Using an SQL Coverage Measurement for Testing Database Applications

María José Suárez-Cabal
University of Oviedo
Department of Computer Science
Campus de Viesques, Gijón, Spain
(+34) 985 18 2506
cabal@uniovi.es

Javier Tuya
University of Oviedo
Department of Computer Science
Campus de Viesques, Gijón, Spain
(+34) 985 18 2049
tuya@uniovi.es

ABSTRACT

Many software applications have a component based on database management systems in which information is generally handled through SQL queries embedded in the application code. When automation of software testing is mentioned in the research, this is normally associated with programs written in imperative and structured languages. However, the problem of automated software testing applied to programs that manage databases using SQL is still an open issue. This paper presents a measurement of the coverage of SQL queries and the tool that automates it. We also show how database test data may be revised and changed using this measurement by means of completing or deleting information to achieve the highest possible value of coverage of queries that have access to the database.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *Testing tools, coverage testing*

General Terms

Experimentation, Languages, Measurement, Verification.

Keywords

verification and validation, software testing, database testing, SQL testing, statement coverage.

1. INTRODUCTION

Testing is one of the most expensive processes in the development and maintenance of software products, with over 30% of resources being committed to this end [7]. A recent NIST study [11] estimates that the costs of software faults total 59.5 billion dollars, more than a third of which are incurred during development. It is estimated that these costs could be reduced by half should an adequate infrastructure for testing be available. As the majority of defects are introduced in the initial phases of programming, it is essential to include improvements in the software testing process that can be used by programmers in these phases [6].

On the other hand, it is common for software applications written in an imperative language to have access to the database through SQL statements embedded in the code. These queries are part of the application's business logic. Because of this, it is necessary to have conducted suitable testing in the same way of the rest of the code. The tests should cover all the query situations and avoid producing undesired results so as to obtain their maximum possible coverage. Test design is a difficult task, mainly due to the information contained in the databases and to the SQL code itself.

The main aims of the present paper are to:

- Define a measurement of coverage of SQL SELECT queries in relation to a database loaded with test data that can be used as an adequacy criterion to carry out the testing of applications with access to databases.
- Present an algorithm that automates the calculation of coverage in order to help the software tester.
- Extract a subset of database information that allows the obtainment of at least the same result as the original set for the established adequacy criterion.
- Guide the expert with respect to how the database tuples might be changed to increase the obtained coverage value, if possible.

The remainder of this paper is organized as follows. Section 2 examines the relationship between software testing and database applications. Firstly, some of the published work related to this topic is briefly described in Subsection 2.1. Then, some characteristics of software testing applied to applications that access databases through SQL statements are discussed in more detail in Subsection 2.2. After that, Subsection 2.3 presents an example of detection of faults in a SELECT query. Following this, a method for measuring the coverage of SQL SELECT queries is established in Section 3,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSOFT'04/FSE-12, Oct. 31–Nov. 6, 2004, Newport Beach, CA, USA.
Copyright 2004 ACM 1-58113-855-5/04/0010…$5.00.

along with how this is calculated and supported by a tool. Subsequently, the results of the algorithm run on the SELECT queries with a real database are described in Section 4. To finish, Section 5 presents the conclusions we have reached, together with future lines of research and development.

2. SOFTWARE TESTING AND DATABASES

2.1 Related work

A number of studies have been found in the literature related to the topic of software testing for databases and applications that access these through SQL statements embedded in the code. Some of these articles are described below.

One of the studies [9] was carried out by the Microsoft research team. Valid SQL statements were randomly generated and run on several systems that contained identical databases. The aim was to evaluate database management systems with the SQL statements so-produced and compare the outputs obtained. In [4], a set of valid and invalid data was generated from a database structure and its constraints at field-level in order to automatically load the initial database However, the SQL statements that run with the generated data were not considered, nor was the adequacy criterion used for the generation of tests indicated.

In a further study [3], a design of a tool was presented whose function is to facilitate testing of applications with testing databases. The test data generation is basically based on specifications and boundary values. Input data were generated to fill a database, satisfying integrity constraints and considering SQL queries of application. Once the output had been obtained, this was compared with the expected results and the final state of the database was checked.

Another study [2] used white box testing on applications with access to databases. The embedded SQL code was translated to the same application imperative code, allowing conventional techniques to then be applied. Other studies showed how to generate instances for databases from the semantics of the SQL statements of a program [12]. Both were based on SQL code, but they considered neither database schema nor database integrity constraints.

