $q[X]$ as in Example 1, where it is shown how

$$q\left[\begin{array}{|c|c|c|}\hline \mathtt{A}&\mathtt{B}&\mathtt{C}\\\hline\hline 0&2&1\\\hline 0&2&1\\\hline 1&2&3\\\hline 1&2&4\\\hline\end{array}\right] \text{ evaluates to } \begin{array}{|c|c|}\hline \mathtt{A}&\mathtt{D}\\\hline\hline 0&4\\\hline 1&2\\\hline\end{array}.$$

In order to simplify the presentation assume that *select_list* and *group_by_list* are like in Example 2. (Generalization is straightforward, but tedious.) The translation is as follows, where `t` is `SELECT a SUM(b) AS d FROM t1 WHERE c1`,

$$\mathbf{Q}(\texttt{t GROUP BY a HAVING c2}) \overset{\text{def}}{=} \mathit{AsBag}(\{z \mid z \in G \wedge \mathbf{Q}(\texttt{c2})[z]\})$$
$$\text{where } G = \{\langle x.0.0, \Sigma^{\mathrm{b}}_{1}(\{y \mid y \in \mathbf{Q}(\texttt{t}) \wedge y.0.0 = x.0.0\})\rangle \mid x \in \mathbf{Q}(\texttt{t})\}$$

Note that the condition $y.0.0 = x.0.0$ corresponds to *group_list*. Note also that `c2` is applied to the result G of the grouping and in the formula $\mathbf{Q}(\texttt{c2})[z]$, $z.0$ corresponds to `a` and $z.1$ corresponds to `d`. The other aggregates are translated similarly. For example, if `SUM(b)` is replaced by `COUNT(b)` then in the above

[7] In general, aggregates may also be used in a select expression without using grouping.

translation Σ^{b}_1 is replaced by $\mathit{Count} \overset{\mathrm{def}}{=} \Sigma_1$. For `MIN` and `MAX` the projected sum operation is not needed, for example:

$$\mathit{Min}(X^{\mathbb{S}(\sigma)}) \overset{\mathrm{def}}{=} \mathit{TheElementOf}(\{y \mid y \in X \wedge \{z \mid z \in X \wedge z < y\} = \emptyset\}) \quad (10)$$

Although we do not consider the aggregate `AVG` here, it can be translated as $\Sigma^{\mathrm{b}}_i(X) \div \mathit{Count}(X)$, where $\div$ is division by positive integer in $\mathbb{R}$ and can be defined as follows:

$$r \div k \overset{\mathrm{def}}{=} \mathit{TheElementOf}(\{x^{\mathbb{R}} \mid k * x = r\}). \quad (11)$$

3.7 Simplifications

Many operations convert bags into sets. There are certain further simplification rules, besides (7) and (9), that are based on the following properties between bag an set operations and are used in the translation to reduce operations over bags to operations over sets, whenever possible.

$$\begin{aligned} \mathit{AsSet}(\mathit{AsBag}(X^{\mathbb{S}(\sigma)})) &= X \\ \Sigma^{\mathrm{b}}_i(\mathit{AsBag}(X^{\mathbb{S}(\sigma)})) &= \Sigma_i(X) \\ \mathit{AsSet}(\{t \mid \varphi\}^{\mathbb{M}(\sigma)}) &= \{t.0 \mid \varphi\} \end{aligned}$$

Moreover, further simplifications are done at the level of basic sorts, such as $\pi_i(\langle t_0, \ldots, t_i, \ldots \rangle) = t_i$, that are also used as part of the simplification process. More accurately, the simplifications are part of an equivalence preserving post processing phase of $\mathbf{Q}(\mathtt{q})$ for a given query `q`.

4 Model generation with SMT

Translation $\mathbf{Q}$ leads to a subclass of expressions in $\mathcal{T}^{\Sigma}$, denoted by $\mathcal{T}^{\Sigma}_{\mathbf{Q}}$. The core problem we are interested in is *model generation* in $\mathcal{T}^{\Sigma}_{\mathbf{Q}}$.

Definition 1 (Model Generation in $\mathcal{T}^{\Sigma}_{\mathbf{Q}}$). Given a quantifier free formula $\varphi[X]$ in $\mathcal{T}^{\Sigma}$, and a query q, decide if $\psi = \varphi[\mathbf{Q}(q)]$ is satisfiable, and if ψ is satisfiable generate a model of ψ.

