The work appears as M.Y. Chan and S.C. Cheung, Testing Database Applications with SQL Semantics, in the *Proceedings of 2nd International Symposium on Cooperative Database Systems for Advanced Applications (CODAS'99)*, Wollongong, Australia, March 1999, pp. 363-374.

Testing Database Applications with SQL Semantics

M.Y. Chan and S.C. Cheung

HKUST
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{mandy, scc}@cs.ust.hk

Abstract

Testing of database applications is crucial for ensuring high software quality as undetected faults can result in unrecoverable data corruption. The problem of database application testing can be broadly partitioned into the problems of test cases generation, test data preparation and test outcomes verification. Among the three problems, the problem of test cases generation directly affects the effectiveness of testing. Conventionally, database application testing is based upon whether or not the application can perform a set of predefined functions. While it is useful to achieve a basic degree of quality by considering the application to be a black box in the testing process, white box testing is required for more thorough testing. However, the semantics of the Structural Query Language (SQL) statements embedded in database applications are rarely considered in conventional white box testing techniques. In this paper, we propose to complement white box techniques with the inclusion of the SQL semantics. Our approach is to transform the embedded SQL statements to procedures in some general-purpose programming language and thereby generate test cases using conventional white box testing techniques. Additional test cases that are not covered in traditional white box testing are generated to improve the effectiveness of database application testing. The steps of both SQL statements transformation and test cases generation will be explained and illustrated using an example adapted from a course registration system. We successfully identify additional faults involving the internal states of databases.

1 Introduction

The increase in both application complexity and reliability expectation has contributed to great demands on software testing activities. Efficient and effective software testing is crucial within the software development and maintenance cycle. A majority of software applications is database applications. Testing of database applications is of great importance in both the development and production phase since undetected faults in these applications may result in incorrect modification or accidental removal of crucial data. Once the data are mistakenly modified, the error may propagate and lead to more data corruption if it is left undetected. Since not all transactions can be unrolled, restoring data from database backups cannot eradicate the problem. Functional dependency rules are well known to be capable of enforcing database integrity, but the costs incurred are generally too expensive for non-trivial commercial database systems [1].

Both static and dynamic tests should be designed for database applications. Static activities such as inspection and verification validate whether both the application program and database design meet all the functional and the data requirements and there is no conflict between these requirements [2]. Furthermore, the applications should restrict unauthorized access [3], allow concurrent access, and support data recovery after hardware or software failures. Although various studies have been conducted to investigate static testing techniques for database design, relatively little attention have been made to explicitly address the dynamic testing of database applications. A study has thus been launched to examine the effectiveness of applying several popular software testing techniques to database applications. Consequently, we propose a testing approach that transforms the embedded SQL statements in database applications to procedures in a general-purpose programming language (GPL). The objective of the transformation is to include the semantics of the SQL statements such that more test cases can be generated to reveal faults relevant to the changes of internal database states. The approach allows test cases that take care of the semantics of both the GPL statements and the SQL statements to be generated using conventional white box testing techniques and hence helps reveal more faults contained in database applications. Particularly, it detects faults happened in some specific internal database states.

The paper is organized as follows. Section 2 presents the mechanisms and issues of testing database applications. Section 3 summarizes related software testing approaches including the black box approach, the traditional white box approach and the conceptual model approach. In section 4, we propose the new testing approach. A simple example will be used to illustrate that the proposed approach can reveal additional faults in database applications in section 5. Conclusion and possible future work will be given in section 6.

2 Database Application Testing

Database applications can generally be classified into two categories. The first category consists of applications that are solely built in Data Manipulation Language (DML) and the language supported by the DBMS. In these applications, queries and transactions are specified in DML and user interfaces are defined in the DBMS language. The other category involves applications that are built in both DML and general-purpose programming languages such as C, C++ and Java [4]. Statements of Structured Query Language (SQL), a powerful declarative query language, are embedded in programs written in any general-purpose language, which is referred to as the host language [4]. In these applications, queries and modifications of data are written as the embedded SQL statements. Other functionality like interacting with users and sending results to a graphical user interface is written in GPL. In this paper, we concentrate on testing this category of database applications.

Testing of database applications is different from the testing of structural programs. The inputs of database applications involve both the user inputs and the database instances. As stated in [9], in addition to checking the outcome with the expected outcome, programmers or testers should also check if the database is consistent and reflects the original environments in database application testing. The problem of database application testing can be divided into three sub-problems as shown in Figure 1. They are the problems of Test Cases Generation, Test Data Preparation, and Test Execution and Outcome Verification.

