

Fault-Based Testing of Database Application Programs with Conceptual Data Model^{*†}

W. K. Chan[‡], S. C. Cheung
Hong Kong University of Science and Technology
{wkchan, sccheung}@cs.ust.hk

T. H. Tse[§]
The University of Hong Kong
thtse@cs.hku.hk

Abstract

Database application programs typically contain program units that use SQL statements to manipulate records in database instances. Testing the correctness of data manipulation by these programs is challenging. When a tester provides a database instance to test such a program, the program unit may output faulty SQL statements and, hence, manipulate inappropriate database records. Nonetheless, these failures may only be revealed in very specific database instances.

This paper proposes to integrate SQL statements and the conceptual data models of an application for fault-based testing. It proposes a set of mutation operators based on the standard types of constraint used in the enhanced entity-relationship model. These operators are semantic in nature. This semantic information guides the construction of affected attributes and join conditions of mutants. The usefulness of our proposal is illustrated by an example in which a missing-record fault is revealed.

Keywords: database application testing, fault-based testing, semantic mutants.

1. Introduction

Testing is the most popular means to assure software quality. Among the factors that may affect test results, the environment such as the operating system and the databases is an important issue. In particular, little research on unit testing techniques focuses on the interaction between an application and the databases [15]. We propose a fault-based testing technique in this paper.

Typical database application programs (*DB applications*) use structured query languages (SQL) [1], such as Oracle 10g, to manipulate data governed by database systems. These data are described structurally in database systems as *database schemas*, which are sets of rules to define the organization and integral relationships of data. A set of data records satisfying a particular database schema is known as a *database instance*. Similarly to [15], we define a DB application to be a *program unit* with a set of database schema. To process a database instance of a database schema, a program unit accepts *parametric input values* and outputs relevant SQL statements, each of which is a specification of the intended manipulation procedure. Actual records in the database instance are then manipulated according to the SQL statements.

To test a DB application, therefore, testers need to handle this indirect manipulation of data. Testers input a combination of database instances and parametric input values, if any, applicable to run the program unit. Such a combination defines a *test case* of the DB application.

The program unit may output SQL manipulation statements, such as SELECT, INSERT, UPDATE, and DELETE. These statements will in turn select or modify records in the target database instances. Testers can use either the direct outputs (SQL statements) to observe the test results of the

* ©2005 IEEE. This material is presented to ensure timely dissemination of scholarly and technical work. Personal use of this material is permitted. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder. Permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

† This research is supported in part by a grant of the Research Grants Council of Hong Kong (Project No. HKUST 6187/02E) and a grant of The University of Hong Kong.

‡ Part of the research was done when Chan was with The University of Hong Kong.

§ All correspondence should be addressed to T. H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Tel: (+852) 2859 2183. Fax: (+852) 2559 8447. Email: thtse@cs.hku.hk.

program unit, or the derived outputs (database instances) to observe the test results of the DB application.

The chance of revealing a failure due to a faulty SQL statement, however, differs between these two types of observation. Since the same SQL statement can generally be applied to different database instances, depending on the particular combinations of values of a database instance, a faulty SQL data manipulation statement may be semantically intact with some database instances. Such a faulty SQL statement may select or modify the same set of records in a database instance as if it were the expected one. Detecting a failure in one database instance does not guarantee the revealing of similar failures in other database instances. Intuitively, when the failure is due to a faulty SQL statement, techniques to reveal a failure at the SQL statement level should be more cost effective than those at the database instance level.

In view of the above, among other problems in testing DB applications [20], there are two major challenges for applying fault-based testing to such programs: (a) formulating strategies to discriminate a SQL statement from its mutants and (b) discriminating SQL statements by means of database instances.

Intuitively, conceptual data models [11] generally contain more semantic information, such as constraints, than database schemas relevant to a DB application. The latter seldom implements all the designed constraints [9]. We propose to use the information captured in conceptual data models to facilitate testers to reason about whether a given SQL statement issued by the program unit of a DB application manipulates the correct sets of data. At the level of parametric inputs and direct outputs of the program unit, there is no need to introduce any particular database instance. Testers may further construct database instances to initialize the selected differentiation scheme. In this way, the evaluation of the DB application can also be conducted at the database instance level.

The main contributions of this paper include: (i) defining semantic mutation operators based on an enhanced entity-relationship model, a type of conceptual data model, (ii) addressing challenge (a) by exploiting the semantic connections between the associated enhanced entity-relationship model and embedded SQL statements to produce SQL-oriented mutants, and (iii) alleviating challenge (b) by cross-comparing the database records manipulated by SQL statements and their mutants.

The rest of the paper is organized as follows: Section 2 summarizes and compares other research work related to the current project. Section 3 will introduce a sample application that we shall use to illustrate our proposed technique. Based on the constraints of the enhanced entity-relationship model defined in Section 4, semantic mutation operators will be defined in Section 5. Section 6 develops

on Section 3, to set the scene for applying our fault-based testing technique in Section 7. Finally, concluding discussions will be given in Section 8.

2. Related work

2.1. Testing of database application programs

Test data selection strategies: Chan and Cheung [4] translated embedded SQL statements into ordinary statements to generate test cases conventionally. Kapfhammer and Soffa [15], on the other hand, proposed data-flow testing criteria to test DB applications directly. Haraty et al. [12, 13] regressed DB applications and, hence, reuse test cases by analyzing program dependencies. Robbert and Maryanski [19], tested against individual attributes, relations, and constraints in a given data model.