Our main application is to generate a database for a given query such that the query satisfies a certain property. In general a query may also include *parameters*, other than the input tables, e.g., in Example 2, the constant `4` can be replaced by a parameter variable `@x`.[8] Thus, one can use model generation for parameter generation as well as database generation, given a (partially) fixed database and a parameterized query `q`, generate a model of $\varphi[\mathbf{Q}(\mathtt{q})]$, where φ represents a test criterion (such as the result being nonempty). Once a model is generated, it is used to generate a concrete unit test, see Section 5.

[8] Parameters are prefixed with the `@` sign in the concrete query language we are using.

For model generation we use the state of the art SMT solver Z3 [24, 11]. For bags and sets we use the built-in theory of extensional arrays in Z3, similarly for tuples, Booleans, integers and reals. In some cases the formula $\varphi[\mathbf{Q}(\mathtt{q})]$ can be first simplified, e.g., so that all bags are reduced to sets. Below we describe the general mechanism without emphasis on such simplifications.

4.1 Eager expansion

Consider a formula $\psi[\overline{X}]$ as an instance of the model generation problem, where every X in $\overline{X}$ is a bag variable. The formula ψ may include other free variables that correspond to parameter variables in the original query. For the analysis, we introduce a special inductively defined term called a *set describer*, with the sort $\mathbb{S}(\sigma)$.

- The constant $\mathit{Empty}^{\mathbb{S}(\sigma)}$ is a set describer.
- If $t^{\mathbb{S}(\sigma)}$ is a set describer then so is the term $\mathit{Set}(\varphi^{\mathbb{B}}, u^{\sigma}, t)$.

Given a state S for $\mathit{Set}(\varphi, u, t)$, the interpretation in S is,

$$\mathit{Set}(\varphi, u, t)^S = \mathit{Ite}(\varphi, \{u\}, \emptyset)^S \cup t^S, \quad \mathit{Empty}^S = \emptyset.$$

Consider a fixed X in $\overline{X}$ and let t_X be the set describer

$$\mathit{Set}(\mathit{true}, \langle x_1, m_1\rangle, \ldots \mathit{Set}(\mathit{true}, \langle x_k, m_k\rangle, \mathit{Empty}) \ldots)$$

where k and all the m_j's are some positive integer constants and each x_i is a variable. Thus, t_X describes the set $\{\langle x_1, m_1\rangle, \ldots, \langle x_k, m_k\rangle\}$. It is also assumed that there is an associated constraint $\mathit{distinct}(x_1, \ldots, x_k)$ stating that all the x_i's are pairwise distinct. Thus t_X is a valid bag term, in any context where the constraint holds.

The *expansion* of $\psi[\overline{t_X}]$, $\mathbf{Exp}(\psi[\overline{t_X}])$, eliminates comprehensions and projected sums from $\psi[\overline{t_X}]$. The definition of $\mathbf{Exp}$ is by induction over the structure of terms. The case of comprehensions is as follows. Here we assume that the comprehension has a single bound variable, the definition is straightforward to generalize to any number bound variables. It is also assumed here that the comprehension has a special form where the bound variable x has a *range expression* $x \in r$ where x is not free in r.

$$\begin{aligned}
\mathbf{Exp}(\{t \mid_x x \in r \wedge \varphi\}) &\stackrel{\text{def}}{=} \mathbf{ExpC}(t, x, \mathbf{Exp}(r), \varphi) \\
\mathbf{ExpC}(t, x, \mathit{Empty}, \varphi) &\stackrel{\text{def}}{=} \mathit{Empty} \\
\mathbf{ExpC}(t[x], x, \mathit{Set}(\gamma, u, \mathit{rest}), \varphi[x]) &\stackrel{\text{def}}{=} \mathit{Set}(\gamma \wedge \mathbf{Exp}(\varphi[u]), \mathbf{Exp}(t[u]), \\
&\qquad \mathbf{ExpC}(t, x, \mathit{rest}, \varphi))
\end{aligned}$$