Adequacy criteria were defined to ensure the quality of the tests designed manually for database applications in [8]. These criteria made use of controlflow and dataflow techniques associated with relational database entities. In [5], dataflow and controlflow analysis and the dependences between components of a database application were used to determine the components that should be tested when any change was produced and to minimize the set of test cases in regression testing, but its aim was not the design of test cases.

In the commercial area, there are a small number of generators of database instances [3,4], e.g. TestByte 3, TestBase or DataTect, that generate random information depending on the type of fields. More sophisticated tools permit the user to define sets of data for names, cities, value ranges, permitted and forbidden values. The problem with these generators is that the user needs to know the database structure. There are other tools, such as SQLUnit [10], that facilitate testing of stored procedures and SQL statements. Calls to stored procedures, variables, constraints and expected results of queries can be specified through XML.

As can be seen, there are not many research papers related to this topic; perhaps as a result of the problems outlined below. What is more, the approach of each of the authors is quite different.

2.2 Characteristics of software testing with databases

The process of software testing of applications with access to databases through embedded SQL queries entails several problems that make this testing difficult for the following possible reasons:

- The first task is to design the initial instances for loading the test database. The selection of this information is one of the most important steps to obtain a good set of unit test cases, as it will be the input to any SQL statement. It is necessary to decide what data are relevant, and which and how many are required. Moreover, combinations between tuples must be taken into account to cover all the SQL query situations. If there is a low number of instances, the costs of loading are much smaller, and finding and resolving possible faults will be much easier.
- Another point to consider in the design of the test database is that applications do not have only one statement. Therefore, the data should be useful for the greatest possible number of statements, as loading a test databases with different information for each query would have a very high cost.
- The information contained in a database will be the input for any SQL statement, but this information is not static; it will be modified during the running of queries. Consequently, when designing a test database, it is necessary to consider the order in which queries are executed and whether these will modify the data that will be the input to others.
- As in imperative languages, SQL statements may be parameterized by variables and constants. When designing the testing plan, these inputs must also be considered, and test data provided for them.
- Another of the problems to overcome lies in judging the adequacy of the unit test data generated: whether this really covers all the possible situations and whether the output obtained through the application of the plan fulfils the requirements for which the software was designed; in this case, the SQL statements.

2.3 Detecting faults in SELECT queries

One of the problems encountered when performing tests of SQL statements is that of estimating the adequacy of the test cases. This means that, given an SQL statement and data from the database, is it possible to know whether all the possibilities of the query are covered?

Let us take a simple example: a small database with suppliers and orders. The query specification is to obtain a list of orders including their suppliers. Figure 1 shows the Entity-Relation diagram, the data in the database, the SELECT query for the specification given, and the results of the query.

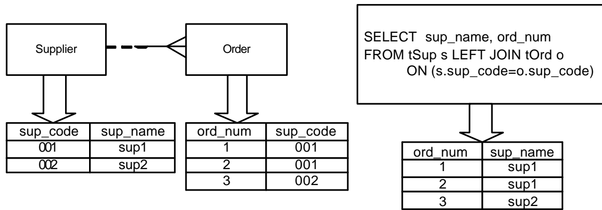

Figure 1. Example with E-R diagram, data and SELECT query.

At first glance, both the database and the query look correct. When the application is run in production, suppliers without orders are possibly included; so will the information returned by the query continue to be correct? The E-R diagram determines that there may be suppliers with no orders, but this situation is not represented in the tuples of the test database, because all suppliers have at least one order. Therefore, it is impossible to know whether the query always returns the information required in the specification: orders with their suppliers, or whether it may produce incorrect outputs: in this case, including suppliers that do not have orders. Hence, we can affirm that an adequate SELECT coverage is not obtained with these test data.

3. MEASUREMENT OF COVERAGE

The approach proposed in this paper is to establish a way of measuring the coverage of an SQL query based on the coverage concept whereby the conditions take into account the true and false values during the explorations of their different combinations [13].

Given the variety of SQL statements that can be found in an application, we have restricted these to a subset of SELECT queries specified in SQL3 [1], according to the grammar in BNF notation shown in Figure 2, in order to first achieve testing with simple SQL queries, to subsequently extend the analysis to other, more complex queries.

3.1 Coverage tree and evaluation of conditions

The solution for the automated search of SQL query situations covered with the data stored in the database is to evaluate the conditions of SELECT queries that are in the FROM clause, when they include JOIN, and in the WHERE clause. Moreover, the null values of fields will be verified at the same time as the conditions are evaluated.