Database instance generation: (Semi-)automatic construction of database instances lowers the cost of testing. Chays et al. [6, 9] constructed database instances from sample files that are produced according to category-partitioning schemes on the database schema. Their tool was also used in [10] to test web-based applications by extracting URL control graphs from applications and selecting control sequences to generate test cases. Zhang et al. [23] translated SQL statements with database schemas into constraints and applied constraint solvers to produce fault-based database instances. The generation of constraint-based database test instances was also considered by Neufeld et al. [17].

Other related work: Suárez-Cabal and Tuya [20] measured the extensiveness of coverage of the conditions in target SQL `SELECT` statements used to identify records in database instances. This inspires us to use located or missing records to distinguish our mutants. Chatterjee et al. [5] used data versioning to allow independent tests to be executed concurrently. Offutt and Xu [18] tested XML-based messages for web services according to mutation operators and partitioning rules for XML schemas. Project SQLUnit [2] uses a JUnit-like framework to test stored procedures in databases. They render research on the testing of DB applications increasingly practical (see Section 2).

2.2. Fault-based and mutation testing

Fault-based testing aims at demonstrating the absence of prescribed faults in a program [16]. According to Zhu et al. [24], there are at least three kinds of relevant adequacy criterion, namely, mutation testing, perturbation testing, and error seeding.

Mutation testing [8, 14] is an effective, though computationally expensive, fault-based software testing

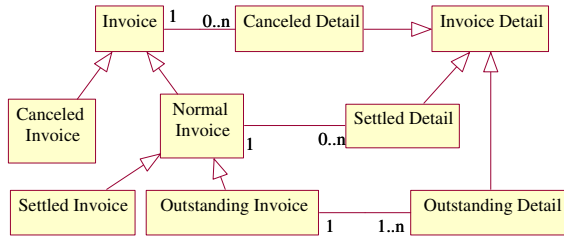


Figure 1. Conceptual data model.

technique. Its core is a set of mutation operators. Each mutation operator involves a legal *syntactic* change of a program statement. Hence, the applications of mutation operators to a program result in a set of mutants. For strong mutation testing [8], a mutant is said to be killed by a test case if the output of the mutant differs from that of the original program. For weak mutation testing [14, 21], a mutant is killed if the program state after a mutated statement (or component) differs from that of the original program. In either case, such a test case can be used to determine whether it reveals a failure of the original program.

Morell [16] generalized mutation testing to allow the classes of alternative expressions to be infinite. He illustrated how to generate a symbolic expression that described the input of an original program and its alternative forms. The solution of a symbolic expression determines whether a set of alternative forms can be distinguished by a test case.

Perturbation testing [22] considers faults as attributes in an error space. A perturbation of a function captures the difference between the correct function and an erroneous counterpart. This allows testers to treat faults as functional properties instead of syntactic changes of the original program.

Error seeding [3], injects artificial faults into a program. Chen et al. [7], for example, injected faults dynamically into system functions to test the exception-handling behaviors of programs.

2.3. Comparisons

This paper proposes to generate SQL-oriented mutants as a test data selection strategy. Like [19], we make use of conceptual data models and require test sets to cover the properties so defined. Unlike [19], however, we produce mutants of SQL statements to be executed by test sets. Moreover, our mutants conform to the data model, whereas previous studies [17, 23] produced fault-based database instances that aimed at violating given database constraints.

Our approach is similar to the work of Morell [16] and Zeil [22] for conventional programs. Based on the semantics of SQL statements and a given EER model (see Section 4), we construct alternative constraints to depict alternative mutants. We use the specification nature of the SQL language to treat a SQL statement as a relation. Our approach also allows testers to compare the mutants with the original SQL statements at the symbolic level.

3. Invoice payment analyzer: a sample application

Consider a conference hosting company that develops a DB application to obtain the payment status of their invoices. Invoices are partitioned as normal or canceled. Normal invoices are further partitioned as settled or outstanding, depending on whether payments from customers are required to follow. Each invoice can have invoice detail(s). Like invoices, invoice details are partitioned as canceled, settled, or outstanding. When all the details of an invoice are canceled or settled, the invoice will be treated as settled. When full payment is received in advance, a settled invoice will be directly created *without* invoice details. Furthermore, every outstanding invoice should have at least one outstanding invoice detail. Any canceled invoice should not have any settled or outstanding invoice detail, but may have canceled invoice detail, if any. All the inheritance relations described above refer to total and disjoint specialization relations.

Figure 1 shows a conceptual data model (in UML notation) of invoices and their details of the sample application. The application will also be used in Sections 5.1 and 6 to illustrate our testing technique.

4. The enhanced entity-relationship model

We use the enhanced entity-relationship model (EER Model) [11] as the conceptual data model. An *entity* is something with properties. They are classified into *entity types*. A *relationship* is an association between two or more entities. A relationship set is a set of relationships of the same type. It can be expressed in the form of table known as a *relation*. Each entity type has *attributes* and each attribute has a value domain. Every value combination of an *entity key* uniquely identifies an entity of a specific type.

We further describe the standard types of constraint used in the EER model. The value of any *derived attribute* is worked out from other attributes. We assume that the given EER models capture the necessary derivation formulas. A *weak entity type* has no entity key of its own. It is identified by an *identifying entity type*. A

participation constraint defines whether an entity must be involved in some relationship. A *cardinality constraint* defines the number of entities of a specified type involved in a relationship.