Not all comprehensions are expanded this way, some expressions use specialized expansion rules. For example, for (10), $\mathbf{Exp}(\mathit{Min}(t))$ is replaced by a fresh variable x and the formula

$$\mathit{Ite}\left(\mathbf{Exp}(t) \neq \emptyset, (\mathbf{IsLeq}(x, \mathbf{Exp}(t)) \wedge x \in \mathbf{Exp}(t)), x = 0\right),$$

which is equivalent to $x = \mathit{Min}(t)$, that is included as a top-level conjunct (in $\mathbf{Exp}(\psi[\boxed{t_X}])$),[9] where

$$\begin{aligned}\mathbf{IsLeq}(x, \mathit{Empty}) &\overset{\text{def}}{=} \mathit{true}\\ \mathbf{IsLeq}(x, \mathit{Set}(\varphi, u, r)) &\overset{\text{def}}{=} (\varphi \Rightarrow x \leq u) \wedge \mathbf{IsLeq}(x, r)\end{aligned}$$

For Σ_i the expansion is as follows.

$$\begin{aligned}\mathbf{Exp}(\Sigma_i(t)) &\overset{\text{def}}{=} \mathbf{Sum}_i(\mathbf{Exp}(t), \mathit{Empty})\\ \mathbf{Sum}_i(\mathit{Empty}, s) &\overset{\text{def}}{=} 0\\ \mathbf{Sum}_i(\mathit{Set}(\gamma, u, \mathit{rest}), s) &\overset{\text{def}}{=} \mathit{Ite}(\gamma \wedge u \notin s, \pi_i(u), 0) + \mathbf{Sum}_i(\mathit{rest}, \mathit{Set}(\gamma, u, s))\end{aligned}$$

Note that the role of s is to accumulate elements that have already been included in the sum, so that the same element is not added twice.

Regarding multiplication, the general form of (5), that involves a comprehension without a range expression, is not needed. Since all multiplicities in the initial tables t_X are fixed constants, it follows that multiplications are either of the form $k_1 * k_2$, where k_1 and k_2 are constants (in formulas created in (8)), which preserves the constant multiplicities in the resulting table), or multiplicities are finite sums of constants (as in (6)), which provides constant upper and lower bounds for the multiplicities. Multiplication under these constraints is supported in Z3.

It is also possible to expand $t \div u$ as defined in (11), by replacing $\mathbf{Exp}(t \div u)$ with a fresh variable $x^{\mathbb{R}}$ and adding the top-level conjunct $\mathbf{Exp}(u) * x = \mathbf{Exp}(t)$. Here $\mathbf{Exp}(u)$ is also a sum of terms that have constant upper and lower bounds.

The overall approach amounts to systematically enumerating the sizes of the tables and the multiplicities, and searching for a model of the resulting expanded formula.

4.2 Lazy expansion

The main disadvantage of the eager approach is that it expands all terms upfront, without taking into account if a certain expansion is actually needed in a particular context. An alternative (or complementary) approach is to delay the expansion of (some) terms by delegating the expansion to the proof search engine of the underlying solver. We explain here a high-level view of how to accomplish such delayed or *lazy* expansion in the context of SMT.

In addition to a quantifier free formula ψ that is provided to the SMT solver and for which proof of satisfiability is sought, one can also provide additional universally quantified *axioms*. During proof search, axioms are triggered by matching subexpressions in ψ. An axiom has the form

$$(\forall \bar{x}(\alpha),\ \mathit{pat}_{\alpha})$$

[9] Note that a formula $\varphi[t]$ is equivalent to the formula $\exists x(\varphi[x] \wedge x = t)$, where x is a fresh variable.

where α is a quantifier free formula, pat_α is a quantifier free term, and $FV(\alpha) = FV(pat_\alpha) = \bar{x}$. The axioms typically define properties of uninterpreted function symbols in an extended signature. The high-level view behind the use of the axioms is as follows. If ψ contains a subterm t and there exists a substitution θ such that $t = pat_\alpha\theta$, i.e., *t matches the pattern pat_α*, then ψ is replaced during proof search by (a reduction of) $\psi \wedge \alpha\theta$.[10] Note that, if a pattern is never matched in this way, the use of the corresponding axiom is not triggered. Thus, the use of axioms is inherently incomplete, and it is not guaranteed that the axioms hold in a model of ψ, if one is found, or even if the axioms are consistent.