A tree structure, called *coverage tree*, is created prior to coverage evaluation, in which each level represents a condition of the query beginning with the conditions of the JOIN clause, if it exists, and then with those of the WHERE clause, in the same order in which they are found in the query. Each node of the tree will store:

- Whether the condition is true for values of the fields; represented in the coverage tree as *T*.
- Whether the condition is false for values of the fields; represented in the tree as *Fl* and *Fr*. Note that, in this case, it is necessary to consider a different treatment for the cases in which the condition is evaluated from left to right and from right to left, as explained below.
- Whether there are null values in condition fields in the database; this information will then be included in the coverage tree as *Nl*, *Nr* and *Nb*. Note the different treatment, too.

```
<select> ::= SELECT <select list>
        <from clause>[<where clause>]
<select list> ::= '*'
    |<column name>[ { ','<column name>} ]
<from clause> ::= FROM <table reference>
        [ { ',' <table reference> } ]
<table reference> ::= <table name>
        [ [ AS ] <correlation name> ]
    | <table reference> [ <join type> ]
        JOIN <table reference>
        ON <search condition>
<join type> ::= INNER
    | <outer join type> [ OUTER ]
<outer join type> ::= LEFT | RIGHT
<where clause> ::= WHERE <search condition>
<search condition> ::= <boolean term>
    | <search condition> OR <boolean term>
<boolean term> ::= <boolean factor>
    | <boolean term> AND <boolean factor>
        <search condition>
<boolean factor> ::= [ NOT ]<boolean primary>
<boolean primary> ::=  <expression>
    | ( <search condition> )
<expression> ::=  <ope1> <op> <ope2>
<op1> ::= <column reference>
<op2> ::= <column reference>
    | <null specification>
    | <literal>
<op> ::= '=' | '!=' | '<' | '>' | '<=' | '>='
<column reference> ::= <column name>
    | <table name> '.' <column name>
    | <correlation name> '.' <column name>
```

Figure 2. Simplified BNF grammar of SELECT query.

Conditions are not evaluated between a single pair of values, but between sets of values, since the information in each field corresponds to a column from a table and several rows in the database. Therefore, during the evaluation of a condition, each value in the first field must be compared with each one in the second field, and each value in the second field with each one in the first, as shown in Figure 3.

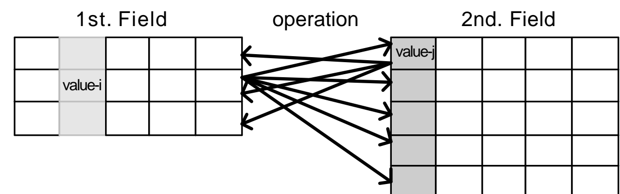

Figure 3. Operation between values of two fields.

The performed evaluation is represented in Figure 4 and is explained below:

- A condition will be true if it is verified for a pair of values from the fields to compare. It is the same result if the condition is evaluated from left to right or from right to left.
- A condition will be false from left to right, *Fl*, if none of the values from the second field verifies the condition with a value from the first field. While there are values from the second field for comparing and the condition remains false, the evaluation is considered temporarily NOT true, because it is not true, although it is not yet known whether it is false.
- A condition will be false from right to left, *Fr*, if none of the values from the first field verifies the condition with a value from the second. As in the previous case, while there are values from the first field for comparing and the condition remains false, the evaluation is considered temporarily NOT true.
- A condition will have null values when a value from the first field is null, *Nl*, when a value from the second field is null, *Nr*, or when both values, from the first and second fields, are null, *Nb*.

$$x \in \text{tuples}(\text{table}_i),\ y \in \text{tuples}(\text{table}_j)$$
$$OP \in \{=, !=, <, >, <=, >=\}$$

$$\exists x, \exists y \,/\, \text{field}_i(x)\ OP\ \text{field}_j(y) \Rightarrow T\ (\text{TRUE})$$
$$\exists x, \forall y \,/\, \neg(\text{field}_i(x)\ OP\ \text{field}_j(y)) \Rightarrow Fl\ (\text{FALSE from left to right})$$
$$\forall x, \exists y \,/\, \neg(\text{field}_i(x)\ OP\ \text{field}_j(y)) \Rightarrow Fr\ (\text{FALSE from right to left})$$

$$\exists x \,/\, \text{field}_i(x) = \text{NULL} \Rightarrow Nl\ (\text{First field NULL})$$
$$\exists y \,/\, \text{field}_j(y) = \text{NULL} \Rightarrow Nr\ (\text{Second field NULL})$$
$$\exists x, \exists y \,/\, \text{field}_i(x) = \text{NULL} \wedge \text{field}_j(y) = \text{NULL} \Rightarrow Nb\ (\text{Both NULL})$$

Figure 4. Evaluation of a condition.

