Applying White Box Testing to Database Applications

Man-yee Chan* and Shing-chi Cheung*

Technical Report HKUST-CS99-01
January 1999

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

Abstract

Software testing is by far the most popular activity currently used by developers to ensure high software quality. Testing of database applications is particularly crucial 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. The database application is largely considered as a black box in the testing process. While this is useful to achieve a basic degree of quality, white box testing is required for more thorough testing of database applications. However, the semantics of the Structural Query Language (SQL) statements embedded in database applications is 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. The steps of transformations and test cases generation will be explained in detail and illustrated using an example adapted from a course registration system. This leads to the identification of additional faults involving the internal states of databases.
Applying White Box Testing to Database Applications

Man-yee Chan and Shing-chi Cheung

Department of Computer Science

Hong Kong University of Science and Technology

Clear Water Bay, Hong Kong

{mandy, scc}@cs.ust.hk

Abstract

Software testing is by far the most popular activity currently used by developers to ensure high software quality. Testing of database applications is particularly crucial 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. The database application is largely considered as a black box in the testing process. While this is useful to achieve a basic degree of quality, white box testing is required for more thorough testing of database applications. However, the semantics of the Structural Query Language (SQL) statements embedded in database applications is 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. The steps of transformations and test cases generation will be explained in detail and illustrated using an example adapted from a course registration system. This leads to the identification of additional faults involving the internal states of databases.
1 Introduction

The combination of increase in application complexity and expectation of higher reliability has placed great demands on software testing activities. Efficient and effective software testing plays a crucial role within the software development and maintenance cycle. To improve the efficiency of software testing, automatic testing instead of manual testing can be used to test software applications. Computer Aided Software Testing (CAST) tools can be employed to improve the quality and productivity of the software testing process. To enhance the effectiveness of software testing, better testing techniques should be invented to trap more faults contained in software programs. Apart from designing new testing techniques, conventional testing techniques could be combined and modified to reveal undetected faults. In this paper, we propose a new testing approach that adopts traditional white box testing techniques to generate test cases targeted to these faults in database applications.

A majority of software applications is database applications. Testing of database applications is of great importance since undetected faults in these applications may result in incorrect modification or accidental removal of crucial data. Improper change of critical data may incur loss of billions of revenues and adversely affect the fame of software companies. 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 integrity, but software costs are generally judged as too high for commercial database systems [1]. It is not practical to use these rules to maintain integrity constraints of databases in production systems. Hence, testing of database applications in both the development and production phase is important. In the development phase, testing should be carried out to reveal and remove faults to enhance the reliability of the applications. Testing should also be done in the production phase for early detection of data modification faults.

Both static and dynamic tests should be designed for database applications. Static activities such as inspection, symbolic execution, and verification can be employed to validate if both the application program and database design meet all the functional and data requirements and there is no conflict between these requirements. The database design consists of the conceptual schema, logical schema and the physical schema. The conceptual schema represents the complex relationship among data and the consistency constraints. The Entity-Relationship (ER) Model and the Object-Oriented Model are commonly used to describe the conceptual schema. The schema should be transformed to a logical schema, which is tailored to conform to the specific implementation features of a particular Database Management System (DBMS). The Hierarchical Model, the Network Model and the Relational Model are examples of logical data models. Among them, the Relational Model is the most commonly used. The physical schema should be designed with respect to the expected frequencies of queries and transactions to achieve good performance of the database applications. The physical
database design should satisfy the criteria such as response time, space utilization and transaction throughput of the systems [2]. Furthermore, the applications should restrict unauthorized access [3], enforce integrity constraints, allow concurrent access, and support recovery of data after hardware or software failures. Although various studies have been conducted to investigate testing techniques for database design, relatively few efforts have been made to explicitly address the testing of database applications. An investigation has thus been made 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 to procedures in general-purpose programming language and thereby generates test cases using conventional white box testing techniques.

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 a new testing approach that transforms SQL statements to statements in general-purpose languages and adopts conventional white box testing techniques on these transformed statements to generate test cases. Transformation rules and examples of transformations are presented. A simple example will be used to illustrate that the proposed approach is capable of revealing additional faults in database applications in section 5. Conclusion will be given in section 6 to state that traditional testing approaches should be complemented with the proposed approach in test cases generation.

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 language supported by the DBMS. Examples of this category are applications developed in the environment of DBASE or FoxPro. In these applications, queries and transactions are specified in DML and user interfaces are defined in the DBMS language. Usually, only simple textual user interfaces are supported by these applications due to the limitation of the DBMS language.