The *generalization / specialization completeness constraint* can be total or partial. Informally, when any member of a superclass must also be a member of at least one of its subclasses, then the constraint is said to be total. Otherwise, it is known as partial. Similarly, the *generalization/specialization disjointness constraint* can be disjoint or overlapping. The former refers to the restriction that any entity cannot also be a member of a sibling class. Otherwise, it is known as overlapping. The *union type completeness constraint* can be total or partial. The former means that any member in a subclass should also be entities in all its direct superclasses. Otherwise, it is known as partial.

5. Mutations according to SQL relations on EER model

SQL is a fundamental element of database application programs. In this section, we illustrate via an example that a database schema may not capture all the constraints defined in an EER model, and then put forward a set of mutation operators based on SQL relations on the EER model.

5.1. Inadequacy of database schema constraints

Let us consider the entity types in Figure 1, in which all the specialization relations are total and disjoint. Consider a particular database schema implementation where each entity type is implemented as a separate table.

Since a canceled detail record can be related to a settled, canceled, or outstanding invoice, and since the latter invoices are in separate tables, there is no foreign key to associate canceled detail records to invoice records. We can see that the association between *Invoice* and *Canceled Detail* in Figure 1 is *not* captured in the above database schema.

5.2. Proposed mutation operators

We propose a set of mutation operators based on the standard types of constraint in the EER model. They cover mutations due to replacements, insertions, and deletions. Table 1 shows a list of replacement mutation operators. They are described and explained briefly via an example in this section. We shall illustrate in more detail in Section 7 how mutants due to two of the operators, namely, PTCR and GS DR, can be killed using the proposed technique.

```

s1 public int getPaymentReceived(int cutoff)
s2     throws SQLException
s3     {
s4         int amount = 0;
s5         String sql = "SELECT sum(D.dtl_pay_amt) " +
s6             "FROM Invoice H, InvDtl D " +
s7             "WHERE H.inv_no = D.inv_no " +
s8             "AND H.inv_status in ('S', 'O') " +
s9             "AND D.dtl_status in ('S', 'O') " +
s10            "AND D.dtl_unit_pay_amt >= " + cutoff + ";";
s11         Statement stmt = new conn.createStatement();
s12         ResultSet rs = stmt.executeQuery(sql);
s13         if (rs.next())
s14             amount = rs.getInt(1) + amount;
s15         return amount;
s16     }

```

Figure 2. A faulty program unit `getPaymentReceived()` of the invoice payment analyzer.

Insertion and deletion mutation operators can be treated similarly.

Consider an embedded SQL statement in Figure 2. It contains a SQL query with four join conditions over two tables, namely *Invoice* and *InvDtl*. These two tables implement the type hierarchies in Figure 1, where *Invoice* and *Invoice Detail* are the respective base types.

PTCR: The participation constraint replacement operator mutates the participate requirement of an entity type from “must-participate” to “non-participate”, or vice versa. Readers may refer to Section 7 for more details.

CDCR: The cardinality constraint replacement operator alters the cardinalities of entity types in the relation. It also forces the mutated entity type to take specific (or extreme) values within the cardinality constraints. For example, a mutant may force the relation to select entities of *Invoice* that are associated with more than 10 entities of *Invoice Detail*. This can be expressed by adding a subquery “10 < (SELECT count(*) FROM InvDtl D2 WHERE H.inv_no = D2.inv_no)”.

IWKR: The identifying / weak entity type replacement operator determines whether an (expression on) identifying or weak entity type(s) is involved in the relation. For an identifying entity type, IWKR will substitute it with a weak entity type. For a weak entity type, IWKR will substitute it with an identifying entity type. Suppose *Invoice Detail* is a weak entity type and its identifying entity type is *Invoice*. A mutant can be formed by replacing “InvDtl” in line s_6 by “Invoice”, “sum(D.dtl_pay_amt)” in line s_5 by “sum(D.pay_amt)”, “sum(dtl_status)” in line s_9 by “sum(inv_status)”, and “sum(D.dtl_unit_pay_amt)” in line s_{10} according to Formula (1) in Section 6.2.

ATTR: The attribute replacement operator substitutes (an expression on) attribute(s) by (an expression on) other attribute(s) of a compatible type. For

Table 1. Replacement mutation operators.

Semantic Mutation Operator	Acronym	Description
Participation Constraint Replacement	<i>PTCR</i>	Toggle the participation requirements of entity types in the relation.
Cardinality Constraint Replacement	<i>CDCR</i>	Replace the cardinalities of entity types in the relation.
Identifying / Weak Entity Type Replacement	<i>IWKR</i>	Replace (an expression of) identifying type(s) by (an expression of) weak entity type(s), or vice versa.
Attribute Replacement	<i>ATTR</i>	Replace (an expression on) attribute(s) by (an expression on) other attribute(s) of a compatible type.
Generalization / Specialization Completeness Replacement	<i>GSCR</i>	Replace an expression on a partial superclass by an expression on a subclass <i>and</i> the negation form of the superclass
Generalization / Specialization Disjointness Replacement	<i>GSDR</i>	Replace (an expression on) sibling entity type(s) by (an expression on) other sibling entity type(s) under the same superclass.
Union Type Completeness Replacement	<i>UTCR</i>	Replace an entity type by a subclass and/or superclasses of the subclass, such that these superclasses have the same union type constraint

example, “sum(D.dtl_pay_amt)” may be replaced by “sum(D.dtl_amt)”. This operator also includes the replacement of a composite or structured attribute by its part, and vice versa.