Figure 5 shows the coverage tree corresponding to the example indicated in Figure 1 (this simple query has only one node representing the join). As can be seen, the node has six elements and only the *T* element is evaluated (represented by Y), because there is a situation in which both terms are true. However the condition is neither false from left to right nor false from right to left and there are no null values in the fields, represented by *N*.

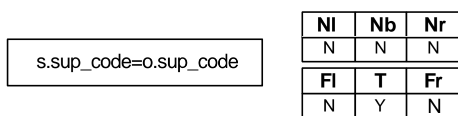

Figure 5. Coverage tree of the simple example.

3.2 Evaluation of the coverage tree and calculation of coverage

The complete evaluation of the query is carried out by crossing over the tuples of the tables that participate in the conditions at each level of the coverage tree. The evaluation finishes when the entire tree has been covered, i.e. 100% coverage has been covered, or when there are no more values for comparing.

For each particular node, the condition is evaluated for a tuple from the first field and another from the second, and:

- If the result is true, these tuples are fixed in order to evaluate the conditions of the lower levels of the tree via the *T* branch.
- If the result is false from left to right, only the tuple from the first field is fixed and, if it is false from right to left, the tuple from the second field is fixed, in order to evaluate the lower levels of the tree, via the branch at which the condition is false, *Fl* or *Fr* respectively.

It is important to fix the tuples, since the same tables, or even the same fields, could appear again at lower levels of the tree, and it is necessary to keep the values of a tuple for the evaluation of all the conditions.

After evaluating the coverage tree, the measurement of coverage may be established taking into account the conditions of the SELECT query. Two different coverage measures are established and automatically calculated:

- Theoretical coverage: which takes into account every possible situation at every node.
- Schema coverage: which takes into account the database schema constraints by excluding the impossible situations due to these constraints.

The percentage of theoretical coverage is calculated using the formula in Figure 6, in accordance with the total number of combinations of values in the conditions and the number of combinations found in the evaluation (v). The total number of combinations will be calculated as a function of the number of conditions of the query (n), the number of condition values in each node (p) and the number of child-nodes of each node (s).

$$\% \text{ coverage} = \frac{v*(s-1)}{p*(s^n-1)}*100$$

where:

v: number of cases (elements of a node) that it has been possible to verify (those marked with Y).

s: number of child-nodes that a node can have.

p: number of possible values that a condition can adopt once it is evaluated, which in the coverage measurement presented here will have six values (*Nl*, *Nr*, *Nb*, *T*, *Fl*, *Fr*).

n: number of levels of the coverage tree; i.e. the number of conditions in the query.

Figure 6. Calculation of theoretical coverage.

Figure 7 shows how this formula is applied to the simple example of Figure 1, whose coverage tree is represented above.

$$\% \text{ coverage} = \frac{1*(3-1)}{6*(3^1-1)}*100 = 16.67\%$$

where:

v=1: cases verified

s=3: each node has 3 child-nodes

p=6: values (*Nl*, *Nr*, *Nb*, *T*, *Fl*, *Fr*).

n=1: levels of the coverage tree.

Figure 7. Theoretical coverage for the simple example.

However, it is not usually possible to reach 100% theoretical coverage, because of forbidden null values or referential integrity constraints. In these cases, we use schema coverage: the maximum possible value of coverage is calculated keeping in mind the elements of each node that are possible to evaluate.

In the example in Figure 1, due to referential integrity, an order always has a supplier, therefore elements *Fr*, *Nb*, and *Nr* will never be verified, being represented in the coverage tree in Figure 8 by the value *X*. There are three possible situations and the only one covered is suppliers with orders. Thus, suppliers without orders (null value) and with non-existent orders should be incorporated into the test database.

s.sup_code=o.sup_code

Nl	Nb	Nr
N	x	x

Fl	T	Fr
N	Y	x

$$\% \text{ coverage} = \frac{1}{3}*100 = 33.33\%$$

Figure 8. Schema coverage for the simple example.