We illustrate the use of axioms with the projected sum operator. Assume that *Empty*, *Set*, and $\mathbf{Sum}_i$ are new function symbols and assume that we have the following axioms:

$$\begin{aligned}
\alpha_1 &= \forall s(\mathbf{Sum}_i(\mathit{Empty}, s) = 0)\\
pat_{\alpha_1} &= \mathbf{Sum}_i(\mathit{Empty}, s)\\
\alpha_2 &= \forall b\, u\, r\, s\, (\mathbf{Sum}_i(\mathit{Set}(b, u, r), s) =\\
&\qquad\qquad \mathit{Ite}(b \wedge u \notin s, \pi_i(u), 0) + \mathbf{Sum}_i(r, \mathit{Ite}(b, \{u\}, \emptyset) \cup s))\\
pat_{\alpha_2} &= \mathbf{Sum}_i(\mathit{Set}(b, u, r), s)
\end{aligned}$$

Note that, unlike we defined $\mathbf{Sum}_i$ in Section 4.1, the argument s here is not a set describer, but a set valued term that has built-in interpretation in the SMT solver.[11] Let us consider an example reduction, let ψ_0 be the formula:

$$x \leq \mathbf{Sum}_1(\mathit{Set}(\mathit{true}, \langle 1, y\rangle, \mathit{Set}(\mathit{true}, \langle 1, z\rangle, \mathit{Empty})), \emptyset)$$

The right hand side of ψ_0 matches pat_{α_2}, so ψ_0 reduces to ψ_1:[12]

$$x \leq y + \mathbf{Sum}_1(\mathit{Set}(\mathit{true}, \langle 1, z\rangle, \mathit{Empty}), \{\langle 1, y\rangle\})$$

The same axiom is applied again, and ψ_1 is reduced to ψ_2:

$$x \leq y + \mathit{Ite}(z \neq y, z, 0) + \mathbf{Sum}_1(\mathit{Empty}, \{\langle 1, y\rangle, \langle 1, z\rangle\})$$

Finally, α_1 is used to reduce ψ_2 to $x \leq y + \mathit{Ite}(z \neq y, z, 0)$. A concrete, self-contained, and simplified example using the smt-lib format [19] is also shown in the appendix.

In general, such axioms can be defined for expanding other constructs. The main tradeoff is whether the additional overhead of the axiomatization of the expansion rules and the loss of completeness pays off. One also has to take into account that in the intended application, discussed in Section 5, we are mostly interested in generating small databases.

[10] In general, one can associate several patterns with an axiom, one of which is used for triggering, and one can also use multi-patterns in Z3. A multi-pattern is a collection of patters all of which must be matched for the axiom to be triggered.

[11] It is not possible to pattern-match against built-in operations in Z3.

[12] To be precise the reduction takes several steps that are skipped here.

5 Application to unit testing

Returning to the main motivation behind this work, we are primarily interested in the problem of generating a database (a collection of tables) and concrete parameters for a given parameterized query that, when evaluated with respect to the database, satisfies a certain test criterion. Examples of standard test criteria are: the answer is empty, the answer is nonempty, and the answer contains a given number of (distinct) rows.

We are abstracting here from the problem of determining what exactly are the intended domains of the values in a column, e.g., a certain column may be declared to have the string type, but effectively the strings are used as enums. In fact, the particular encoding of the domain values depends of the query. We suppose that we have domain specific functions, that enable us to map models generated by the analysis engine, to corresponding concrete tables and parameter values for the query, e.g., that the value 12 in a certain column corresponds to the string "Bob". See also Section 3.1.

With this encoding in mind, we view the analysis engine here as a black box, called *Qex*, which given a parameterized query, produces a set of tables and parameters to that query. In the following we look at some examples and illustrate a concrete application of Qex in the context of generating database unit tests in Visual Studio.