GSCR: The generalization / specification completeness replacement operator substitutes an expression on a partial superclass by an expression on a subclass *and* the negation form of the superclass. Suppose we have a subclass of normal invoices with a status of “N”, which are not settled or outstanding. The statement “H.inv_status in (‘S’, ‘O’)” on line *s*₈ may be mutated to “H.inv_status = ‘N’ and H.inv_status not in (‘S’, ‘O’)”.

GSDR: The generalization / specialization disjointness replacement operator substitutes (an expression on) sibling entity type(s) by (an expression on) other sibling entity type(s) under the same superclass. A sample mutant is to replace the set {*Settled Invoice*, *Outstanding Invoice*} by the set {*Settled Invoice*, *Canceled Invoice*}. Readers may refer to Section 7 for more details.

UTCR: The union type completeness replacement operator replaces an entity type by (an expression on) a subclass and/or superclass(es) of the subclass, such that these superclasses have the same union type constraint with the original entity type. Suppose, for example, an entity type *A* has three superclasses *B*, *C*, and *D*, and the inheritance relation between the former and the latter superclasses is a complete union type relation. The entity type *B* may be replaced by *A*, *C*, *D*, or other combinations of *B*, *C*, and *D*.

6. An implementation of invoice payment analyzer

The sample DB application contains a database schema (Figure 3), an EER model (Figure 1), and a program unit (Figure 2).

```

CREATE TABLE Invoice (
  inv_no      char(10)
             primary key,
  inv_dt0     datetime,
  inv_stl_dt  datetime,
  inv_status  char(1),
  inv_amt     int,
  inv_qty     int,
  pay_amt     int,
  cust_no     char(10),
  order_no    char(15) )

CREATE TABLE InvDtl (
  inv_no      char(10)
             foreign key references
             Invoice (inv_no),
  dtl_no      char(4),
  dtl_status  char(1),
  dtl_amt     int,
  dtl_qty     int,
  dtl_pay_amt int,
  dtl_unit_pay_amt int,
  order_no    char(15),
  order_dtl_no char(4),
  conf_item   char(26),
             constraint invdtl_pk
             primary key (inv_no, dtl_no) )

```

Figure 3. Database schema definition.

6.1. Database schema

Figure 3 shows the database schema for invoices and invoice details in Microsoft SQLServer 2000 syntax. The table *Invoice* implements all the entity types concerning invoices, including *Invoice*, *Normal Invoice*, *Settled Invoice*, *Outstanding Invoice*, and *Canceled Invoice*. The derived attribute of invoice amount, shown in the column *inv_amt*, is equivalent to (or derived from) the sum of *dtl_inv_amt* of the corresponding outstanding and settled invoice details. The interpretations of the columns *inv_qty* and *pay_amt* with respect to their invoice details are similar. In particular, *dtl_unit_pay_amt* is derived through dividing *dtl_pay_amt* by *dtl_qty* of the same row. The other columns are self-explanatory.

The table *InvDtl* implements all the entity types concerning invoice details, including *Invoice Detail*, *Canceled Detail*, *Settled Detail*, and *Outstanding Detail*. All the columns are self-explanatory.

There are two types of constraint in the database schema in Figure 3. The constraint “*inv_no* char(10) primary key” specifies that the attribute/column *inv_no* is an

entity key of the table `Invoice`. Similarly, the entity key of the table `InvDtl` is a combination of `inv_no` and `dtl_no`. The constraint “`inv_no char(10) foreign key references Invoice (inv_no)`” specifies that any entity in table `InvDtl` should have a corresponding entity in table `Invoice` in such a way that the attributes `inv_no` on both tables must agree with each other.

The symbols “x” for canceled, “s” for settled, and “o” for outstanding are used for the columns `inv_status` and `dtl_status`. Thus, an invoice may be described as canceled, settled, or outstanding using the predicates `inv_status = 'X'`, `inv_status = 'S'`, or `inv_status = 'O'`, respectively. The predicates for invoice details are defined similarly: `dtl_status = 'X'` for a canceled detail, `dtl_status = 'S'` for a settled detail, and `dtl_status = 'O'` for an outstanding detail.

6.2. Other EER model constraints and the sample program unit

The expected function of the program unit `getPaymentReceived()` calculates the total amount of payments received for those invoices that have paid at least some cutoff amount per quantity amount.¹ The cutoff amount is specified as an input parameter.

Suppose that, according to user requirements, the processing logic should also take into account the effect of directly settled invoice *without* invoice details. It should use the `Invoice` table to approximate the average detail amount per quantity amount as if the settled invoice details were present, thus:

$$dtl_unit_pay_amt = \frac{Invoice.pay_amt}{Invoice.Inv_qty} \quad (1)$$

Since this is an important requirement, we also expect this information to be captured as constraints in the EER model.

A sample implementation of the function is shown in Figure 2. Statements s_5 to s_{11} in the implementation show a SQL query statement. It simply joins the invoice table and the invoice detail table via the invoice number attribute (statement s_7). It wants to select those records in either outstanding or settled status with the detail payment amount per quantity amount being not less than the cutoff amount. This sample program is faulty, however, as if the data model perceived by the developer were an oversimplified master-detail model, as shown in Figure 4,

¹ Apart from using JDBC primitives, there are other methods, such as SQLJ, in which an entire SQL statement with binding variables are defined in an embedded SQLJ-directive block. The identification of SQL statements will be easier than what has been illustrated in [15]. In the latter case, software testers may use database application control flow graphs [15] to find out the possible sets of statements within a program for each database interaction point, such as `executeQuery()`, then mutate the statements, and finally make changes to the program accordingly.