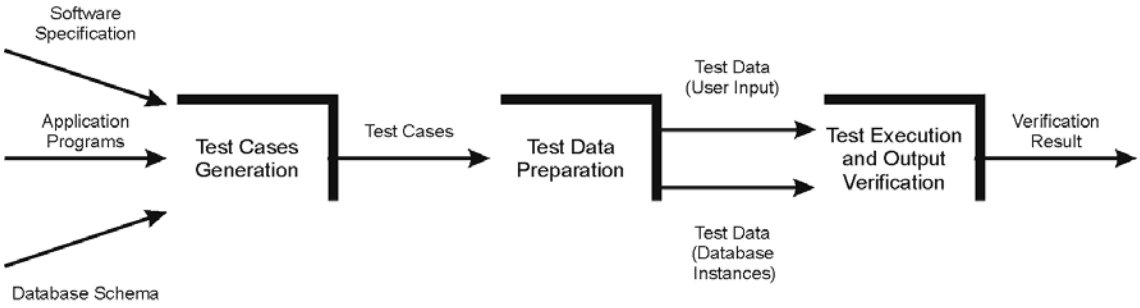

Figure 1: Sub-problems of database application testing

Among the three sub-problems, the problem of test cases generation has priority over the others to be solved since the effectiveness of database application testing is directly related to the test cases generated. As discussed in the Introduction, undetected faults in database applications may cause serious damage to critical production data. In this paper, we focus on the problem of test cases generation.

Conventionally, black box testing techniques are employed to generate functional test cases. White box testing is required to achieve a higher degree of software quality and reliability. In [8], Robbert

and Maryanski proposed a conceptual model approach to generate database application test cases. In the next section, these approaches are summarized and discussed.

3 Related Works

3.1 The Black Box Approach

Nowadays, most database applications are tested using black box testing techniques [10][11] such as Equivalence Partitioning, Boundary Value Analysis, and Cause-effect Graphing techniques. Test cases are derived without reference to the construction of application programs. According to the software specification, test cases are produced to test the functionality *independently*. It is typically used to check if the application conforms to its specification [12]. Examples of applying black box techniques in testing database applications were summarized in [9].

One advantage of black box testing techniques is that test cases can be generated independent of the programs at an earlier stage of the software development cycle. Programmers can have the test cases in mind when they develop an application. Thus, the resultant application tends to satisfy most of the generated test cases. Relatively few efforts are required to test and debug the applications. Another advantage is that the costs of generating test cases using black box techniques are rather low as compared to the costs associated with white box techniques. Thus, the development cost can be reduced. Besides, many black box testing techniques have already been well designed. Testers can select the most suitable one for testing their database applications.

On the other hand, without examining the programs, we do not know how much of an application is being tested. Practically, functional specifications are usually specified in natural language. As mentioned in [13], using natural language to express functional specification causes problems including redundancy, inconsistency, and incompleteness. In addition, most black box testing methods are insensitive to some kinds of faults [14]. Common kinds of faults can still go undetected even when all the tests defined by the testing method succeed [15]. A typical example is the kind of faults that only happens in some specific sequences of execution of functions. A simple menu-driven C++ application shown in Figure 2 illustrates the problem. The application is designed to support three functions *F1*, *F2*, and *F3*. In black box testing, each function may be tested independently in a randomly chosen order, say *F1*, *F2*, and *F3*. These functions are claimed to be correct when we have executed all the defined tests successfully. However, as indicated in [15], absolute correctness cannot be established by testing (based on the halting problem in which it is proven that no general mechanical procedure can determine in advance whether a program will eventually halt). Only relative correctness with respect to significant and comprehensive classes of faults can be accomplished. Besides, in many testing techniques, only a subset of the inputs, rather than all inputs, is tested to imply the correctness of the functionality. Thus, these functions can only be claimed as

relatively correct rather than absolutely correct. In the production phase, when these functions are run in another order (for example, *F2*, *F1*, *F3*), some of the functionality may fail. For instance, if the database is modified in *F2* and *F1* does not work in that modified state, the program fails when *F1* is executed after *F2*. It is highly possible that black box testing will miss revealing this kind of faults.

```
void F1();
void F2();
void F3();

int main(int argc, char** argv) {
  int option;
  cin >> option;
  while (option != QUIT) {
    switch (option) {
      case 1:
        F1(); break;
      case 2:
        F2(); break;
      case 3:
        F3(); break;
    }
    cin >> option;
  }
  return 0;
}
```

```
Testing Sequence of option:
      1 2 3

Testing function sequence:
      F1()
      F2()
      F3()



Another Sequence of option:
      2 1 3

Another function sequence:
      F2()
      F1()
      F3()
```