Experiments. We consider here a sample database for an online store that contains tables for products, orders and customers; products have a product id, a name and a price; customers have a customer id and a name; orders have an order id and a customer id. Figure 2 illustrates some sample queries over the database. Query `q1` selects customers and related orders based on a constraint on the ids. Query `q2` selects those customers and corresponding number of orders, who have more than one order. Query `q3` selects "good" customers and has a parameter named `@value`. Table 1 shows some performance measures of model generation for different input table sizes and test conditions for the queries in Figure 2 using the eager expansion. The total evaluation time is divided into *expansion time* t_{exp} and *proof search time* t_{z3} with Z3. The current prototype implementation of the eager expansion algorithm is unoptimized and uses a naive representation of terms in $\mathcal{T}^{\Sigma}$ without structure sharing, e.g., the size of the expanded term $\mathbf{Q}(\mathtt{q2})$ for $k = 3$ is over 5 million symbols. This is reflected by the fact that in most cases $t_{\mathrm{exp}} \gg t_{\mathrm{z3}}$, although the actual parameter and table generation takes place during proof search. Note that t_{exp} is independent of the test condition, whereas t_{z3} clearly depends on it. In general, exhaustive search for models, in the case when the formula is unsatisfiable, is more time consuming than when a model exists. Note also that query `q2` is unsatisfiable with 1 row in each input table due to the condition `Count(O.OrderID) > 1`. The actual tables and parameters generated for `q3` using Pex [18] integration in Visual Studio Database edition are illustrated by a screenshot in Figure 3. The integration also generates automatically a unit test that can be accessed using the `Go To` button in Fig-

```
q1: SELECT C.CustomerID, O.OrderID
    FROM Orders AS O
    JOIN Customers AS C ON
         O.CustomerID = C.CustomerID
    WHERE O.CustomerID > 2 AND
          O.OrderID < 15
```

```
q2: SELECT C.CustomerID,
           Count(O.OrderID)
    FROM Orders AS O
    JOIN Customers AS C ON
         O.CustomerID = C.CustomerID
    GROUP BY C.CustomerID HAVING
             Count(O.OrderID) > 1
```

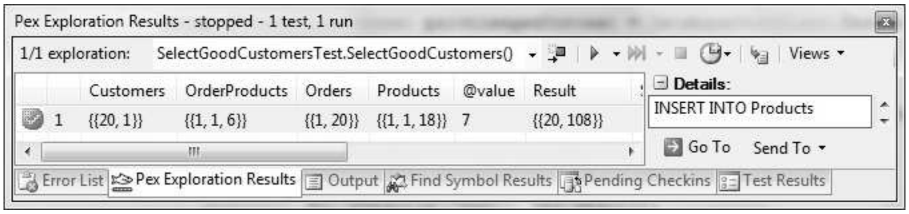
```
q3: DECLARE @value AS INT;
    SELECT C.CustomerID, SUM(OP.OrderProductQuantity * P.ProductPrice)
    FROM OrderProducts AS OP
    JOIN Orders AS O ON OP.OrderID = O.OrderID
    JOIN Products AS P ON OP.ProductID = P.ProductID
    JOIN Customers AS C ON O.CustomerID = C.CustomerID
    WHERE @value > 1
    GROUP BY C.CustomerID
    HAVING SUM(OP.OrderProductQuantity * P.ProductPrice) > 100 + @value
```

Fig. 2. Sample queries.

Table 1. Model generation for sample queries. Evaluation times t_{exp} and t_{z3} are given in seconds; k is the expected number of rows in each of the generated input tables; all multiplicities of rows in input tables are 1.

query	condition	k	check	t_{exp}	t_{z3}
q1	$res \neq \emptyset$	1	sat	.03	.001
		2	sat	.05	.005
		3	sat	.3	.02
		4	sat	1.4	.13
	$res = \emptyset$	1	sat	.03	.001
		2	sat	.05	.006
		3	sat	.3	.12
		4	sat	1.4	2

query	condition	k	check	t_{exp}	t_{z3}
q1	$\lvert res\rvert = 5$	1	unsat	.03	.001
		2	unsat	.05	.01
		3	unsat	.3	.16
		4	unsat	1.4	10
q2	$res \neq \emptyset$	1	unsat	.03	.001
		2	sat	.7	.006
		3	sat	26	.03
q3	$res \neq \emptyset$	1	sat	.34	.001



Pex Exploration Results - stopped - 1 test, 1 run

1/1 exploration: SelectGoodCustomersTest.SelectGoodCustomers()

	Customers	OrderProducts	Orders	Products	@value	Result
1	{{20, 1}}	{{1, 1, 6}}	{{1, 20}}	{{1, 1, 18}}	7	{{20, 108}}

Details:

INSERT INTO Products

Fig. 3. Screenshot of model generation through Pex integration of Qex in Visual Studio.

ure 3 and executed against the actual database, which in this case is provided through MS SQL server 2005.

