Automatic Generation of Database Instances for White-box Testing

Jian Zhang, Chen Xu
Laboratory of Computer Science, Institute of Software
Chinese Academy of Sciences, Beijing 100080, China

S.-C. Cheung
Department of Computer Science
Hong Kong University of Science and Technology

Abstract

Testing is a critical activity for database application programs as faults if undetected could lead to unrecoverable data loss. Database application programs typically contain statements written in an imperative programming language with embedded data manipulation commands, such as SQL. However relatively little study has been made in the testing of database application programs. In particular, few testing techniques explicitly consider the inclusion of database instances in the selection of test cases and the generation of test data input. In this paper, we study the generation of database instances that respect the semantics of SQL statements embedded in a database application program. The paper also describes a supporting tool which generates a set of constraints. These constraints collectively represent a property against which the program is tested. Database instances for program testing can be derived by solving the set of constraints using existing constraint solvers.

Keywords: Database applications, embedded SQL, constraint solving, automatic test data generation, software testing

1 Introduction

Database application programs play an important role in commercial systems. These programs interact with a database system to realize a predefined logic in manipulating business data. While database application programs realize application logic in some host language [6], such as C, C++ or Java, database systems provide mechanisms for efficient access to and manipulation of massive volume of data. Database programs are often expected to exhibit high reliability. Faults, if occurring in a database program, can result in unrecoverable data corruption. Since not all database transactions can be unrolled, restoring data from backups cannot eradicate the errors. However relatively little study has been made in the testing of database application programs. In particular, few testing techniques explicitly consider the automatic generation of database instances.

Previously, Chan and Cheung [2] proposed an approach to the testing of database applications. Its basic idea is to transform the embedded SQL statements into classical program fragments (e.g., statements in C). This allows software testing personnel to test the database applications using conventional program testing techniques (like path-based white-box testing methods) [7, 1]. More recently, Chays *et al.* [3] proposed a framework for testing database applications. They also described a semi-automatic tool which includes an SQL parser. However, user interaction is needed when generating the tables.

In this paper, we present a fault-based approach to the generation of test data for database application programs. More specifically, we study the test data generation to support white-box testing of embedded SQL programs.

2 Basics of SQL

In this section, we overview some basic concepts of database application programs. Readers are referred to [4, 9] for more details.

2.1 The Relational Model

Existing database management systems are mostly relational. In such a model, the data are stored as a set of relations, usually in *table* forms. Each row of the table represents an entity (called a *tuple*), and the columns denote

0-7695-1372-7/01 $10.00 © 2001 IEEE

eid	*name*	*age*	*gender*	*position*
101	John	45	male	president
112	Mary	32	female	secretary
105	Kate	38	female	manager

Table 1. Information about Employees

its *attributes*. An attribute can take a value from some *domain*. For instance, the attribute *age* at Table 1 is an integer within the interval [0..200]. Besides the ordinary values specified in the domain, an attribute can also take a special value called NULL. Essentially it indicates the absence of valid information.

Example 1. An *Employee* table may have the attributes *eid*, *name*, *age*, *gender*, *position*, etc. Each row gives information about a particular employee. A specific *Employee* table is given as Table 1.

The relational model is closely related to the first-order predicate calculus. It is known that the relational algebra with the basic operators such as union and join has the same expressive power as first-order predicate logic [4]. In this paper, we focus on a subset of SQL and formulate it using the many-sorted predicate calculus. In the calculus, a table is treated as a sort which consists of all the rows of the table. Each column corresponds to a function. Let us look at Table 1. We can consider *Employee* as a set of three elements: e_1, e_2 and e_3. The functions are defined as follows:

$$eid(e_1) = 101,\ eid(e_2) = 112,\ eid(e_3) = 105;$$
$$age(e_1) = 45,\ age(e_2) = 32,\ age(e_3) = 38;$$
$$\ldots$$

2.2 The Query Language

In relational databases, data are retrieved using SQL (Structured Query Language). A query statement of SQL maps one or several input tables into a result table.

Example 1. (Cont'd) The following is a simple query statement:

SELECT *eid*
FROM *Employee*
WHERE *position* = 'manager' and
gender = 'female';

The result table has only one tuple whose *eid* is 105.

More complicated queries may involve multiple tables. Other conditions (or even subqueries) can be added to the WHERE clause.

2.3 Embedded SQL

SQL queries can be embedded into programs written in conventional high-level programming languages (called *host* languages) such as C/C++ or Java. Some features of SQL allow us to manipulate the data in the relations using variables of the host language.