3.3 The tool

Figure 9 presents a schema of the tool developed for measuring the coverage of a SELECT query.

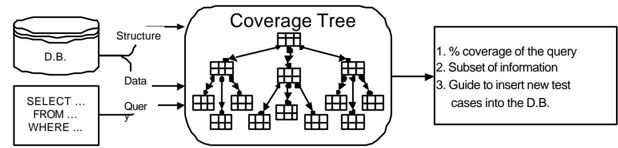

Figure 9. Inputs and outputs of the program.

As for the inputs, these will be:

- Conditions of the SELECT query. The coverage tree will be formed on the basis of these.
- Database structure: tables and columns that appear in the query.
- Data or tuples from the tables: these will be the values used for the evaluation of the conditions.

The outputs automatically obtained by the process are:

- After executing the program, the percentage of coverage of the SELECT query can be determined using the coverage tree, achieving 100% coverage if all possible situations have been verified at any time.
- During the evaluation of the coverage tree, a trace of those tuples that give new values for nodes is generated. By revising this information, a subset of tuples can be obtained that supply at least the same coverage as the original data, and that can drastically reduce the size of the test database.
- Unevaluated nodes are highlighted taking into consideration the coverage tree. By observing their conditions, their parent information, the database structure and the tuples, the expert can be guided in finding the information missing from the test database to cover all possible cases.

4. CASE STUDY

The application of the above SQL coverage measurement tool is described below for a case study with a real database and an SQL query.

The database used for testing was supplied by a company in the steel industry as the loading database of an application involved in a research project between the company and the University of Oviedo

4.1 The system

The company's department is responsible for managing the lamination rolls used in the rolling mills for the manufacture of steel sheets. The work of the department involves preparing the rolls, sending them to mills for their assembly and picking up worn, damaged, or broken-down rolls, as well as trying to resolve the problems presented for returning them to use. Each roll always

works in the same rolling mill, where the rolls are arranged in boxes, so that a mill may consist of one or more boxes, and if the mill is rolling, there are several rolls in the boxes, as can be seen in Figure 10. After a certain period of usage in the boxes, the rolls become worn and must be repaired; they are hence removed from the boxes and replaced by others.

The department manages thousands of rolls, as well as dozens of boxes and mills. It is thus necessary to maintain this information in a database and to efficiently manage this via a software application. As regards the database used for testing, note should be taken of the large amount of information: it has about a thousand rolls and twenty boxes and mills.

Several SELECT queries that contain distinct tables of the system have been analyzed with the tool, and several faults in the queries and incomplete test cases have been found. Table 1 shows the number of tables and conditions for each query and the percentages of theoretical and schema coverage.

Table 1. Queries analyzed

Query	Tables	Cond.	%TCov	%SCov
1	1	1	33.33%	66.67%
2	1	2	25.00%	37.50%
3	1	3	20.51%	30.77%
4	2	1	50.00%	50.00%
5	2	2	20.83%	20.83%
6	2	2	25.00%	40.00%
7	2	2	29.16%	58.33%
8	3	3	11.54%	11.84%
9	3	3	11.54%	17.65%
10	3	3	21.79%	28.33%
11	3	4	7.92%	8.19%
12	3	3	19.23%	46.87%

Below, we detail the use of coverage for the query numbered 12. This query is large enough to illustrate the tool's performance, while it is small enough to allow us to understand the results and resolve the faults.

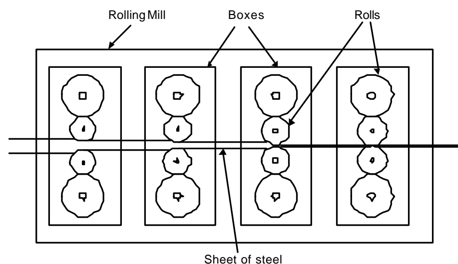

Figure 10. Schema of a rolling mill with boxes and rolls.

4.2 Data model, database design and query specification

Figure 11 shows the E-R model in which the information on the rolling mills, boxes and rolls is related. Each mill may be made up of several boxes, but there may be mills (that have been recently installed) that do not have any associated box. On entering the department, the rolls are assigned to a mill according to their physical properties and this assignment is fixed during the entire life of each roll. However, rolls are not always associated with the same box, in fact they only have a box when they are working in a rolling mill.

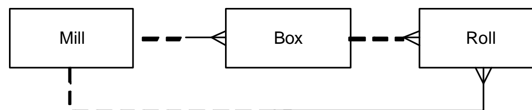

Figure 11. E-R model.