The other category involves applications that are built in both DML and general-purpose programming languages. Statements of Structured Query Language (SQL), which is a powerful declarative query language, are embedded in programs written in any general-purpose language, which is referred to as the host language, to access and update data stored in databases [4]. Embedded SQL programs are processed by special preprocessors prior to compilation to replace the embedded SQL requests with host-language declarations and procedure calls that allow run-time execution of database access. SQL statements can be embedded in DB-Library calls in C or C++ programs to access databases. They can also be embedded in function calls of Open Database Connectivity
(ODBC), which is a standard Application Programming Interface (API), in Visual Basic or Visual C++ projects. Besides, they can also be passed to methods of Java Database Connectivity (JDBC) [5][6][7] classes in Java applications for database access. In these applications, queries and modifications of data are expressed in embedded SQL statements. Other functionality like interacting with users and sending results to a graphical user interface is written in the general-purpose language such as C, C++ and Java [4]. Presently, this type of database applications is more common than the former one. Queries that cannot be expressed in SQL, and graphical user interface can be written in the host language to support more complex functionality.

Testing of database applications is different from that of structural programs. In [8], Robbert et al mentioned that the single input and output of each process is mapped into a single node path of a directed graph which is shown to be consistent in structural program testing. They stated that in database application testing, the problem is mapped into a model which is demonstrated to be syntactically correct and the database semantics are shown consistent with the desired world. A program test succeeds when a program has no errors while a database application test succeeds when the database remains in a valid state and duplicates the original environments. In other words, in addition to checking the outcome with the expected outcome, programmers or testers should also check if the database is consistent, does not violate the integrity constraints and represent the original environments in database application testing. Although it is impractical to use functional dependency rules to enforce integrity constraints in production systems, it can be employed in the development phase to ensure the consistency of the database. Though the enforcement of integrity constraints that represent rules pertaining to the organization, the accuracy of information in databases can be accomplished. 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 (TCG), Test Data Preparation (TDP), and Test Execution and Outcome Verification (TEOV).

![Figure 1: Sub-problems of database application testing](image)

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.

After test cases have been generated, test data should be constructed for each test. In database application testing, test data involve database instances in addition to user inputs. A database instance should be generated in such a way that it can be employed to execute as many test cases as possible to improve the efficiency of software testing. However, the construction of test data for each test needs careful study [8]. It is difficult to design a set of database instances that support the execution of all the generated test cases in some predefined execution orders. In [9], a Data Generation Language (DGL) has been developed to allow easy and rapid generation of any relational database. Besides, tools have been developed to support automatic generation of large quantities of test data. In [10], a commercial CAST tool has been developed to automatically create the initial test data that satisfy both application and database requirements by using black box testing and sampling techniques. It claims to generate the least number of test cases with maximum coverage based on equivalent class partitioning and boundary condition analysis. Bates et al created a set of tools that allows generation of test data in a manner which renders them amenable to subsequent independent analysis [11]. At first glance, it can solve the problem of test data generation. However, further investigation may be needed to derive the solution of this problem.

After the execution of the applications, the outcome should be verified to indicate whether the tests are successful. As mentioned, a database application test succeeds when the database remains in a valid state and duplicates the original environments. It is theoretically possible to check whether the database fulfills the aforementioned criteria and determine the expected outcome if any. However, it is too difficult to determine the correct outcome and consider if these criteria are matched when the database applications are complex and the test database instance is large. To cite an example, it is time consuming for the tester to find all students who got a Graduation Grade Average (GGA) of ten or higher in a university from her database containing millions of student records. More importantly, it seems impossible for the tester to determine whether the database is still in a valid state after the execution of the application. For example, the tester may not discover the application has mistakenly modified the students’ GGA during the execution of the above functionality. The mechanism which checks the correctness of a program on test data given the outcome produced by the program on the test data is known as an oracle [12]. A program is considered non-testable if either of the following conditions occurs: (1) there does not exist an oracle; (2) it is theoretically possible, but practically too difficult to determine the correct outcome [12]. As such, most database applications are considered to be non-testable. Typically, database applications are checked either by the tester “eyeballing” the outcome and the database to see if they “look okay” or by examining error-prone portions of the outcome in detail and inferring the correctness of all the outcome from the correctness of these portions. Weyuker suggested building a pseudo-oracle, an independently written program intended to
fulfill the same specification as the original program to test a non-testable program [12]. The two programs are run on identical data sets and the outcome is compared to determine whether the original program is correct. However, the use of pseudo-oracles is not practical for database applications since it requires a great deal of overhead. Metamorphic testing [13], which is a technique for selection of test cases when the tests are successful, could be adapted to alleviate the problem of missing oracle. In general, equivalent classes of queries could be generated to produce outcome to compare with outcome of the original queries. Queries that return subsets or supersets of the original outcome could also be generated to produce outcome for comparison. Many efforts have to be made to examine the details for adapting the technique to solve the problem.