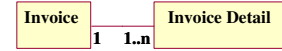


Figure 4. An oversimplified data model of invoices and invoice details.

```
SELECT sum(D.dtl_pay_amt)
FROM Invoice H, InvDtl D
WHERE H.inv_no = D.inv_no
AND H.inv_status in ('S', 'O')
AND D.dtl_status in ('S', 'O')
AND D.dtl_unit_pay_amt >= <cutoff value>
UNION ALL
SELECT sum(H.pay_amt)
FROM Invoice H
WHERE H.inv_status = 'S'
AND (H.pay_amt >= H.inv_qty * <cutoff value>)
AND NOT EXISTS (SELECT 1 FROM InvDtl D
                 WHERE D.inv_no = H.inv_no
                 AND D.dtl_status = 'S')
```

Figure 5. An expected SQL query for `getPaymentReceived()`.

such that every invoice must have complete records of invoice details.

Figure 5 shows the SQL query statement expected for a correct implementation. Compared to the faulty version in Figure 2, it also selects those invoices *without* any settled invoice details. In the next section, we shall present the fault-based approach to detect this error.

7. Applying the fault-based approach

This section illustrates how to derive fault-based mutants from the embedded SQL statements using the mutation operators defined in Section 5. It first analyzes SQL statements using the predicates of corresponding entity types. Consider the faulty SQL query in Figure 2 again. The SQL statements s_5 – s_{10} access the tables `Invoice` and `InvDtl`. Table `Invoice` is mapped to all kinds of invoice entity type. Similarly, table `InvDtl` is mapped to all kinds of invoice detail entity type. According to the EER model in Figure 1, by including the specialization relations, there are in total two kinds of association related to the SQL statement. They are the associations between *Normal Invoice* and *Settled Detail*, and those between *Outstanding Invoice* and *Outstanding Detail*.

There are known limitations in mutation testing, which also applies to our approach. For example, the question of

determining whether a program mutant cannot be killed by any possible test case is generally undecidable. However, since SQL is a formal language and since every SQL statement is self-contained, we believe that, unlike the case of conventional mutants where we must consider all relevant program states before concluding whether a (weak) mutant can be killed, some SQL mutants can be analyzed and compared with the original SQL statement to determine whether it is an equivalent mutant. When it is indeed an equivalent mutant, there is no need to use it to synthesize a program fault for the DB application. This also reduces the number of program mutants and, hence, the cost of testing a subject program.

7.1. Applying the GSDR mutation operator

Let us consider the total and disjoint specialization relation of the entity type *Invoice Detail*. *Settled Detail* and *Outstanding Detail* are defined in the relation. Testers can apply the GSDR mutation operator. Since there are three subclasses of *Invoice Detail*, by exhaustion, there are seven possible non-empty subsets of $\{Canceled\ Detail, Settled\ Detail, Outstanding\ Detail\}$. One of them is defined in the relation. Hence, there are six possible mutations for this operator based on the specialization relation. Consider a substitution that replaces the original subset by $\{Settled\ Detail, Canceled\ Detail\}$. It will result in the following condition fragment:

$$\begin{aligned}
 & \text{InvDtl.inv_no} = \text{Invoice.inv_no} \\
 & \text{AND Invoice.inv_status IN ('S', 'O')} \\
 & \text{AND InvDtl.dtl_status IN ('S', 'X')} \\
 & \text{AND InvDtl.dtl_unit_pay_amt} \geq \langle \text{cutoff value} \rangle
 \end{aligned} \tag{2}$$

We note the following: When testers apply conventional, syntactic mutation operators, such as “replace a constant by another constant”, they can also obtain a mutant similar to that in equation (2). For example, by replacing the constant symbol “O” by the symbol “X” in statement s_9 of the faulty program, testers can generate the same mutant. There is, however, a difference between such syntactic mutants and our semantic mutants. Our approach replaces the sets of logical subclasses of the superclass in a specialization hierarchy. For example, we will generate a mutant that replaces “(‘S’, ‘O’)” of statement s_9 by “(‘X’)”. An application of the conventional mutation operator cannot produce such a change.

To unify the identifiers of equation 2 with those of Figure 2, testers substitute the table identifiers *Invoice* and *InvDtl* in equation 2 by the instance identifiers “H” and “D”, respectively. They then substitute the unified equation 2 into the SQL statements s_7-s_9 of Figure 2 to produce a

mutant M_1 . The embedded SQL part of M_1 will be:

```

SELECT sum(D.dtl_pay_amt)
FROM Invoice H, InvDtl D
WHERE H.inv_no = D.inv_no
AND H.inv_status in ('S', 'O')
AND D.dtl_status in ('S', 'X')
AND D.dtl_unit_pay_amt >= < cutoff value >

```

7.2. Applying the PTCR mutation operator

Let us illustrate the application of another mutation operator. Consider the relation between entity types *Normal Invoice* and *Settled Detail*. We apply the PTCR mutation operator. According to Figure 1, any entity of *Settled Detail* must be associated with one entity of *Normal Invoice*. According to the EER model, it is *not* possible to have a settled invoice detail without a corresponding invoice. Hence, the result of applying the mutation operator that toggles the participation requirement of *Normal Invoice* from “must-participate” to “non-participate” cannot result in a legitimate mutant.