Figure 2: A simple program to illustrate the problem of black box testing techniques

This kind of faults is common in database applications. A function of a database application (say A) is said to be dependent on other functions (say, B and C) when the set of relations the former accesses overlaps with that accessed by the latter. The outcome of the function A may be affected by the functions B and C as the latter may change the internal state of the database. The function may fail after some sequences of execution of other functions. As such, it is necessary to look for techniques beyond black box testing.

3.2 The Traditional White Box Approach

White box testing techniques like statement testing, branch testing, condition coverage, and path testing [12] can be employed to test some portions of or the whole database application to attain a more complete testing of the application. White box testing lets testers examine the code in detail and make sure that at least a certain degree of test coverage such as execution of every statement has been achieved [12].

Nevertheless, traditional white box techniques have their own limitations in testing database applications. Most importantly, they have not explicitly considered the SQL statements embedded in application programs. The SQL statements are treated as black boxes. Figure 3 shows a code

segment of a C program where SQL statements are embedded in the DB-Library calls from line 7 to line 11 to access a database. Usually, only a few test cases are generated by white-box techniques to test the embedded SQL statements. No test cases are generated intentionally to include the semantics of the SQL statements, which reflect the changes of the internal database states. We suspect traditional white box testing may miss the type of faults related to the internal database changes.

```
 1  void process(DBPROCESS* dbproc) {
 2    RETCODE result_code;
 3    char sid[IDLEN+1];
 4    char cid[IDLEN+1], cname[NAMELEN+1];
 5    fprintf(IN_CH, "\nPlease enter the student ID: ");
 6    fscanf(IN_CH, sid);
 7    dbfcmd(dbproc, "select COURSE.CID, COURSE.name");
 8    dbfcmd(dbproc, " from TAKE, COURSE");
 9    dbfcmd(dbproc, " where COURSE.CID = TAKE.CID and");
10    dbfcmd(dbproc, " TAKE.SID = '%s'", sid);
11    dbfcmd(dbproc, " order by TAKE.semester");
12    dbsqlexec(dbproc);
13    while ((result_code = dbresults(dbproc)) != NO_MORE_RESULTS) {
14      if (result_code == SUCCEED) {
15        dbbind(dbproc, 1, NTBSTRINGBIND, (DBINT)0, (BYTE*)cid);
16        dbbind(dbproc, 2, NTBSTRINGBIND, (DBINT)0, (BYTE*)cname);
17        if (dbnextrow(dbproc) == NO_MORE_ROWS)
18          fprintf(OUT_CH, "Data not found.\n");
19        else {
20          fprintf(OUT_CH, "Student with ID %d has taken the ", sid);
21          fprintf(OUT_CH, "following course(s):\n");
22          fprintf(OUT_CH, "\t%s\t%s\n", cid, cname);
23          while (dbnextrow(dbproc) != NO_MORE_ROWS)
24            fprintf(OUT_CH, "\t%s\t%s\n", cid, cname);
25        }
26      }
27    }
28  }
```

Figure 3: A code segment of a C program embedded with SQL statements

3.3 The Conceptual Model Approach

Both of the aforementioned approaches do not utilize database schemas in designing test cases for database applications. In [8], Robbert et al proposed a cyclic testing scheme, which generates test cases based on the conceptual database model such as the Entity-Relationship (ER) model of the application database. In this approach, the test configuration is produced by dividing the system structure into subgroups that can be independently certified. A set of tests is enumerated for each subgroup. When a means of testing is provided for all constructs of the conceptual model, the testing is said to be exhaustive. A matrix form can be used to demonstrate the completeness of the testing. Syntax testing is also employed to define test cases with both complex and rigid data structure definitions.

The generated test plan includes inheritance specification, property description and constraint specifications. It ensures that exceptional conditions, extremes, and boundary cases are tested, valid inputs are accepted, invalid inputs are rejected, and all entities and relationships are exercised. An example of the test plan for the entity *GRAD_STU* is shown in Figure 4.