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 above, undetected faults remaining in database applications may cause serious damage to critical production data. In this paper, we focus on the problem of test cases generation and propose a novel mechanism to improve the effectiveness of database application testing by generating additional test cases which are not covered in traditional white box testing techniques. Additional interesting faults were identified in our experiments based on a database application adapted from a course registration system.

3 Related Works

3.1 The Black Box Approach

Nowadays, most database applications are tested using only black box testing techniques [14][15] such as Equivalence Partitioning (EP), Boundary Value Analysis (BVA), and Cause-effect Graphing techniques. Test cases are derived without reference to the construction of application programs. According to the specification of the software, test cases are produced to test the functionality independently. It is typically used to check if the application conforms to its specification [16].

As an example, in [17], a relational database application was designed using the Cleanroom Software Engineering Methodology, which is a team approach to the incremental development of software under statistical quality control. Design inspection and code walkthrough were performed to check the correctness of the design and the code. Statistical certification testing were employed to test the application. Dospisil et al in [18] proposed the Multidatabase Management and Development (MMD) toolkit, which supports software designers and developers to develop multidatabase applications. The distributed database management tool of the toolkit supports functional and system testing of the application while the multi-database access simulation tool increases the efficiency of functional testing. The work in [19] deals with database environments managed by a Command Interface. In such a system, each command is considered as an independent process to be tested. Testing means
verifying that each command \((C_i)\) performs a database updating \((Sdb')\) as described in its functional specifications, also in regards to the current state of the database \((Sdb)\). For a generic command-instance \(C_i(j)\) for the command \(C_i\), \(I_j\) equals \((C_i(j), Sdb)\), \(O_j\) equals \((Sdb')\) and \(Sdb'\) equals \(F(C_i(j), Sdb)\).

One of the advantages of black box testing techniques is that test cases can be generated independent of the programs by testers 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. However, relatively few efforts have been made to test and debug the applications objectively. Another advantage is that the cost of generating test cases using black box techniques is rather low as compared to the cost of those using white box techniques. Thus, the development cost can be reduced. Besides, many black box testing techniques have already been well designed. Testers can select those which are most suitable for their database applications.

On the other hand, without examining the programs, it is not known how much of an application is being tested. Practically, functional specifications are usually specified in natural language. As mentioned in [19], using natural language to express functional specification causes problems including redundancy, inconsistency, and incompleteness. In addition, black box testing is blind to some kinds of faults. A typical example is the kind of faults related to execution sequences of functionality. A simple C++ program shown in Figure 2 illustrates the problem. The application is designed to provide the functionality \(F1\), \(F2\), and \(F3\). When the program is tested under black box techniques, the functions are tested in a randomly chosen order, say \(F1\), \(F2\) and \(F3\). During the testing phase, all the three functions work well and no errors are discovered. However, in the production phase, the functions may be executed in any orders such as the order of \(F2\), \(F1\), and \(F3\). Suppose the database is modified in \(F2\) and that \(F1\) does not work in that modified state. The program fails when \(F1\) is executed after \(F2\).
This kind of faults is common in database applications. A function of a database application is said to be dependent on other functions when the set of relations it accesses overlaps with those accessed by others. Its outcome is affected by its related functions as these functions change the internal state of the database. The function may fail in the modified state of some sequences of execution of other functionality. In database applications with graphical user interface, there can be many combinations of different execution sequences. As such, it is necessary to look for techniques beyond black box testing.

3.2 The Traditional White Box Approach

Normally, practitioners do not make use of white box testing techniques for database applications. Only when the data structures are extremely complex, white box techniques like statement testing, branch testing, condition coverage, and path testing [16] are employed to test some portions of database applications. White box testing lets testers examine the code in detail and make sure that at least a level of test coverage such as execution of every statement have been achieved [16].