6 Extensions

The Qex project is a new project that has some flavor of model-based testing as well as parameterized unit testing. The current implementation is a prototype that needs further evaluation and case studies. The approach can also be extended in several ways. The choice of the background theory $\mathcal{T}^{\Sigma}$ was partly motivated with some of those extensions in mind. Here we discuss a few of the extensions that are ongoing and future work. Further directions for research are discussed in Section 7.

Side-effects. The background theory $\mathcal{T}^{\Sigma}$ is an extension of the background theory $\mathcal{T}$ with the projected sum operator and restricted to finite sets; $\mathcal{T}$ is used for symbolic *model program* analysis [3, 23] by reduction to SMT solving. The projected sum operation has not been considered in that context; one reason is that it causes undecidability of some fragments that are otherwise decidable. In principle though, model programs can also be based on the background $\mathcal{T}^{\Sigma}$. A model program can be used to describe an evaluation of a query together with side-effects, where the side-effects are computed as update sets to respective tables that are applied at the end of the evaluation in a single transaction. In this setting one can also symbolically analyze the resulting model program for potential *update inconsistency* [3].

Data types. Another extension is better support for data types that are in the current approach encoded with tuples. This encoding is not fully adequate for supporting commonly used algebraic data types such as trees and lists, or terms in the sense of a free-algebra with a separate sort. The encoding of such data types in $\mathcal{T}^{\Sigma}$ is both expensive and incomplete (from the analysis point of view). Also, one can adopt existing techniques to represent strings, and solve constraints involving common operations over strings, in the context of an SMT solver [4].

Integration with parameterized unit testing of code. From the practical perspective, more complex unit tests, used for testing store procedures, may use a combination of queries and code. It is possible to combine parameterized unit testing of managed code [21] with query evaluation discussed in this paper.

7 Related work

Deciding satisfiability of SQL queries requires a formal semantics. While we give meaning to SQL queries by an embedding into our formal background theory $\mathcal{T}^{\Sigma}$, which is in turn mapped to a logic of an SMT solver, there are other approaches, e.g., defining the semantics in the Extended Three Valued Predicate

Calculus [17], or using bags as a foundation [8]. Satisfiability of queries is also related to logic-based approaches to semantic query optimization [6]. The general problem of satisfiability of SQL queries is undecidable and computationally hard for very restricted fragments, e.g., deciding if a query has a nonempty answer is NEXP-hard for nonrecursive range-restricted queries [10].

Several research efforts have considered formal analysis and verification of aspects of database systems, usually employing a possibly interactive theorem prover. For example, one system [20] checks whether a transaction is guaranteed to maintain integrity constraints in a relational database; the system is based on Boyer and Moore-style theorem proving [5].

There are many existing approaches to generate databases as test inputs. Most approaches create data in an ad-hoc fashion. Only few consider a target query. Tsai et.al. present an approach for test input generation for relational algebra queries [22]. They do not consider comprehensions or bags. They propose a translation of queries to a set of systems of linear inequalities, for which they implemented an ad-hoc solving framework which compares favorably to random guessing of solutions. A practical system for testing database transactions is AGENDA [12]. It generates test inputs satisfying a database schema by combining user-provided data, and it supports checking of complex integrity constraints by breaking them into simpler constraints that can be enforced by the database. While this system does not employ a constraint solver, it has been recently refined with the TGQG [7] algorithm: Based on given SQL statements, it generates test generation queries; execution of these queries against a user-provided set of data groups yields test inputs which cover desired properties of the given SQL statements.

Some recent approaches to test input generation for databases employ automated reasoning. The relational logic solver Alloy [14, 15] has been used by Khalek et.al. [16] to generate input data for database queries. Their implementation supports a subset of SQL with a simplified syntax. In queries, they can reason about relational operations on integers, equality operations on strings, and logical operations, but not about nullable values, or grouping with aggregates such as `SUM`; they also do not reason about duplicates in the query results. QAGen [2] is another approach to query-solving. It first processes a query in an adhoc-way, which requires numerous user-provided "knob" settings as additional inputs. From the query, a propositional logic formula is generated, which is then decided by the Cogent [9] solver to generate the test inputs. Recently, test input generation of queries has been combined with test input generation of programs that contain embedded queries in the program text [13], using ad-hoc heuristic solvers for some of the arising constraints from the program and the queries.