The approach is targeted to generate test cases for testing database applications. Making use of the conceptual model of the application database, test cases can be designed to test all constructs of the ER model and the constraints of the databases. With the input of the user-defined functions, test cases can be generated to test the database functions supported by each entity.

```
STUDENT is the supertype of GRAD_STU

The structure of the supertype relation requires that a GRAD_STU
inherits all the following attributes:
ID, NAME, GPA, REG_CARD, ADVISOR, CREDITS

Check for the existence and appropriateness of each attribute.

The inherited operations of GRAD_STU are:
REGISTER, GRADUATE, DROP_OUT, ENROLL, GPA

Check each operation:
REGISTER an existing GRAD_STU
REGISTER a non-existing GRAD_STU
GRADUATE a valid GRAD_STU who meets the requirements exactly
GRADUATE a valid GRAD_STU who is overqualified
GRADUATE a valid GRAD_STU who does not meet one of the
requirements.
…

The additional operations available to a GRAD_STU are:
TEACH

Check the operation …
```

Figure 4: An example of the test plan for the entity GRAD_STU

Nevertheless, Robbert et al have not provided in [8] details for selecting test cases and determining the completion point of testing. While the authors have defined the definition of exhaustive testing, it does not explain the procedures of test cases generation. Different testers may thus produce different sets of test cases in order to examine all constructs of the conceptual model. This may significantly influence the effectiveness of testing. Like the black box testing counterpart, this approach may miss those faults arisen from improper execution sequences. Instead of testing the functionality in a random order as in black box testing, an optimal set of test sequences is defined by the test case generator.

4 The WHODATE Approach

Conventional white box testing techniques have been well established for general software applications. To cater these techniques for database applications, we propose the mechanism WHODATE, which stands for WHite bOx Database Application TEsting, to generate additional test cases from embedded SQL statements. The mechanism depicted in Figure 5 works as follows. Given a database application with embedded SQL statements, these SQL statements are transformed into GPL statements. Conventional white box techniques are then applied to both the transformed statements and other statements written in the host language. One of our goals is to include the semantics of the SQL statements in test cases generation. In other words, we test whether the semantics of the SQL statements combined with the GPL statements are correct. Researchers may wonder whether the SQL statements can be treated as black boxes and be tested separately. This approach may not be possible because no functional specification is defined for the SQL statements. Moreover, SQL may not be strong enough to perform all database functionality and GPL statements may have to be used to complement with SQL statements in performing some functionality.

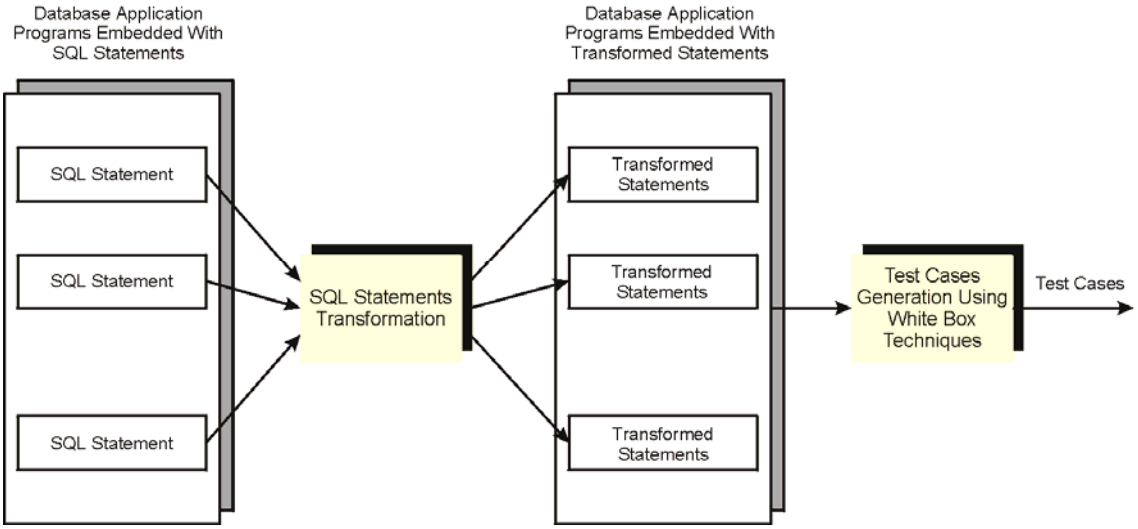

Figure 5: The WHODATE approach: Transformation of SQL statements and test cases generation using both the transformed statements and other statements in the programs

4.1 Transformations Between SQL Statements and Relational Algebraic Expressions

In order to facilitate the transformation of SQL statements, they are first converted into equivalent Relational Algebraic Expressions (RAEs). Due to the slight variation between the sets of SQL commands supported by different major DBMSs, RAE is employed as a common interface language for the transformation process. In fact, RAEs have been widely used to represent database queries. The transformation between SQL statements and RAEs can be found in most database textbooks [2]

[4]. RAEs are composed of fundamental relational algebraic operations (RAOs). These operations include *Selection*, *Projection*, *Union*, *Difference*, *Cartesian Product*, and *Rename*. Their notations and definitions were given in [4].