On the other hand, each *Normal Invoice* may or may not be associated with any *Settled Detail*. Applying the mutation operator, it will generate a candidate fragment of a mutant that requires non-participation of *Settled Detail*.

$$\begin{aligned}
 & \text{Invoice.inv_status IN ('S', 'O')} \\
 & \text{AND NOT EXISTS (SELECT 1 FROM InvDtl D1} \\
 & \quad \text{WHERE D1.inv_no} = \text{Invoice.inv_no} \\
 & \quad \text{AND D1.dtl_status} = \text{'S'})} \\
 & \text{AND InvDtl.dtl_unit_pay_amt} \geq \langle \text{cutoff value} \rangle
 \end{aligned} \tag{3}$$

Here, “EXISTS (SELECT 1 ... WHERE ...)” is the standard SQL technique for checking the existence of an entity that fulfills a specified condition.

Thus, a second mutant M_2 can be formed using equation (3) instead of equation (2). However, the condition “InvDtl.dtl_unit_pay_amt >= < cutoff value >” on the last line of equation (3) uses a column of table *InvDtl*, which does not exist in this case. Testers must apply equation (1) in Section 6.2 to derive the value of *dtl_unit_pay_amt* instead. Moreover, since the data types of the relevant columns *Invoice.pay_amt*, *Invoice.inv_qty*, and *dtl_unit_pay_amt* are integers, the formulation of the mutant uses “Invoice.pay_amt >= Invoice.inv_qty * < cutoff value >” instead of “Invoice.pay_amt / Invoice.inv_qty >= < cutoff value >”. Furthermore, according to the EER model, the derived attribute *Invoice.pay_amt* is semantically equivalent to the sum of *InvDtl.pay_amt* attributes of relevant invoices. The sum of the former attribute readily substitutes the sum of the latter attributes (as far as each entity type in the relation is concerned). The other parts of the mutant M_2 follow a similar synthesis approach as above. Thus, the embedded

(a) Invoice Table

inv_no	inv_dt	inv_stl_dt	inv_status	inv_amt	inv_qty	pay_amt	cust_no	order_no
419	10/22/2004	12/31/2004	S	360	1	360	scsheung	qsic04-01
3554	10/18/2005	12/02/2005	S	360	1	360	wkchan	qsic05-02
304	09/22/2003	09/22/2003	S	360	1	360	thtse	qsic03-03
1	06/18/2006	null	X	1080	3	0	tiger	qsic06-04
711	04/04/2007	null	O	520	2	420	woods	qsic07-05

(b) InvDtl Table

inv_no	dtl_no	dtl_status	dtl_amt	dtl_qty	dtl_pay_amt	dtl_unit_pay_amt	order_no	order_dtl_no	conf_item
419	01	S	360	1	360	360	qsic04-01	01	mm test
3554	01	S	360	1	360	360	qsic05-02	01	db test
3554	02	X	200	2	0	0	qsic05-02	01	rasm test
1	01	X	1080	3	0	0	qsic06-04	01	tournament
711	01	O	100	1	99	99	qsic07-05	01	champion
711	02	S	420	1	420	420	qsic07-05	02	sponsorship

Figure 6. Sample instances of the invoice and invoice detail tables.

SQL part of M_2 will be:

```
SELECT sum(H.pay_amt)
FROM Invoice H
WHERE H.inv_status IN ('S', 'O')
AND NOT EXISTS (SELECT 1 FROM InvDtl D1
                WHERE D1.inv_no = H.inv_no
                AND D1.dtl_status = 'S')
AND (H.pay_amt >= H.inv_qty * <cutoff value>)
```

7.3. Weak mutation testing

Suppose that testers are interested in weak mutation testing [14, 21]. Informally, a weak mutant is said to be killed if the state of the mutant immediately following the execution of the mutated statement (or component) is not identical to that of the original program. Since we are interested in using SQL statements to distinguish different sets of data model constraints, we interpret that, for the same entity type applicable to both the original program and a mutant, if they affect different sets of records, the mutant is said to be killed. We elaborate the concept in more detail as follows:

In general, a program unit may involve a number of databases and hence a number of database schemas. Let C be a program unit with input domain D for its input parameters and $\Sigma = \langle \sigma_1, \sigma_2, \dots, \sigma_n \rangle$ be a tuple of database schemas. Following the description in Section 1, a DB application P is a tuple $\langle C, \Sigma \rangle$ in which each database schema σ_i in Σ is associated with an EER model E_i . A database instance is a concrete state of a database schema. The current program state of the DB application is the current program state of its program unit C with the current

database instances whose schemas are defined in P . For example, the program unit `getPaymentReceived()` with the database schema in Figure 3 constitutes the sample DB application for illustrating our unit testing technique. The parametric input domain for `getPaymentReceived()` is the set of valid integers.

By a similar token, a test case for P is a tuple $T = \langle x, \Psi \rangle$ in which x is an element in domain D and $\Psi = \langle \psi_1, \psi_2, \dots, \psi_n \rangle$ is a sequence of database instances such that each ψ_i corresponds to a state of the database schema σ_i of the DB application. For example, the database instance in Figure 6 with the input parameter `cutoff = 200` forms a test case for our sample DB application. The direct output of the test case from the program unit consists of SQL statements, which will be executed by a database system with the database instances to produce derived output, such as updated records in the database instances.

Let S be a SQL statement issued by the program unit C associated with the database schemas Σ . Let \bar{S}_i (such as M_2 in our example) be a mutated SQL statement of S after a mutation operator has been applied to S and the EER model E_i . Replacing S in C by \bar{S}_i produces a mutant $\bar{C}_{\bar{S}_i}$ of the program unit C . The tuple $\langle \bar{C}_{\bar{S}_i}, \Sigma \rangle$ forms a mutant of the DB application $P = \langle C, \Sigma \rangle$.²

We note that a sequence of database instances, say Ψ' , is part of a state of the DB application under test and, hence, we distinguish mutants from the original program using the intermediate state at the point where the SQL statement is issued from the program unit. We may consider our

² We observe that neither a database schema nor an EER model is changed in our approach.

approach as a type of weak mutation test data selection strategy.