The tables corresponding to the entities of the E-R model described above and their primary keys (PK) are shown in Figure 12. Thus, the field "mill_type" (mill type) of the tables "box" and "roll" should be a foreign key of the identically named field in the table "mill". Moreover, as it is not mandatory for rolls to stay in a box, the field "box_code" (box code) of the table "roll" can be null, but the mill type of roll will always be not null.

mill	box	roll
mill_type (PK)	box_code (PK) mill_type (not null) ...	roll_num (PK) mill_type (not null) box_code ...

Figure 12. Tables, primary keys and possible null fields.

For the case study, the query numbered 12 has been chosen. It obtains information about all mills and their respective boxes, if any, and the rolls that are working in the mills at that moment. Its code is presented in Figure 13.

```
SELECT *
FROM (mill LEFT JOIN box ON
            mill.mill_type=box.mill_type)
        LEFT JOIN roll ON
            (box.mill_type=roll.mill_type) AND
            (box.box_code=roll.box_code)
```

Figure 13. SELECT query for mills, boxes and rolls.

Since mills without boxes may exist (when the installations are recent) and also mills and boxes without rolls (for example if the mill is under maintenance and it is not laminating at the moment of query), it seems adequate to use two "LEFT JOIN" clauses: one for the tables "mill" and "box", so that all mills are obtained with or without boxes; and another for "box" and "roll", with the goal of obtaining all boxes, with or without rolls.

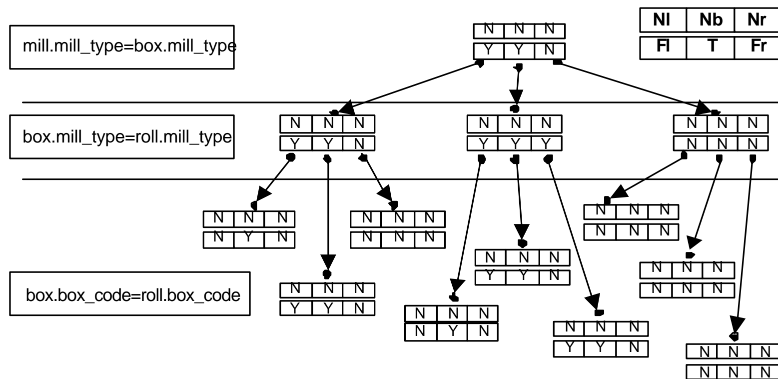

Figure 14. Coverage tree.

4.3 Ad-Hoc testing

Running the query stated above, the result obtained is more than a thousand rows with mills, boxes and rolls. The doubt that arises is whether every situation that could occur in the database is covered, or whether introducing modifications in the database might lead to the query returning wrong data.

Via the use of "LEFT JOIN", the following situations must be given with the test database:

- rolling mills without boxes in them,
- rolling mills with boxes,
- couples of mills and boxes without rolls laminating on them,
- couples of mills and boxes with rolls installed.

Of these situations, there are mills in the result obtained in the query that do not have boxes and mills that do, but there are no couples of mills and boxes without associated rolls. This means all mills with boxes in them have rolls and hence they are laminating.

Moreover, the structure of the database tables should be kept in mind when detecting fields that can be null and checking the existence of null and non-null values. In the case study, rolls may be laminating and thus their box code may have a null value. However, in the table "roll", there are no rolls with this field null. This means that all rolls are working in some box. In practice this is impossible, since rolls have to exist in the department for replacing those that are working with others when necessary.

Consequently, it seems that there are situations in the SELECT query that are not covered with the information loaded in the database. What is more, further hidden faults might be present. To eliminate these problems, new test data ought to be inserted into the database to cover all situations and avoid subsequent errors.

It should be verified whether all the conditions of the SQL query present true and false values as well as the constraints at field-level for null values. The amount of information managed (about twenty mills and boxes and more than a thousand rolls) means that it is not viable to check all situations manually in order to complete the test cases. Therefore, the need arises to automate the process as explained below.

4.4 Analysis of coverage results

Applying the tool and concepts described above, the coverage tree shown in Figure 14 is generated for the query and database of our case study. The conditions considered are in the "join" clauses; therefore the corresponding coverage tree that will be created will have three levels, one for each condition.

After running the algorithm, the obtained theoretical coverage value of the SELECT query is 19.23%. 100% coverage is not achieved, as only 15 out of the 78 possible situations established are tested.