There may exist more than one RAE equivalent to the SQL statement. The choice of the equivalent RAE may influence the transformation of SQL statements to GPL statements. In later subsections, we will discuss whether it affects the effectiveness of the test cases generated.

Besides, SQL is more powerful than relational algebra. In some cases, we cannot transform SQL statements into RAE expressions using the fundamental RAOs. In these situations, we define additional RAOs to support the transformation. For instance, the RAO *S* is defined for the SQL phrase *ORDER BY* in [9]. In the present work, we treat those operations of the SQL statements as black boxes in case the operations are either too difficult to define or simply cannot be defined.

4.2 Transformations Between RAEs and GPL Statements

A query evaluation plan (QEP), which describes the query computation, can be expressed in terms of set-oriented operations [16]. In order to be executed efficiently by the physical database processor (PDBP) of a DBMS, the plan must be translated into a program that operates on tuples as its basic objects. Freytag et al in [16] have developed two algorithms to translate QEPs into iterative programs directly executable on the PDBP. A set of procedures that access data using tuple-oriented operations by the PDBP is utilized to independently implement the various set-oriented operators permitted in QEPs. Similarly, the relational operations in RAEs are translated into a set of C++ procedures.

In relational databases, data are usually stored in the form of records [4]. In our translation, data structures are defined to store the tuples of records in a relation. The classes *Relation* and *Record* are defined in [9] to store the tuples of records. *Relation* is the super class of all types of relations while *Record* is the super class of all kinds of records. *Relation* keeps a list of records and supports functionality including insertion, removal and sorting of tuples. *Record* maintains attributes of different primitive data types and supports concatenation and comparison of records.

Each fundamental algebraic operation in RAEs is transformed to a procedure that takes one or more relations as inputs and returns one new relation. In addition to relations, some procedures may receive other sorts of parameters like the selection criteria. Here is the definition of procedure for the fundamental algebraic operation Cartesian_Product. The definitions for other fundamental RAOs are shown in [9]. Note that the symbol "+" equals the concatenation operator of a record.

Cartesian_Product

```
Relation& Cartesian_Product(Relation& r, Relation& s) {
  Relation* newR = new Relation;
```

```
    for (int i=0; i<r.getNoOfTuples(); i++) {
      for (int j=0; j<s.getNoOfTuples(); j++) {
        newR->insert(r.getTuple(i) + s.getTuple(i));
      }
    }
    return *newR;
  }
```

The first step of the transformation is to pass one or more relations as parameters to one of these procedures. The new relation that is outputted by this procedure will be passed as a parameter to another procedure. Another relation may be passed together with this new relation if necessary. The transformations between RAOs and iterative procedures continue in a similar manner until all the operations of the relational algebraic expression have been replaced. Examples of transformations are shown in [9].

4.3 Generation of White Box Test Cases

After RAEs have been transformed to procedures in GPL language, different white box techniques could be used to produce test cases from the aggregation of the transformed procedures and the host statements. The choice of the techniques may affect the effectiveness of the WHODATE approach. In the next section, a case study will be conducted to demonstrate the use of statement testing, branch testing and path testing in generating test cases for database applications. Investigation will be performed to compare the effectiveness of different white box techniques in the near future.

In general, when carrying out white box testing, it is convenient to examine the flow graph of a program that shows the flow of control through the program using a network of nodes and edges. The execution paths and the corresponding test cases are computed from the graph of both the translated procedures and the host statements to meet the coverage requirements of different white box techniques. In conventional white box testing, each SQL statement is treated as a single node when test cases are produced. In the WHODATE approach, the flow graph of each SQL statement is composed of the flow graphs of all procedures of the RAE as shown in Figure 6.