Let $f(\cdot)$ be a function that accepts a SQL statement S , a sequence of database schemas, and a sequence of their corresponding database instances to return a set of records manipulated by S , such that each record is annotated with the relevant SQL statements affecting it. Let $S(P, T)$ denote an instance of S when the DB application P accepts a test case T . A mutant \bar{S}_i of S is said to be killed if $f(S(P, T), \Sigma, \Psi')$ and $f(\bar{S}_i(P, T), \Sigma, \Psi')$ differ. The mutant $\bar{C}_{\bar{S}_i}$ of the program unit C is said to be killed if the SQL-oriented mutant \bar{S}_i is killed. A mutant $\langle \bar{C}_{\bar{S}_i}, \Sigma \rangle$ of DB application $P = \langle C, \Sigma \rangle$ is said to be killed if the mutant $\bar{C}_{\bar{S}_i}$ is killed.

Let us continue our example to distinguish mutants from an original SQL statement at the database instance level. Consider the database instances shown in Figure 6 with the input parameter `cutoff = 200`. The faulty sample program `getPaymentReceived()` will select *settled invoices* with invoice numbers 419 and 3554. On the other hand, the mutant M_2 will select the settled invoice with invoice number 304. Since the records selected from the entity type *Settled Invoice* are not identical, the mutant is killed. This missing settled invoice reveals that the faulty sample program cannot retrieve a desirable record from the *Invoice* table. For this test case, the faulty DB application produces a parametric output of 1140, which is the sum of the 1st, 2nd, and 6th rows of the *InvDtl* table in Figure 6. The expected value should be 1500, since the function is expected also to select invoice number 340. Thus, the test case reveals a failure.

We would like to add an interesting epilogue. Suppose the faulty program is corrected, by substituting the expected SQL query in Figure 5 into the relevant part on lines $s_5 - s_{10}$ of Figure 2. Since the correct statement is a union query, given an input `cutoff = 200` for the database instance shown in Figure 6, it will return two rows, one from each sub-query. The two returned rows are 1140 (based on the 1st, 2nd, and 6th rows of the *InvDtl* table) and 360 (based on the 2nd row of the *Invoice* table). As the order of the record set is not specified in the SQL statement, it can be the row of 1140 followed by that of 360, or vice versa. We discover another fault on line s_{13} of Figure 2. It should be a `while` statement instead of an `if` statement; otherwise, it cannot process the results from both sub-queries in the same run. Suppose the first row in the record set is 1140. Similarly to the description in the above paragraph, it will miss the row due to the second sub-query. In this case, the failure is detected by mutant M_2 . On the other hand, suppose the first row in the record set is 360. The faulty program will miss to report those invoices with invoice details. In this situation, the failure is detected by mutant M_1 instead.

8. Concluding discussions

Testing database application programs with SQL as the tools to access and modify database instances is gaining an increasing amount of research attention. This paper proposes a novel fault-based testing approach to address the special characteristics and challenges for testing DB applications at the unit level.

The notion of mutating statements for mutation testing of a program has a long history. It should not be difficult to apply conventional mutation operators such as replacing a logical connector by another syntactically legal logical operator (say, “`Invoice.inv_no = InvDtl.inv_no`” by “`Invoice.inv_no <> InvDtl.inv_no`”) to produce SQL-related mutants. Hence, we assume that such mutation operators can be defined without problem.

Furthermore, non-database-related statements can be regarded as conventional statements of the programming language in question. They can be tested using techniques applicable to that programming language. For instance, by ignoring embedded SQL statements, testers can apply predicate testing to test Boolean expressions in a database application program as if it were a conventional program. Similarly, they can also apply conventional mutation testing to test non-SQL statements.

We are currently doing experimental studies to evaluate the effectiveness of our approach. Our initial investigation is conducted on the sample application presented in this paper. We mechanically generate 35 non-equivalent mutants for this subject program, where a mutant is said to be non-equivalent if its outcome is different from that of the original program. For the purpose of control, we also apply conventional mutation operators to mechanically generate 34 conventional non-equivalent mutants. On average, it needs 4 test cases to kill all the semantic mutants, and 6 test cases to kill all the conventional mutants. The case study shows that our approach detects 89.5% of faulty (but not mutated) versions of the subject program, whose faults are not limited to embedded SQL statements. On the other hand, the conventional approach detects 78.9%. We also find that, for the two mutation operators explained in this paper, PTCR is more effective than GSDR, which, in turn, is more effective than conventional operators. We must emphasize, however, that the preliminary results are based on a limited amount of data. We shall report our major findings when we have accumulated more experience.

We focus, therefore, on semantic mutants of embedded SQL statements based on the semantics of the conceptual data model. We employ the standard types of constraint of the EER model to formulate mutation operators. Generally speaking, semantic information, which is invaluable to software testers, is not as extensively available in database schemas as in conceptual data models. For various reasons

such as performance [9], some desirable constraints are not implemented in the database schemas associated with the programs under test. To the best of our knowledge, this work is the first of its kind to consider semantic relationships defined in conceptual data models to generate mutants for fault-based testing of DB applications.