References

1. SELECT (T-SQL). http://msdn.microsoft.com/en-us/library/ms189499.aspx.
2. C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. Qagen: generating query-aware test databases. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD inter-*

national conference on Management of data, pages 341–352, New York, NY, USA, 2007. ACM.

3. N. Bjørner, Y. Gurevich, W. Schulte, and M. Veanes. Symbolic bounded model checking of abstract state machines. Technical Report MSR-TR-2009-14, Microsoft Research, February 2009. Submitted to IJSI.
4. N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 307–321. Springer, 2009.
5. R. S. Boyer and J. S. Moore. *A computational logic handbook*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.
6. U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Trans. Database Syst.*, 15(2):162–207, 1990.
7. D. Chays, J. Shahid, and P. G. Frankl. Query-based test generation for database applications. In *DBTest '08*, pages 1–6, New York, NY, USA, 2008. ACM.
8. H. R. Chinaei. An ordered bag semantics of SQL. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, 2007.
9. B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In *Proceedings of CAV 2005, volume 3576 of Lecture Notes in Computer Science*, pages 296–300. Springer, 2005.
10. E. Dantsin and A. Voronkov. Complexity of query answering in logic databases with complex values. In *LFCS '97*, pages 56–66, London, UK, 1997. Springer.
11. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'08)*, LNCS. Springer, 2008.
12. Y. Deng, P. Frankl, and D. Chays. Testing database transactions with AGENDA. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 78–87, New York, NY, USA, 2005. ACM.
13. M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA '07*, pages 151–162, New York, NY, USA, 2007. ACM.
14. D. Jackson. Automating first-order relational logic. *SIGSOFT Softw. Eng. Notes*, 25(6):130–139, 2000.
15. D. Jackson. *Software Abstractions*. MIT Press, 2006.
16. S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid. Query-aware test generation using a relational constraint solver. In *ASE*, pages 238–247, 2008.
17. M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of SQL queries. *ACM Transactions on Database Systems*, 17(3):513–534, September 1991.
18. Pex. http://research.microsoft.com/projects/pex.
19. S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.
20. T. Sheard and D. Stemple. Automatic verification of database transaction safety. *ACM Trans. Database Syst.*, 14(3):322–368, 1989.
21. N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153, Prato, Italy, April 2008. Springer.
22. W. T. Tsai, D. Volovik, and T. F. Keefe. Automated test case generation for programs specified by relational algebra queries. *IEEE Trans. Softw. Eng.*, 16(3):316–324, 1990.
23. M. Veanes and N. Bjørner. Symbolic bounded conformance checking of model programs. In *PSI'09*, LNCS, 2009. Extended version available as Microsoft Research Technical Report, MSR-TR-2009-28.
24. Z3. http://research.microsoft.com/projects/z3.

A SMT-LIB samples

The following samples illustrate some of the principles behind the encoding of the expansion rules discussed in Section 4.2 as axioms in Z3. The examples are written using the standard smt-lib format [19] and therefore do not use built-in Z3 support for sets, tuples, etc. In order to execute the samples and to inspect the result, copy and paste them into text files and use the command line tool:

```
> z3.exe <filename> /m
```

Z3 can be downloaded from [24]. Note that, when universally quantified assumptions (axioms) are used, Z3 always answers `unknown` (rather that `sat` or `unsat`) due to incompleteness of the use of axioms. The option `/m` prints the generated model of the formula for which satisfiability is checked.