As an example, the following is a code fragment embedding SQL statements:

```
dbfcmd(dbproc, "select eid");
dbfcmd(dbproc, "from employee");
dbfcmd(dbproc, "where position =
        'manager'");
dbsqlexec(dbproc);
while (dbnextrow(dbproc) !=
          NO_MORE_ROWS) {
    /* ... */
}
```

3 Database Instance Generation

3.1 The Problem

To test a program with embedded SQL statements, we need to find appropriate input data, which include a database instance. The conventional white-box testing methods are inadequate because the execution of the program is affected by the result of SQL statements. Let us consider the code fragment in Section 2.3. The loop is not executed if the query results in an empty table. For the purpose of white-box testing, we need input data such that the loop is executed once or not executed at all.

Thus, test data generation for database applications necessitates the tackling of the following general problem:

> Given an SQL statement and a property, find a database instance such that the result of executing the statement satisfies the property.

We understand that test data generation also comprises finding the values of ordinary program variables. But in this paper, we would like to focus on the database-specific aspects and the generation of a database instance.

We refrain from giving a language to describe the properties. The following are some examples of them:

(P1) The result table is empty, i.e., it does not have any row.

(P2) The result table has a row which has a negative attribute value.

(P1) and (P2) represent the scenario of null case and exception, respectively. These are two common properties that practitioners used to test database application programs. Let

us denote the result table by Rst. Without loss of generality, we assume that it has a single column. The above properties can be represented by the following two formulas in the syntax of SQL:

(p1) NOT EXISTS Rst

(p2) 0 > ANY Rst

3.2 The Constraint Satisfaction Approach

A *constraint satisfaction problem* (CSP) consists of a set of unknowns, each of which can bind to a value in some domain [5, 8]. The values of these unknowns are governed by a set of constraints. In the study, we put no restriction on the forms of the constraints. A constraint can be a logical formula or a linear equation. Unknowns are modeled using variables. To solve a CSP, we need to find the values of the variables such that all the constraints are satisfied.

For the purpose of test data generation, we use a formula to describe the relationship between the input (database instance) and the output (result table). Solutions satisfying the formula can be computed using existing techniques and tools available in the CSP community.

For simplicity, we assume that there are two tables (t_1 and t_2) and the attributes are denoted by $a_0, a_1, a_2, \ldots$ The selected attribute is a_s. Suppose table t_1 contains the rows t_{1i} ($1 \leq i \leq m$) and table t_2 contains the rows t_{2j} ($1 \leq j \leq n$). Let $Cond$ denote the condition in the WHERE clause. We first consider the simpler case where $Cond$ does not involve subqueries. Our goal is to determine the values of the following entries such that some formula holds.

- $t_{1i}.a_k$, for each row i and each attribute a_k of t_1
- $t_{2j}.a_l$, for each row j and each attribute a_l of t_2

The formula is generated based on an SQL statement and a property. For example,

- if the property is "EXISTS Rst", the formula is $\bigvee_{i,j} C'(i,j)$;
- if the property is "$const\ rop$ ALL Rst" ($const$ is a numeric constant, rop is a relational operator such as $\geq$), the formula is $\bigwedge_{i,j} (C'(i,j) \rightarrow (const\ rop\ a'_s))$.

Here $C'(i,j)$ is derived from $Cond$ by changing $t_1.a_k$ to $t_{1i}.a_k$ and changing $t_2.a_l$ to $t_{2j}.a_l$. Note that $t_1.a_k$ and $t_2.a_l$ are used in the SQL statement while $t_{1i}.a_k$ and $t_{2j}.a_l$ denote unknowns of the constraint satisfaction problem. If the selected attribute a_s is from table t_1, a'_s is $t_{1i}.a_s$; and if a_s is from table t_2, a'_s is $t_{2j}.a_s$. If both t_1 and t_2 have the attribute a_s, the user should specify explicitly which one is to be selected.

Example 2. Suppose there are two tables (t_1 and t_2) and 5 attributes ($a_0, \ldots, a_4$). Table t_1 has the attributes a_0, a_1, a_3; while Table t_2 has the attributes a_2, a_3, a_4. A possible query could be:

SELECT a_1, a_2, a_4
FROM t_1, t_2
WHERE $t_1.a_3 = t_2.a_3$

If there is i ($1 \leq i \leq m$) and j ($1 \leq j \leq n$) such that ($t_{1i}.a_3 = t_{2j}.a_3$), the result table contains the row $\langle t_{1i}.a_1, t_{2j}.a_2, t_{2j}.a_4 \rangle$.