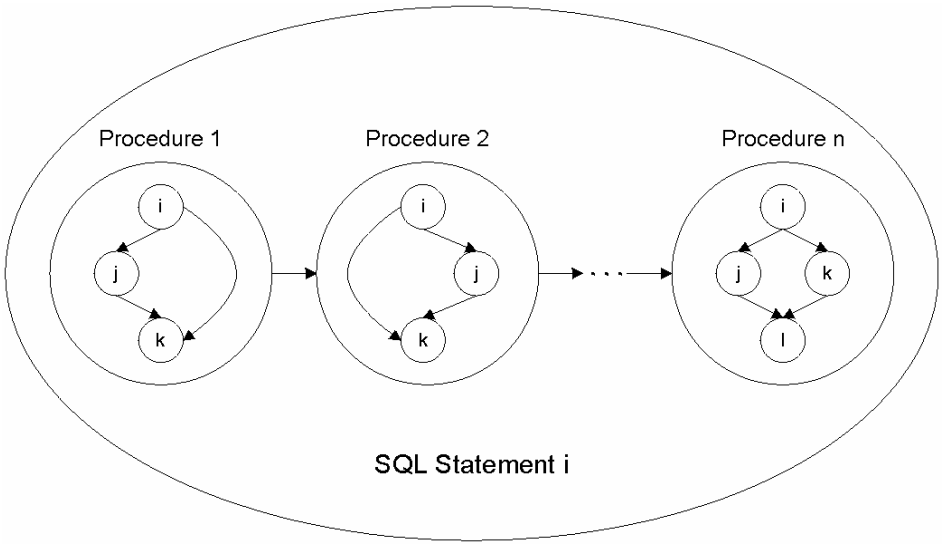

Figure 6: The complete flow graph of all procedures of the SQL statement i

The definition of completion point of the testing depends on the white box techniques employed in test cases generation. For instance, if statement testing is chosen, the testing will be completed when the test cases produced execute every transformed and host statement at least once. As another example, if path testing is used, the generated test cases should cause execution of all paths in the transformed and host statements.

Similarly, the test coverage relies on the chosen white box technique. We can achieve larger test coverage when path testing rather than branch testing is used. The test effort required is also flexible. More efforts have to be paid to generate test cases when the test coverage required is larger for critical database applications. Fewer efforts are needed when the test coverage required is medium for simple database applications. Practitioners should decide which white box techniques to be employed in test cases generation according to the test coverage they want to achieve.

Compared with the traditional white box approach, using particular testing techniques like statement coverage and branch coverage in our approach generates more or less the same number of test cases. However, when some kinds of white box testing methods like path testing are employed, the number of test cases generated is a few times of that produced by the same techniques in the conventional approach. The number of test cases generated using path testing with coverage of equivalent classes of paths in our approach is $\prod_{i=1}^{n} b_i$ times as much as that produced in the conventional white box approach in the worst case (b_I refers to the number of test cases generated by the SQL statement *i* while *n* is the total number of SQL statements in the program). Suppose three paths (*1*-*3*) are generated from the flow graph of the original program whereas another three paths (*A*-*C*) are

generated from the flow graph of the procedures of the SQL statement as shown in Figure 7. The final set of test cases is the nine resultant execution paths (*1A, 1B, …, 3B, 3C*).