We exploit the semantic information captured from the conceptual data model, such as the relationships between the stored and derived attributes, and combine them with embedded SQL statements to produce SQL-oriented mutants. Specifically, we analyze the SQL statements to be mutated, and then find out the affected entity types, relations, and constraints. Based on the analysis results, mutation operators are applied to the SQL statements to produce mutants. Without such semantic transformation, some useful mutants, such as M_2 of the sample application, are difficult to formulate.

In view of the preference by testers to check database instances of a DB application instead of checking SQL data manipulation statements in its program unit, we use database instances plus parametric inputs and derived outputs to conduct fault-based testing to kill SQL-oriented mutants at the database instance level.

We have illustrated our proposal through a SQL query example. However, subquery and join conditions of SQL data manipulation statements, including INSERT, UPDATE, and DELETE statements, share a lot of similarities with query statements. The discussions in this paper can be applied to these language constructs. Nevertheless, more thorough investigations are recommended.

We shall further construct and evaluate formal models to represent legal constraints and business logics for testing purpose. Since we have not yet come up with a satisfactory technique to derive non-equivalent SQL-related mutants automatically, we shall seriously address this issue. We believe that, since SQL is a language with formal semantics, it should allow us to analyze and create non-equivalent mutants, rather than generating a large class of equivalent mutants. We shall conduct experiments to determine the fault-detection capability of our proposal. Testing DB applications with data definition statements will also be an area to be examined. The issue of automatic test oracles will also be addressed. Finally, we shall construct a prototype tool to implement our approach.

Acknowledgement

We would like to thank Yong Jian Wu for conducting experiments to help us improve the paper.

References

- [1] Information Technology — Database Languages — SQL — Part 2: Foundation (SQL/Foundation). Document No. ISO/IEC 9075-2-2003, International Organization for Standardization, 2003.
- [2] SQLUnit Project Home Page. <http://sqlunit.sourceforge.net/>, 2005.
- [3] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: a methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2): 166–182, 1990.
- [4] M.Y. Chan and S.C. Cheung. Testing database applications with SQL semantics. In *Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications (CODAS '99)*, pages 363–374. Wollongong, Australia, 1999.
- [5] R. Chatterjee, G. Arun, S. Agarwal, B. Speckhard, and R. Vasudevan. Using applications of data versioning in database application development. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 315–325. IEEE Computer Society Press, Los Alamitos, California, 2004.
- [6] D. Chays, Y. Deng, P.G. Frankl, S. Dan, F.I. Vokolos, and E.J. Weyuker. An AGENDA for testing relational database applications. *Software Testing, Verification and Reliability*, 14(1): 17–44, 2004.
- [7] F. Chen, B.G. Ryder, A. Milanova, and D. Wonnacott. Testing of Java web services for robustness. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA 2004)*, pages 23–34. ACM Press, New York, 2004.
- [8] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: help for the practicing programmer. *IEEE Computer*, 11(4): 34–41, 1978.
- [9] Y. Deng, P.G. Frankl, and D. Chays. Testing database transactions with AGENDA. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 78–87. IEEE Computer Society Press, Los Alamitos, California, 2005.
- [10] Y. Deng, P.G. Frankl, and J. Wang. Testing web database applications. *ACM SIGSOFT Software Engineering Notes*, 29(5): 1–10, 2004.
- [11] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison Wesley, Reading, Massachusetts, 2003.

- [12] R. A. Haraty, N. Mansour, and B. Daou. Regression testing of database applications. *Journal of Database Management*, 13 (2): 31–42, 2002.
- [13] R. A. Haraty, N. Mansour, and B. Daou. Regression test selection for database applications. In volume 3 of *Advanced Topics in Database Research*, K. Siau (editor), pages 141–165. Idea Group, Hershey, Pennsylvania, 2004.
- [14] W.E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8 (4): 371–379, 1982.
- [15] G.M. Kapfhammer and M.L. Soffa. A family of test adequacy criteria for database-driven applications. In *Proceedings of the Joint 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC 2003/FSE-11)*, pages 98–107. ACM Press, New York, 2003.
- [16] L.J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16 (8): 844–857, 1990.
- [17] A. Neufeld, G. Moerkotte, and P.C. Lockemann. Generating consistent test data: restricting the search space by a generator formula. *The VLDB Journal*, 2 (2): 173–214, 1993.
- [18] J. Offutt and W. Xu. Generating test cases for web services using data perturbation. *ACM SIGSOFT Software Engineering Notes*, 29 (5): 1–10, 2004.
- [19] M. A. Robbert and F.J. Maryanski. Automated test plan generator for database application systems. In *Proceedings of the 1991 ACM SIGSMALL/PC symposium on Small Systems (SIGSMALL 1991)*, pages 100–106. ACM Press, New York, 1991.
- [20] M.J. Suárez-Cabal and J. Tuya. Using a SQL coverage measurement for testing database applications. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12)*, *ACM SIGSOFT Software Engineering Notes*, 29 (6): 253–262, 2004.
- [21] M.R. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Proceedings of the 2nd Workshop on Software Testing, Analysis and Verification*, pages 152–158. IEEE Computer Society Press, Washington DC, 1988.
- [22] S.J. Zeil. Testing for perturbations of program statements. *IEEE Transactions on Software Engineering*, 9 (3): 335–346, 1983.
- [23] J. Zhang, C. Xu, and S.C. Cheung. Automatic generation of database instances for white-box testing. In *Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC 2001)*, pages 161–165. IEEE Computer Society Press, Los Alamitos, California, 2001.
- [24] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Survey*, 29 (4): 366–427, 1997.