Nevertheless, traditional white box techniques have their own limitations in testing database applications. Most importantly, they have not considered the testing of SQL statements embedded in application programs. 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 one test case is generated by white-box techniques to test the embedded SQL statements. For example, a test case with test data of user input “AB0001” and a database instance containing no information about the student with ID “AB0001” satisfies the requirements of statement testing, branch testing.
condition coverage, and path testing. We suspect that faults in SQL statements may not be revealed with the test cases generated by white box techniques despite a few more test cases are generated to cover the statements from line 13 to line 27.

```c
void process(DBPROCESS* dbproc) {
    RETCODE result_code;
    char sid[IDLEN+1];
    char cid[IDLEN+1], cname[NAMELEN+1];
    fprintf(IN_CH, "Please enter the student ID: ");
    fscanf(IN_CH, %s, sid);    
    dbfcmd(dbproc, "select COURSE.CID, COURSE.name");
    dbfcmd(dbproc, " from TAKE, COURSE");
    dbfcmd(dbproc, " where COURSE.CID = TAKE.CID and");
    dbfcmd(dbproc, " TAKE.SID = %s", sid);
    dbfcmd(dbproc, " order by TAKE.semester");
    dbsqlExec(dbproc);
    while ((result_code = dbresults(dbproc)) != NO_MORE_RESULTS) {
        if (result_code == SUCCEED) {
            dbbind(dbproc, 1, NTSTRINGBIND, (DBINT)0, (BYTE*)cid);
            dbbind(dbproc, 2, NTSTRINGBIND, (DBINT)0, (BYTE*)cname);
            if (dbnextrow(dbproc) == NO_MORE_ROWS)
                fprintf(OUT_CH, "Data not found. 
";    
            fprintf(OUT_CH, "Student with ID %d has taken the ", sid);
            fprintf(OUT_CH, "following courses: 
");
            fprintf(OUT_CH, "%s, %s\n", cid, cname);
            while (dbnextrow(dbproc) != NO_MORE_ROWS)
                fprintf(OUT_CH, "%s, %s\n", cid, cname);
        } else {
            fprintf(OUT_CH, "Student with ID %d has taken the ", sid);
            fprintf(OUT_CH, "following course(s): 
");
            fprintf(OUT_CH, "%s\n", cname);
        
        }    
    }
}
```

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. As indicated in Figure 4, the testing scheme involves a testing cycle composed of three phases: modular and operational testing, application testing, and simulated user testing. In the first phase, standard manual or software driven error-checking procedures are used to produce program codes which satisfy the outcome specifications. The second phase consists of the generation of a test plan that is claimed to be capable of checking the robustness of the software systems. In the last phase, independent testers follow the generated test plan to test the application.
The test procedure relies on the conceptual model, which can be 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 input of the test plan generator includes user-defined operations, types, entities and relationships of the conceptual model. The generated test plan includes the 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 5.

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 in a systematical way. With the input of the user-defined functions, test cases can be generated to test the database functions supported by each entity.
Nevertheless, Robbert et al have not provided in [8] systematic procedures for selecting test cases and determining the completion point of testing. To cite an example, it defines exhaustive testing to be “a means of testing is provided for all constructs of the conceptual model” but it does not explain the procedures of test case generation. Different testers may thus produce different sets of test cases in order to examine all constructs of the conceptual model. These different sets of test cases 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. As mentioned, failures that only happen in some particular sequences of execution may not be revealed by the generated test cases.

### 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 works as follows. Given a database application with embedded SQL statements, these SQL statements are transformed into statements in a general-purpose programming language. Conventional white box techniques are then applied to
both the transformed statements and other statements written in the host language. The mechanism of
test case generation is depicted in Figure 6.

Figure 6: The proposed approach: Transformation of SQL statements and test cases generation using
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, SQL statements are first converted into
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.
Relational algebraic expressions are composed of fundamental relational algebraic operations. These
operations include Selection, Projection, Union, Difference, Cartesian Product, and Rename [4].
Their notations and definitions are given in Table 1.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Notations</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>$\sigma_F(R)$</td>
<td>Selection of R under F is the set of tuples t in R, for which F becomes true when the column numbers in F are replaced by the corresponding components of t.</td>
</tr>
<tr>
<td>Projection</td>
<td>$\pi_{t_1,\ldots,t_k}(R)$</td>
<td>Projection of R onto $t_1, \ldots, t_k$ is the set of tuples $(\alpha_1, \ldots, \alpha_k)$ such that there is a tuple $(b_1, \ldots, b_n)$ in R and $a_j = b_j$ for $j = 1, \ldots, k$.</td>
</tr>
<tr>
<td>Union</td>
<td>$R \cup S$</td>
<td>Union of R and S is the set of tuples that are in R and S (or both).</td>
</tr>
<tr>
<td>Difference</td>
<td>$R - S$</td>
<td>Difference of R and S is the set of tuples that are in R, but not in S.</td>
</tr>
<tr>
<td>Operations</td>
<td>Notations</td>
<td>Definitions</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Cartesian Product</td>
<td>$R \times S$</td>
<td>Cartesian product of $R$ and $S$ is the set of tuples whose first $m$ components form a tuple in $R$ and whole last $n$ components form a tuple in $S$.</td>
</tr>
<tr>
<td>Rename</td>
<td>$\rho_x(R)$</td>
<td>Rename returns an additional name $x$ for the relation $R$.</td>
</tr>
</tbody>
</table>