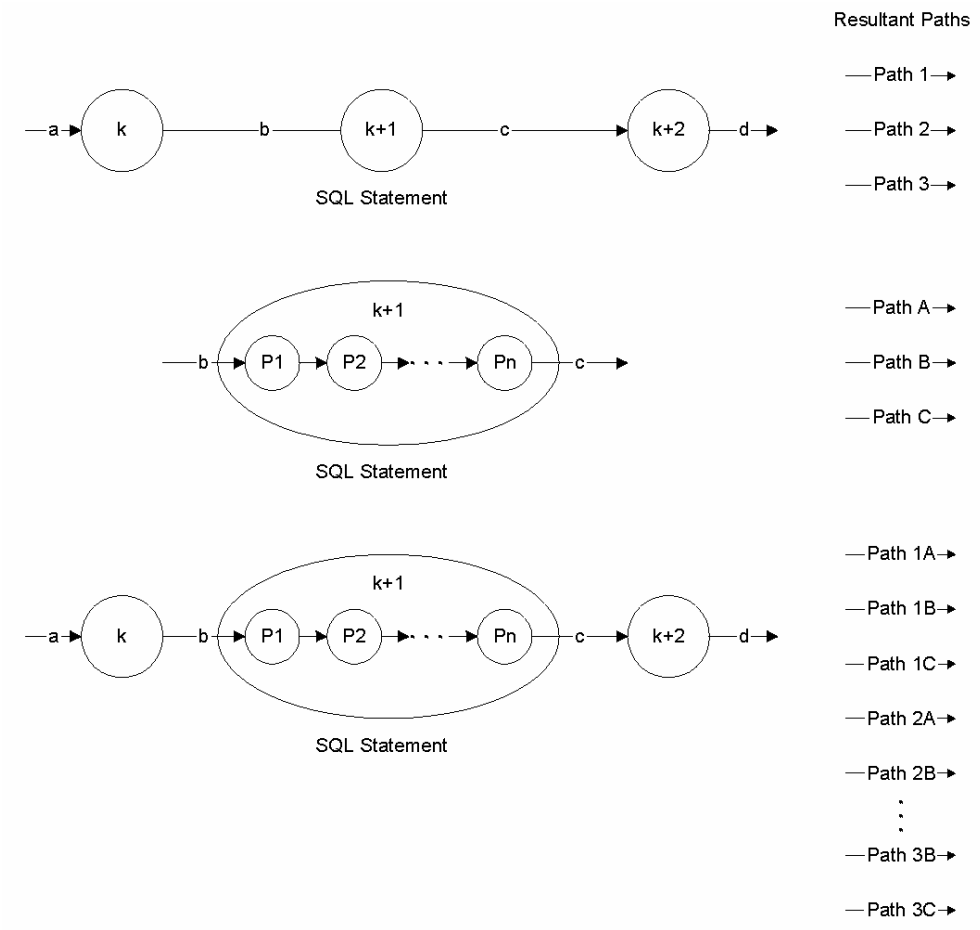

Figure 7: Resultant execution paths of the transformed procedures and host statements

To reduce the number of test cases generated while maintaining the effectiveness of our testing, the conceptual model of the database application could be employed. Further studies should be made to utilize the conceptual model to produce more effective test cases.

Apparently, our proposed method aims at discovering faults of the SQL statements embedded in a database application. This complements conventional techniques of white box testing, which does not specifically consider the semantics of embedded SQL statements in test cases generation. Besides, we can reveal faults that only occur in some particular states of the database as the number of tuples in relations is considered in test cases generation.

In many cases, the functional specifications of database applications are vague and informal, the test cases generated from these specifications may miss significant faults in the applications. The sets of test cases generated by different testers may be different and the effectiveness of the test cases may

depend on the skill of testers. Using the proposed approach, additional test cases can be generated systematically from the transformed procedures of the SQL statements. The variations that may be caused in the transformations between SQL statements and RAEs do not influence the effectiveness of the testing greatly because the flow graphs of all the procedures in the RAEs are employed in test cases generation. More importantly, as the combination of the transformed procedures and other programming statements are considered when generating test cases, our approach can discover faults that occurs only when some functions are being executed in some particular sequences.

5 A Case Study

In [9], we illustrated the effectiveness of the WHODATE approach using a programming assignment submitted by our undergraduate students in a database course. Experiments were conducted upon its combination with various white box testing techniques, namely statement testing, branch testing and path testing. The assignment is to design the schemas and develop five functions of a database system to keep track of the information about teachers, students and courses of an education centre. One function is to read a student ID from the terminal and then list all courses taken by the student so that each course appears after its prerequisite courses. Figure 3 shows the incorrect codes written by a programmer (one of the undergraduate students) to perform this function.

Compared with conventional white box techniques, more test cases can be generated [9]. In particular, by including the semantics of the embedded SQL statements, test cases are produced to reveal faults relevant to the changes of internal database states. Traditional testing approaches usually overlook this type of faults. For example, the programmer who wrote the incorrect codes in Figure 3 might assume that each student was allowed to take a course only after s/he had taken its prerequisite courses in previous semesters. In normal situation, the assumption is valid and the application produces the correct outcome. However, the student might fail the prerequisite course and s/he may be approved to take a course in the current semester and retake its prerequisite course in the next semester in special situations. In this case, the application cannot list the courses taken by the student in the correct way.

For example, the course `102` is the prerequisite of the course `251`. The student with ID `DS9001` has failed the course `102` in the Fall semester of 1997. However, he is allowed to take the course `251` in the Spring semester of 1998 and retake the prerequisite course `102` in the Fall semester of 1999. In this situation, the following incorrect outputs are produced by the application. First, the course `102` has outputted twice. Moreover, the course `251` appears before its prerequisite course `102`.

```
Please enter the Student ID: DS9001
The student with ID DS9001 has taken the following course:
        101             Computer Applications
```

```
102             C Programming
111             Software Tools
251             Software Engineering
102             C Programming
```

As shown in [9], our testing approach requires the student to take more offerings that the total number of courses in the database. Thus, we generate the test case where some of the students have to retake at least some of the courses. While there is no such requirement for conventional white box testing, it cannot reveal this fault. Both independent testing of the SQL statements and black box testing also do not work since the functional specification may not be clear to state this requirement.