A.1 Projected sum

The function `Sum` represents a simplified version of the projected sum operation. The generated model of the below formula yields $res = 12$.

```
(benchmark sum_test
:logic Ints
:extrasorts (Z3Set)
:extrafuns ((EmptyZ3Set Z3Set) (Store Int Z3Set Z3Set))
:extrapreds ((NotIn Int Z3Set))
:extrasorts (SetDescriber)
:extrafuns ((Empty SetDescriber) (Set bool Int SetDescriber SetDescriber))
:extrafuns ((Sum SetDescriber Z3Set Int))

;definition of NotIn
:assumption (forall (i Int)
             (iff (NotIn i EmptyZ3Set) true)
            :pat{(NotIn i EmptyZ3Set)}
            )
:assumption (forall (i Int) (s Z3Set)
             (iff (NotIn i (Store i s)) false)
            :pat{(NotIn i (Store i s))}
            )
:assumption (forall (i Int) (j Int) (s Z3Set)
             (implies (not (= i j))
                      (iff (NotIn i (Store j s))
                           (NotIn i s)))
            :pat{(NotIn i (Store j s))}
            )
;definition of Sum
:assumption (forall (s Z3Set)
             (= (Sum Empty s) 0)
             :pat{(Sum Empty s)}
            )
:assumption (forall (r SetDescriber) (b bool) (u Int) (s Z3Set)
             (= (Sum (Set b u r) s) (+ (ite (and b (NotIn u s)) u 0) (Sum r (Store u s))))
             :pat{(Sum (Set b u r) s)}
            )
:extrafuns ((x Int) (y Int) (z Int) (res Int))
:extrafuns ((b bool))
:assumption (and (> x 0) (> y 0) (> z 0) (> res 10))

;the actual formula being checked
:formula (= res (Sum (Set (< x y) 3 (Set b 5 (Set (< y z) 4
                     (Set (< z x) 4 (Set b 5 Empty))))) EmptyZ3Set))
)
```

A.2 Join

The function Join represents a function that produces a list of pairs from a list of singleton tuples. The sample represents a simplified version of expanding a comprehension term that represents a join of two tables. Given (tables) $l_1 = \{\langle 1\rangle, \langle 2\rangle, \langle 3\rangle\}$ and $l_2 = \{\langle 5\rangle, \langle 6\rangle\}$,

$$\texttt{Join}(l_1, l_2) = \{\langle 1,5\rangle, \langle 1,6\rangle, \langle 2,5\rangle, \langle 2,6\rangle, \langle 3,5\rangle, \langle 3,6\rangle\}.$$

```
(benchmark join_test
:logic Ints

:extrasorts (Tuple1)
:extrafuns ((T1 Int Tuple1))
:extrafuns ((Elem1 Tuple1 Int))
:assumption (forall (i Int)
             (= (Elem1 (T1 i)) i)
             :pat{(Elem1 (T1 i))}
            )

:extrasorts (Tuple2)
:extrafuns ((T2 Int Int Tuple2))
:extrasorts (List1)
:extrafuns ((Nil1 List1) (Cons1 Tuple1 List1 List1))

:extrasorts (List2)
:extrafuns ((Nil2 List2) (Cons2 Tuple2 List2 List2))

:extrafuns ((Join List1 List1 List2))
:extrafuns ((Join1 Tuple1 List1 List1 List1 List2))

;definition of Join
:assumption (forall (x List1)
             (= (Join x Nil1) Nil2)
            :pat{(Join x Nil1)}
            )
:assumption (forall (t Tuple1) (x List1) (y List1)
             (implies (not (= y Nil1))
                      (= (Join (Cons1 t x) y)
                          (Join1 t x y y)))
            :pat{(Join (Cons1 t x) y)}
            )

;definition of helper function Join1
:assumption (forall (t Tuple1) (x List1) (y List1)
             (= (Join1 t x Nil1 y)
                (Join x y))
            :pat{(Join1 t x Nil1 y)}
            )
:assumption (forall (t Tuple1) (u Tuple1) (x List1) (y List1) (z List1)
             (= (Join1 t x (Cons1 u y) z)
                (Cons2 (T2 (Elem1 t) (Elem1 u))
                       (Join1 t x y z)))
            :pat{(Join1 t x (Cons1 u y) z)}
            )

:extrafuns ((l List2))
:extrafuns ((l1 List1) (l2 List1))
:formula (and (= l1 (Cons1 (T1 1) (Cons1 (T1 2) (Cons1 (T1 3) Nil1))))
              (= l2 (Cons1 (T1 5) (Cons1 (T1 6) Nil1)))
              (= l (Join l1 l2)))
)
```