Table 1: Fundamental relational operations of relational algebraic expressions

To cite an example of transformations between SQL statements and RAES, let us consider the partial schema of a course registration application.

```sql
COURSE(ID, Name, CreditNo)
OFFERING(CID, Semester, Schedule, Quota, Room)
```

The following SQL statement outputs the name of all courses offered in the Fall semester of 1998.

```sql
SELECT COURSE.Name
FROM COURSE, OFFERING
WHERE OFFERING.Semester = "1998F" AND OFFERING.ID = COURSE.ID
```

The equivalent relational algebraic expression to perform the above function is formulated as:

$$\pi_{\text{COURSE.Name}} \sigma_{\text{COURSE.ID}=\text{OFFERING.ID}} (\pi_{\text{Semster}=\text{"1998F."}} \text{OFFERING})$$

Note that there may exist more than one relational algebraic expression equivalent to the above SQL statement. The following relational algebraic expression is an example.

$$\pi_{\text{COURSE.Name}} \sigma_{\text{COURSE.ID}=\text{OFFERING.ID} \text{ and Semster=}\text{"1998F."}} (\pi_{\text{Semster=}\text{"1998F."}} \text{OFFERING})$$

However, the choice of the equivalent relational algebraic expression may influence the transformation of SQL statements to statements in general-purpose programming language. In later subsections, we will discuss whether the choices affect the effectiveness of the test cases generated.

### 4.2 Transformations Between Relational Algebraic Expressions and Statements in General-purpose Programming Language

A query evaluation plan (QEP), which describes the query computation, can be expressed in terms of set-oriented operations like `SCAN`, `FILTER`, `PROJECT`, and `LJOIN` [20]. 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 [20] 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 are used to independently implement the various set-oriented operators permitted in QEPs. Similarly, the fundamental relational operations in RAES are translated into a set of 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 as follows 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.

```cpp
class Relation {
public:
    Relation(); // default constructor
    virtual ~Relation(); // default destructor
    int getNoOfTuples(); // return the number of tuples
    Record& getTuple(int i); // return the ith tuple
    void insert(Record& tuple); // insert a tuple
    void remove(Record& tuple); // remove a tuple
    void swapTuple(int i, int j); // swap the ith and the jth tuple
    void operator=(Relation& r); // overload assignment operator
private:
    int noOfTuples; // no. of tuples in the relation
    Record* tuples; // the list of records
};

class Record {
public:
    Record(); // default constructor
    virtual ~Record(); // default destructor
    bool operator==(Record& r); // overload comparison operator
    Record& operator+(Record& r); // overload concatenation operator
private:
    int noOfAttr; // no. of attributes
};
```

Each fundamental algebraic operation in the RAE is transformed to a procedure that receives one or more relations and outputs one new relation. In addition to relations, some procedures may receive other sorts of parameters like the list of projection, and the functionality criteria. The definitions of procedures for the fundamental algebraic operations are as follows.

**Selection**

```cpp
Relation& Selection(bool (*predicate)(Record&), Relation& r) {
    Relation* newR = new Relation;
```
for (int i=0; i<r.getNoOfTuples(); i++) {
    if ((*predicate)(r.getTuple(i))) {
        newR->insert(r.getTuple(i));
    }
}
return *newR;

Projection

Relation& Projection(Record (*project)(Record&), Relation& r) {
    Relation* newR = new Relation;
    for (int i=0; i<r.getNoOfTuples(); i++) {
        newR->insert((*project)(r.getTuple(i)));
    }
    return *newR;
}

Union

Relation& Union(Relation& r, Relation& s) {
    Relation* newR = new Relation;
    bool identity;
    for (int i=0; i<r.getNoOfTuples(); i++) {
        newR->insert(r.getTuple(i));
    }
    for (int i=0; i<s.getNoOfTuples(); i++) {
        identity = false;
        for (int j=0; j<r.getNoOfTuples(); j++) {
            if (s.getTuple(i) == r.getTuple(j)) {
                identity = true;
                break;
            }
        }
        if (!identity) {
            newR->insert(s.getTuple(i));
        }
    }
    return *newR;
}

Difference

Relation& Difference(Relation& r, Relation& s) {
Relation* newR = new Relation;
bool identity;
for (int i = 0; i < r.getNoOfTuples(); i++) {
    identity = false;
    for (int j = 0; j < s.getNoOfTuples(); j++) {
        if (r.getTuple(i) == s.getTuple(j)) {
            identity = true;
            break;
        }
    }
    if (!identity) {
        newR->insert(r.getTuple(i));
    }
}
return *newR;
}

**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;
}

**Rename**

Relation& Rename(Relation& r) {
    Relation* newR = new Relation;
    for (int i = 0; i < r.getNoOfTuples(); i++) {
        newR->insert(r.getTuple(i));
    }
    return *newR;
}