In the following subsections, we will use this information to reduce the amount of database records and to complete it with new cases. Moreover, we will consider the constraints for null values and referential integrity.

4.5 Simplification of tuples

Owing to the number of records being handled and the number of comparisons that need to be performed for the evaluation, it is very complex to determine new tuples that complete the previously achieved result. To simplify matters, we make use of the values (traces) that have allowed new information to be incorporated into elements of nodes of the coverage tree while its evaluation was being carried out; at least the same percentage coverage result would be obtained with these. These traces are automatically generated by the tool.

For the "mill", "roll" and "box" tables, only the database rows shown in Table 2 will be necessary. Moreover, it is necessary to include the tuple indicated in Table 3 in the "mill" table so as to maintain the referential integrity of the "roll" table.

Furthermore, tuples in the "box" table should be inserted to maintain referential integrity, as there is information in the "roll" table with values for the box code that does not correspond to any information in the "box" table. Although the database constraints permit these situations, this indicates a potential error of referential integrity in the database structure.

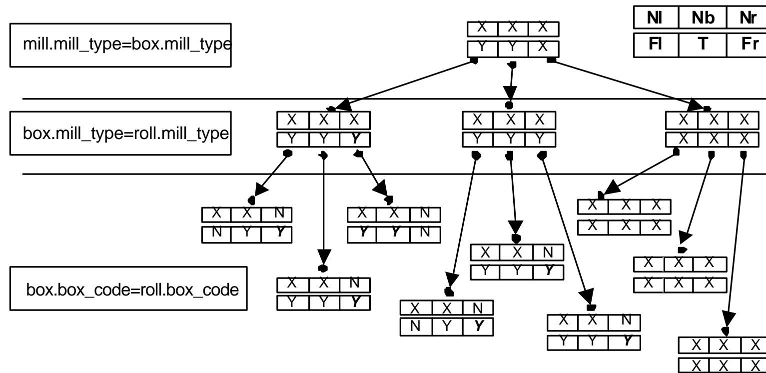

Figure 15. Coverage tree for query and simplified data.

Table 2. Simplified database information

mill

mill_name	mill_type
AP. TANDEM 1	TAN1A
DESCASCAR.	DES42
FINISHER F0	ACTF0
ACA F1/F6	ACTF6

box

box_code	box_name	mill_type
F0	CAJA F0	ACTF0
F1	CAJA F1	ACTF6
1	D. VERT	DES42

roll

box_code	roll_num	mill_type
F0	F211	ACTF0
F3	1569	ACTF6
1	K001	SKINP
F1	1558	ACTF6

Table 3. Information added to maintain referential integrity

mill

mill_name	mill_type
SKINPASS	SKINP

Obviously, the number of tuples for each table is much lower than in the original; only five rows are returned by the query and of course both the execution of the program and the loading of the database with these data are much quicker. Additionally, when the new coverage tree is measured for the SELECT query, the number of nodes achieved is extended due to the elimination of information and hence these encompass conditions that now are evaluated as false. The coverage tree for the query and simplified information is shown in Figure 15, in which the newly achieved nodes are marked in italics and the impossible ones are marked 'X'. Even tuples that cover the same situations as others can be deleted; for example the "roll" table tuple whose code table is "F0" covers the same situations as the one whose code box is "F1", and so the latter may be removed. In this case, the percentage of theoretical coverage is 29.49% and the schema coverage is 71.87%.

4.6 Completing the test data

The next task after simplifying information in the test database is to complete with new test cases in order to increase the coverage. In order to do so, the coverage tree generated will be examined. The null values will be completed first and then the remaining situations in a top-down fashion. New tuples for completing the database can be seen in Table 4.

As regards null values, there are none in the tables. Some are impossible to add: fields which are primary keys (mill type of "mill" table, box code of "box" table and roll number of "roll" table) and fields that have constraints in the database structure for null values (mill type of "box" and "roll" tables). These cases will be indicated in the coverage tree by an *X*, as these values are forbidden. However, it would be appropriate to try to insert null values to ensure that the database has considered these constraints and that they can be used in regression testing. (Cases 1, 2, 3, 4, 5).