For the property (P1), the formula is

$$(t_{11}.a3 \neq t_{21}.a3) \wedge (t_{11}.a3 \neq t_{22}.a3) \wedge \ldots \wedge (t_{11}.a3 \neq t_{2n}.a3)$$
$$\ldots$$
$$(t_{1m}.a3 \neq t_{21}.a3) \wedge (t_{1m}.a3 \neq t_{22}.a3) \wedge \ldots \wedge (t_{1m}.a3 \neq t_{2n}.a3)$$

Now let us consider the property (P2). Suppose we desire that the result table has a row, in which there is a negative entry (e.g., the value of the attribute a_1 is negative). Then the formula is:

$$\bigvee_{i,j} ((t_{1i}.a3 = t_{2j}.a3) \wedge (t_{1i}.a1 < 0))$$

When there are subqueries, $Cond$ is not a pure formula, and we cannot derive C' from it by simply substituting variable names. In this case, we use a recursive procedure, and work from the innermost subqueries to the outermost one.

Constraint Solving

The constraint satisfaction problem (CSP) represents a general class of problems. Given a set of constraints, we need to find appropriate values for the variables such that all the constraints hold, or determine the existence of contradiction. In the later case, there is no database instance of the given size, which has the desired properties.

Depending on the form of constraints, we can solve them using different automatic tools. Here we use the general-purpose tool BoNuS [10]. It can solve constraints involving variables of different types (e.g., enumeration, integers, reals). Rather than describing the tool in detail, we give a simple example here. Suppose we would like to find the integers x and y such that the following conditional expression holds: (($x \geq 10$) or ($x \leq 4$)) and ($x \geq 2y + 1$) and ($y \geq 6$). We may give the following as input to BoNuS:

```
int x, y;
bool B1 = (x >= 10);
bool B2 = (x <= 4);
bool B3 = (x >= 2*y + 1);
bool B4 = (y >= 6);
```

```
{
  and(or(B1,B2), B3,B4);
}
```

A solution will be found, e.g., $x = 13$, $y = 6$.

3.3 Other Constraints

In addition to the constraints generated from the SQL statement, there are usually other kinds of constraints to be satisfied. Examples include uniqueness constraints (given by primary keys) and business rules.

A *primary key* is a column or (set of columns) that uniquely identifies the rest of the data in any given row. For example, a student's ID number (SID) may be a primary key for a Student table. In other words, each row in the Student table is uniquely identified by the SID column. We can represent this as the following constraints:

$$stud_i.SID \neq stud_j.SID$$

(for $1 \leq i < j \leq n$). Here $stud_i$ and $stud_j$ represent the i'th and j'th row, respectively, and n is the number of rows.

If the primary key consists of two or more attributes, each constraint is a disjunctive formula. In *Example 2*, suppose a_2 and a_3 form the primary key of the table t_2. Then we have the following constraints:

$$(t_{2i}.a_2 \neq t_{2j}.a_2) \vee (t_{2i}.a_3 \neq t_{2j}.a_3)$$

(for $1 \leq i < j \leq n$).

Besides primary keys, some attributes can be *foreign keys*. The purpose of these attributes are to establish links between different tables. The values of a foreign key should appear as values of some attribute in another table. For example, a customer may place several orders. Suppose the CID is the primary key of the table CUSTOMER, and it is also a foreign key of the table ORDER. The later can be expressed by the following constraints:

o_i.CID = c_1.CID $\vee$ o_i.CID = c_2.CID $\vee$
$\ldots \vee$ o_i.CID = c_m.CID
(for every row i of ORDER)

Here o_i (c_j) denotes a row in the table ORDER (CUSTOMER, resp.), and m is the number of rows in the table CUSTOMER.

3.4 An Automatic Tool

We have implemented an automatic tool for generating database instances, in the C language. The input of the tool consists of three parts: *schema*, *SQL statement* and *assertion*. The output is a set of constraints (which can be given to BoNuS).

First let us see a very simple example.

Example 3. The input is like this:

```
CREATE table t1
  ( a1 int, a2 float, a3 int) [1] ;
CREATE table t2
  ( b1 int, b2 int) [2] ;

SELECT t1.a1, t2.b2
FROM t1, t2
WHERE a1=b1;

EXISTS RESULT;
```

There are two tables, having 3 and 2 attributes, respectively. Note that we have specified that the first table has one row and the second table has two rows. These are just our assumptions. They can be changed, of course. For instance, we may look for two tables, each having three rows. The keyword `RESULT` denotes the result of the query. In this example, we are looking for two tables, such that the result table is not empty. With our tool, we get the following set of constraints:

```
int t1_0_a1;
real t1_0_a2;
int t1_0_a3;
int t2_0_b1, t2_1_b1;
int t2_0_b2, t2_1_b2;
bool B0_0_0 = ( t1_0_a1==t2_0_b1 );
bool B0_0_1 = ( t1_0_a1==t2_1_b1 );
{
  or(B0_0_0, B0_0_1);
}
```

Here `t1_0_a1` denotes the value of the attribute `a1` in the first table's only row, `t2_1_b1` denotes the value of the attribute `b1` in the second table's second row, and so on. The above set of constraints has solutions, which can be found by BoNuS. Thus we are able to generate database instances such that the result of a query has certain properties (as specified by the assertion).