Note that a *predicate* function is required to indicate whether the inputted record fulfills the predicate while a *project* function is needed to output a new record that only consists of the projected attributes. In addition, the symbol “==” refers to the comparison operator of a record and the symbol “+” shows the concatenation operator of a record.
The first step of the transformation process 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 relational algebraic operations and iterative procedures continue in a similar manner until all the operations of the relational algebraic expression have been replaced. To illustrate the transformation process, the following relational algebraic expression is considered.

\[ \pi_{\text{COURSE.Name}} \sigma_{\text{COURSE.ID}=\text{OFFERING.ID} \text{ and } \text{Semster}={1998}} \left( \text{COURSE} \times \text{OFFERING} \right) \]

The following procedure calls and the statements in the procedures called in the statement are the transformation of the above relational algebraic expression.

```plaintext
Projection(*project), Selection(*predicate),
Cartesian_Product(COURSE, OFFERING));
```

### 4.3 Generation of White Box Test Cases

After relational algebraic expressions have been transformed to procedures in general-purpose programming language, different white box testing techniques could be used to produce test cases from the aggregation of the transformed procedures and the host statements. The choice of white box 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 testing methods. 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 7.
The definition of completeness of the testing depends on the white box techniques employed in test cases generation. As an example, 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 white box techniques used to generate test cases. To give an example, the level of test coverage when using path testing will be much higher than that when using branch testing. The amount of test effort required is also flexible. The number of test cases generated is larger and much test effort has to be paid when higher level of test coverage is required for critical database applications. Fewer test cases are produced and less test effort has to be given when the level of test coverage is required to be medium for simple database applications. Practitioners should decide what white box techniques they 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 multiple 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 the WHODATE approach is \( \prod_{i=1}^{n} b_i \) times of that produced in the conventional white box approach. \( b_i \) 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 8. The final set of test cases is the nine resultant execution paths (1A, 1B, …, 3B, 3C).

As another simple example, when there are 5 SQL statements in an application and each statement generates 3 execution paths, the number of test cases generated is \( 243 \) times of that produced by the conventional white box approach. To reduce the number of test cases generated while maintaining the effectiveness of the testing approach, the conceptual model of the database application could be employed to combine only those execution paths that are relevant to one another. Say, path 1A is formed if the sets of relations accessed in path 1 and path A are not disjoint. Further study should be made to utilize the conceptual model to produce more effective test cases.

Apparently, the proposed testing approach aims at discovering faults of the SQL statements embedded in a database application. The transformation procedures map the semantics of embedded SQL statements to that of general-purpose programming languages. This complements conventional techniques of white box testing, which does not specifically take into account the semantics of
embedded SQL statements when test cases are generated. As a result, the test cases thus generated are useful to reveal faults that only occur in some particular states of the database.

In many cases, the functional specifications of database applications are vague and informal, the test cases generated from these specifications by testers may not be capable of revealing 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 to reveal faults in the applications. The variations that may be caused in the transformations between SQL statements and RAEs do not influence the effectiveness of the testing greatly as the flow graphs of all the procedures in the RAEs are employed in test cases generation. Most 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 functionality are being executed in some particular sequences.

5 A Case Study

As mentioned, the inclusion of the semantics of SQL statements in test case generation is useful in identifying additional faults related to the states of the database. Let us illustrate this using a programming assignment submitted by our undergraduate students in a database course. Experiments were also conducted to examine the effectiveness of WHODATE upon its combination with various white box testing techniques. The programming 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 of the functions 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. The codes of the function have been shown in Figure 3. For the ease of explanation, let us represent the procedures transformed from embedded SQL statements using flow graphs. To examine the effectiveness of WHODATE upon its combination with different white box techniques, experiments are conducted based on three popular white box testing techniques, namely statement testing, branch testing and path testing. In the path testing, we generated test cases to execute all equivalent classes of paths rather than all paths of the function. We compared the sets of test cases generated with those produced by applying the three testing techniques conventionally on all statements without transformation of SQL statements. The comparison shows that our approach of including the semantics of SQL statements in test case generation helps uncover additional faults.

Before test cases can be generated, the SQL statement in the function being tested should be transformed to procedures in general-purpose programming language. As mentioned in section 4, the
first step of the transformation is to transform the SQL statement to a relational algebraic expression. The SQL statement and its corresponding relational algebraic expression are as follows.

```
SELECT COURSE.CID, COURSE.Name  
FROM TAKE, COURSE  
WHERE TAKE.SID = <ID> AND COURSE.CID = TAKE.CID  
ORDER BY TAKE.SEMESTER
```

The symbol $S$ in the above relational algebraic expression refers to the operation $\text{Sorting}$. It outputs the inputted relation in the ascending order of the specified attribute. It is an operation designed to support the function performed by the SQL phrase $\text{ORDER BY}$. Similar to the procedures of Selection and Projection, a comparison function should be implemented and passed to the procedure of Sorting to define the comparison criterion. The definition of the procedure is shown as follows.