In addition, we also reveal the repeated output of the same course even when the student retakes a course in the same semester. These kinds of faults may not be detected by black box testing since functional specifications are ambiguous in most situations.

6 Conclusion

The problem of database application testing can be partitioned into the problems of test cases generation, test data preparation and test execution and outcome verification. Traditionally, functional testing is employed to generate test cases to test database applications while the SQL statements embedded in the application programs are not specifically considered in white box testing. In [8], a new testing approach utilizing the conceptual schema of the database in test cases generation has been introduced to test database applications. A thorough investigation of these testing approaches has been conducted and we find that all these approaches have weaknesses in testing database applications.

In this paper, we propose the new testing approach WHODATE that transforms SQL statements to procedures in general-purpose programming language and apply conventional white box techniques on both these transformed procedures and the host statements to generate test cases. The process of transformations and methods of test cases generation is presented. In addition, a case study is given to illustrate the effectiveness of the testing approach. We can see that extra effective test cases are generated to reveal more faults compared with the conventional approaches. Besides, the test cases generated help reveal faults related to the internal states of databases in database applications. To conclude, we should employ black box techniques to verify that database applications fulfill their functional requirements while the proposed technique should be complemented with white box techniques to reveal faults relevant to the internal states of databases.

More empirical studies will be conducted to improve the effectiveness of the WHODATE approach upon its combination with various conventional testing approaches for database application testing.

Applications will also be designed to generate test cases automatically by inputting the embedded SQL statements.

7 References

[1] Hurson A.R., Miller L.L., Pakzad S.H., Fan C. *Functional Dependencies to Enforce Integrity Constraints in Database Machine Environments*, Computer Systems Science & Engineering, vol.6, no.2, April 1991, pp.91-101.

[2] Ramez Elmasri, Shamkant B. Navathe, *Fundamentals of Database Systems (Second Edition)*, The Benjamin/Cummings Publishing Company, Inc., 1994.

[3] Gopal R.D., Goes P.B., Garfinkel R.S., *Interval Protection of Confidential Information in a Database*, Informs Journal on Computing, vol.10, no.3, Summer 1998, pp.309-22.

[4] Abraham Silberschatz, Henry F. Korth, S. Sudarshan, *Database System Concepts (Third Edition)*, The McGraw-Hill Companies, Inc., 1997.

[5] Linthicum D.S., *Database APIs and Java*, Component Strategies, vol.1, no.2, Aug. 1998, pp.47, 49-51.

[6] Forestier J.P, *JDBC-Java Database Connectivity*, Databases Journal, no.8, May-June 1997, pp.8-11.

[7] Yang A, Linn J, Quadrato D., *Developing Integrated Web and Database Applications Using JAVA Applets and JDBC Drivers*, ACM. Sigcse Bulletin, vol.30, no.1, March 1998, pp.302-6.

[8] Robbert M.A., Maryanski F.J., *Automated Test Plan Generator for Database Application Systems*, Proceedings of the 1991 ACM SIGSMALL/PC Symposium on Small Systems, ACM Press, 1991, pp.100-6.

[9] Chan M. Y., Cheung S. C., "*Applying White Box Testing to Database Applications*", Technical Report HKUST-CS-9901, Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong.

[10] Roger S. Pressman, Software Engineering: A Practitioner's Approach (Third Edition), The McGraw-Hill Companies, Inc., 1992.

[11] Glenford J. Myers, *The Art of Software Testing*, John Wiley and Sons, 1979.

[12] Marc Roper, *Software Testing*, The McGraw-Hill Companies, Inc., 1994.

[13] Sabbatini E., Crubellati M., Siciliano S., Automating Test by Adding Formal Specification: An Experience for Database Bound Applications, Software Quality Engineering, 1997, pp.277-86.

[14] Beizer B., *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, John Wiley & Sons, Ltd, 1995.

[15] Howden W.E., *The Theory and Practice of Functional Testing*, IEEE Software, vol.2, no.5, Sept. 1985, pp.6-17.

[16] Freytag J.C., Goodman N1., *On the Translation of Relational Queries Into Iterative Programs*, ACM Transactions on Database Systems, vol.14, no.1, March 1989, pp.1-27.