The next example involves a slightly more complicated SQL statement.

Example 4. Suppose there is a table called `book`, which has two attributes `isbn` and `year`. We would like to generate a database instance in which there is no book which is published in two different years.

The input is as follows:

```
CREATE TABLE Book
  ( isbn int, year int) [2];

SELECT isbn
FROM Book Old
WHERE year < ANY
{
```

```
  SELECT year
  FROM Book
  WHERE isbn = Old.isbn
);

NOT EXISTS RESULT;
```

Our tool will generate the following constraints:

```
int Book_0_isbn, Book_1_isbn;
int Book_0_year, Book_1_year;
bool B0= (Book_0_year< Book_0_year);
bool B1= (Book_0_isbn==Book_0_isbn);
bool B2= (Book_0_year< Book_1_year);
bool B3= (Book_1_isbn==Book_0_isbn);
bool B4= (Book_1_year< Book_0_year);
bool B5= (Book_0_isbn==Book_1_isbn);
bool B6= (Book_1_year< Book_1_year);
bool B7= (Book_1_isbn==Book_1_isbn);
{
   not(or(or(and(B1,B0),and(B3,B2)),
          or(and(B5,B4),and(B7,B6))
      ));
}
```

BoNuS can tell us that they are satisfiable. On the other hand, if we change the "less-than" operator ('<') to the equal sign, there will be no solution. The result is always non-empty, provided that the input table is not.

4 Concluding Remarks

The testing of database application programs has not received much attention previously. In particular, few test data generation techniques consider the inclusion of database instances. In this paper, we propose a method for automatic generation of database instances, which can be used for white-box testing. A prototype tool is also described. Its input consists of an SQL statement, the database schema definition, together with an assertion which represents the requirement of the tester. The output is a set of constraints which can be given to existing constraint solvers. If they are satisfiable, we obtain the desired database instances.

Certainly, it is impossible to work out a fully automatic tool for test data generation, even when the program does not involve databases. But it is reasonable to expect that a powerful tool will assist the tester significantly. Currently, our tool still has some limitations. For instance, it does not handle string variables. We need to represent them as integers, instead. In the future, we will improve the constraint generation tool described in this paper, as well as the constraint solver.

Acknowledgments

The work described in this paper has been supported by the Research Grants Council of Hong Kong, China (Project no. DAG99/00.EG05), National Natural Science Foundation of China (grant No. 69703014) and the Chinese "973" project (No. G1998030600).

References

[1] B. Beizer, *Software Testing Techniques (Second Edition),* Van Nostrand Reinhold International Company Limited, New York, 1990. Chapter 1, pages 8, 10–11, 20–22, 546, 550.

[2] M.Y. Chan and S.C. Cheung, Testing database applications with SQL semantics, *Proc. of the 2nd Int'l Symp. on Cooperative Database Systems for Advanced Applications (CODAS'99)*, 364–375, March 1999.

[3] D. Chays, S. Dan, P.G. Frankl, F.I. Vokolos and E.J. Weyuker, A framework for testing database applications, *Proc. Int'l Symp. on Software Testing and Analysis (ISSTA'00)*, 147–157, August 2000.

[4] E.F. Codd, *The Relational Model for Database Management: Version 2,* Addison-Wesley, Reading, Mass., USA, 1990.

[5] P. G. Jeavons, D. A. Cohen, M. Gyssens, Closure Properties of Constraints, *Journal of the ACM,* 44(4): 527–548, July 1997.

[6] B. K. Patel, Automated tools for database design and criteria for their selection for aerospace applications, *IEEE Aerospace Applications Conference Digest,* 23–25, Feb 1989.

[7] M. Roper, *Software Testing,* New York, The McGraw-Hill, Inc., 1994. Chapters 1–3, pages 32–96.

[8] E. Tsang, *Foundations of Constraint Satisfaction,* Academic Press, London, 1993. Chapters 1, 2, 6, 10, pages 1–52, 157–188, 299.

[9] J.D. Ullman and J. Widom, *A First Course in Database Systems,* Prentice Hall, 1997.

[10] J. Zhang, Specification analysis and test data generation by solving Boolean combinations of numeric constraints, *Proc. of the First Asia-Pacific Conf. on Quality Software (APAQS)*, (eds.) T.H.Tse and T.Y.Chen, 267–274, 2000.