**Sorting**

```
Relation& Sorting(int (*compare)(Record&, Record&), Relation& r)  
{  
    Relation* newR = new Relation;  
    for (int i=0; i<newR->getNoOfTuples()-1; i++)  
    {  
        for (int j=0; j<newR->getNoOfTuples()-i-1; j++)  
        {  
            if ((*compare)(newR->getTuple(j), newR->getTuple(j+1)) > 0)  
            {  
                newR->swapTuple(j, j+1);  
            }  
        }  
    }  
    return *newR;  
}
```

Next, the RAE should be transformed to procedures in general-purpose programming language. The transformed statements of the RAE consist of the following procedure calls and the statements in these procedures.

```
Projection(*project), Sorting(*comparison),  
Selection(*predicate2), Cartesian_Product(Selection(*predicate1),  
TAKE), COURSE))
```

Then, execution paths and the corresponding test cases are generated by applying the three white box techniques on the transformed statements. Note that the input relations in the arguments of different procedures may be different. The corresponding relations need to be considered when test cases are generated. First, test cases are generated from the statements of the procedure Selection. The statements within the procedure Selection when the parameter relation $r$ is replaced by the input relation $TAKE$ are:
1 Relation* newR = new Relation;
2 for (int i=0; i<TAKE.getNoOfTuples(); i++) {
3   if ((*predicate)(TAKE.getTuple(i))) {
4     newR->insert(TAKE.getTuple(i));
5   }
6 }
7 return *newR;

The flow graph of the procedure is shown in Figure 9. Each node in the graph represents a block of codes while an edge represents the transfer of control.

![Flow graph of Selection((*predicate1), TAKE)](image)

Figure 9: Flow graph of Selection((*predicate1), TAKE)

Table 2 shows the test cases generated by applying statement testing, branch testing and path testing, respectively on flow graph.

<table>
<thead>
<tr>
<th>Statement Testing</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcefgh</td>
<td>The student has taken one course.</td>
</tr>
<tr>
<td>Branch Testing</td>
<td></td>
</tr>
<tr>
<td>abcefgh</td>
<td>The student has taken one course.</td>
</tr>
<tr>
<td>abdfgh</td>
<td>The student has taken no course.</td>
</tr>
<tr>
<td>Path Testing</td>
<td></td>
</tr>
<tr>
<td>agh</td>
<td>No student has taken any courses (TAKE is empty).</td>
</tr>
<tr>
<td>abdfgh</td>
<td>The student has taken no course (TAKE has 1 tuple).</td>
</tr>
<tr>
<td>abcefgh</td>
<td>The student has taken one course (TAKE has 1 tuple).</td>
</tr>
<tr>
<td>a(bdf)^n gh</td>
<td>The student has taken no course (TAKE has n tuples).</td>
</tr>
<tr>
<td>ab^n(ce)d^{n-k}f^{n} gh</td>
<td>The student has taken k courses (TAKE has n tuples).</td>
</tr>
<tr>
<td>a(bcelf)^n gh</td>
<td>The student has taken n courses (TAKE has n tuples).</td>
</tr>
</tbody>
</table>

Table 2: Test cases generated from statements in Selection((*predicate1), TAKE)
Second, test cases are produced with respect to the procedure Cartesian_Product. The following enlists the transformed statements of the procedure when \( r \) is replaced by the output (\( TEMP \)) of \( Selection(*\text{predicate}, \text{TAKE}) \) and \( s \) is replaced by the relation \( COURSE \).

1. \( \text{Relation* newR = new Relation;} \)
2. \( \text{for (int i=0; i<TEMP.getNoOfTuples(); i++)} \) {
3. \( \quad \text{for (int j=0; j<COURSE.getNoOfTuples(); j++)} \) {
4. \( \quad \quad \text{newR->insert(TEMP.getTuple(i) + COURSE.getTuple(i));} \)
5. \( \quad } \)
6. \( \) } \)
7. \( \text{return *newR;} \)

The corresponding flow graph of the above statements is shown in Figure 10.

![Figure 10: Flow graph of Cartesian_Product(TEMP, COURSE)](image)

Different categories of combinations of the number of tuples contained in the relation \( TEMP \) and \( COURSE \) are considered in path testing. Some of the classes of combinations result in the same paths while some of them violate the integrity constraints of the database. Thus, these test cases are ignored and rejected. Table 3 shows the test cases produced by the three white box techniques.

<table>
<thead>
<tr>
<th>Statement Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcedfgh</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Branch Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcefgfgh</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Path Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>agh</td>
</tr>
<tr>
<td>abcefgfgh</td>
</tr>
</tbody>
</table>
Table 3: Test cases generated from statements in Cartesian_Product(TEMP, COURSE)

Similarly, test cases can be generated by considering the transformed statements of the remaining procedures Sorting and Projection. Note that the characteristics of both the intermediate relations and relations in the database are used in generating test cases. We can observe that some procedures produce more test cases while some simpler procedures produce fewer test cases.

Next, we generate test cases from the original program shown in Figure 3 from the flow graph depicted in Figure 11. The test cases generated by applying the three white box techniques conventionally on the function are as shown in Table 4.

<table>
<thead>
<tr>
<th>Statement Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>xyabcefhmno</td>
</tr>
<tr>
<td>xyabcgijklmno</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Branch Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>xyabcefhmno</td>
</tr>
<tr>
<td>xyabcgijklmno</td>
</tr>
</tbody>
</table>

Figure 11: Flow graph of the function under investigation
Path Testing

<table>
<thead>
<tr>
<th>xyabcefhmno</th>
<th>The student has taken no course.</th>
</tr>
</thead>
<tbody>
<tr>
<td>xyabcegilmno</td>
<td>The student has taken one course.</td>
</tr>
<tr>
<td>xyabcegijklmno</td>
<td>The student has taken two courses.</td>
</tr>
<tr>
<td>xyabcegijklmno</td>
<td>The student has taken n courses.</td>
</tr>
</tbody>
</table>

Table 4: Test cases generated by conventional white box testing techniques

After the execution paths of procedures of each SQL statement and statements in the original program have been enumerated, these execution paths are combined. Execution paths generated from the procedures of the SQL statement in line 7-11 are inserted between the edge y and a in the paths shown in Table 4.

More test cases can be generated by the proposed testing approach compared with the conventional white box testing approach as the semantics of SQL statements are considered in the generation of test cases. Consequently, faults relevant to the internal states of database can be revealed. Traditional testing approaches usually overlook this type of faults. For example, the programmer may assume that each student is allowed to take a course only after he or she has 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 he or she may be permitted to take a course in the current semester and retake its prerequisite course in the next semester. In this case, the application cannot list the courses taken by the student in the way that each course appears after its prerequisite courses.

As an example, the course 102 is the prerequisite course of the courses 221, 231 and 251. The student with student ID DS9001 has failed the course 102 in the Fall semester of 1997. However, he or she is allowed to take the courses 221, 231 and 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 courses 221, 231 and 251 appear before their prerequisite course 102.

Please enter the Student ID: DS9001
The student with ID DS9001 has taken the following course:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Computer Applications</td>
</tr>
<tr>
<td>102</td>
<td>C Programming</td>
</tr>
<tr>
<td>104</td>
<td>C++ Programming</td>
</tr>
<tr>
<td>111</td>
<td>Software Tools</td>
</tr>
<tr>
<td>221</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>231</td>
<td>Database Systems</td>
</tr>
</tbody>
</table>
Table 4 indicates that faults generated from the original program using conventional white box testing techniques cannot reveal this fault. However, the last execution path in Table 3 requires the final set of test cases to include the test case where the student has taken $n_1$ courses and there are totally $n_2$ courses. If $n_1$ is greater than $n_2$, the students may have retaken some of the courses. This fault can be revealed.

In addition, the test case can also reveal the repeated output of the same course even when the student retakes a course in the same semester. These faults may also not be found by test cases produced by black box testing techniques since the functional specification may be vague and does not specify these situations. As a summary, the proposed approach complements conventional white box techniques in database application testing.

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 techniques are employed to generate test cases to test database applications and the SQL statements embedded in the application programs are not specifically considered in white box testing techniques. 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 of these approaches have cons in testing database applications.

In this paper, we have proposed a new testing approach that transforms SQL statements to procedures in general-purpose programming language and applies conventional white box testing techniques on both these transformed procedures and the host statements to generate test cases. The process of transformations and methods of test cases generation have been presented. In addition, a case study has been given to illustrate the test cases generation and 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. In addition, the test cases generated helps reveal faults related to the internal states of databases in database applications. As a conclusion, black box techniques should be employed to verify that the database application fulfills the specified functional requirements while the proposed technique should be used to reveal faults relevant to the internal states of databases.

More empirical studies have to be conducted to examine the effectiveness of the WHODATE approach upon its combination with various conventional testing approaches for database application
testing. In addition, applications can be designed to generate test cases automatically by inputting the embedded SQL statements.

7 References