Table 4. Tuples for completing the database

mill

Case	mill_name	mill_type
(1)*	TREN-NULL	NULL

roll

Case	box_code	roll_num	mill_type
(2)*	1	NULL	ACTF0
(3)*	1	3333	NULL
(6)	NULL	6666	ACTF0
(7)	NULL	7777	SKINP
(10)	10	1010	SKINP

box

Case	box_code	box_nam e	mill_type
(4)*	NULL	C-NULL	ACTF0
(5)*	F5	CT-NULL	NULL
(8)*	F8	CAJA F8	ACAAP
(9)	F9	CAJA F9	DES42

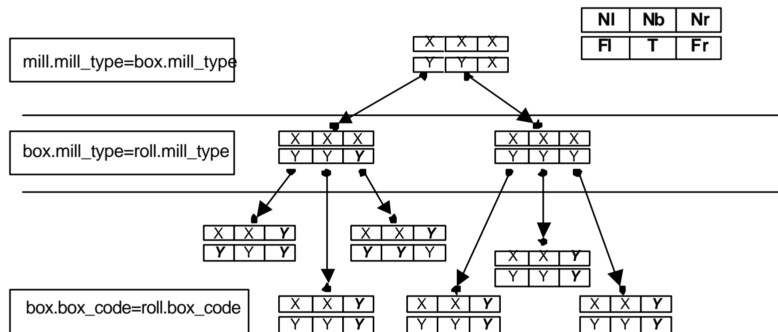

Figure 16. Coverage tree with the maximum possible coverage.

On other hand, there are fields that can be null but are never so: the box code of the "roll" table. To complete the test cases, it will be necessary to add two tuples to the "roll" table with a null value for their box code: one whose mill type coincides with one from the "box" table (Case 6), and another whose mill type does not exist (Case 7).

The level of the coverage tree for the condition "mill.mill_type=box.mill_type" indicates that:

- There are mill types in the "mill" and "box" tables that coincide (*T*).
- The condition is evaluated as false from left to right (*Fl*); therefore some mill type in the "mill" table does not correspond to any one in the "box" table.
- The condition is not evaluated as false from right to left (*Fr*), because all mill types in the "box" table coincide with some in the "mill" table.

These situations are the desired ones, as the E-R model establishes that mills without boxes may exist, but boxes cannot exist without being linked to a mill. However, it would be interesting to insert a tuple in the "box" table whose mill type was different from any one in the "mill" table so as to force the Database Management System (DBMS) to detect and indicate that it is impossible to add it (Case 8).

After simplification of tuples in the database, the level of the coverage tree for the condition "box.mill_type=roll.mill_type" has covered all possible situations.

On the level of the coverage tree corresponding to the "box.box_code=roll.box_code" condition:

- There are tuples from the "box" table whose box code coincides with tuples from the "roll" table, whether their mill type also coincides or not.
- The *Fl* situations are not obtained at two nodes on this level. This indicates that all the tuples in the "box" table coincide with respect to their box code with some from the "roll" table having a different mill type. Therefore, a new tuple in the "box" table should be inserted in the database whose mill type and box code are distinct to any one of the rolls at the same time (Case 9).
- The *Fr* situations are never achieved at any node on this level; i.e. all the rolls have a box code that coincides with one from the "box" table. In the light of the E-R model, this point seems to be correct. However, it would be appropriate to try to insert a roll with a box code that was not included in the "box" table to ensure the database has considered this constraint. Some of these situations are covered with the simplified database by means of the tuple in the "mill" table whose box code is "F3". However, it would be necessary to add a new tuple to this table with a mill type and box code that is simultaneously different to any of the tuples from the "box" table (Case 10).

With the aforementioned insertions, the coverage tree (Figure 16) evaluates the conditions for all possible values and with all combinations, and the maximum possible coverage of the query is 100% schema coverage. It is impossible to reach 100% theoretical coverage owing to referential integrity, but it would be necessary to test the insertions in the database with test cases marked with '*' in the Table 4.

Table 5. Result of query

mill_type	box_code	roll_num
TAN1A		
SKINP		
DES42	1	
DES42	F9	
ACTF6	F1	1558
ACTF0	F0	F211

In this case, the result of the query obtained is six rows with mills with and without boxes and couples of mills and boxes with and without rolls, as can be seen in Table 5.

Another fault is observed with the simplified database and Case 10: rolls may have a box code that does not figure in the "box" table. This means that the information will not be valid or fulfill the specifications. This would be a problem of database design and would need to be solved by means of including the field box code in the "roll" table as a foreign key of the "box" table, or by controlling

these situations by program code, with triggers or stored procedures.

5. CONCLUSIONS AND FUTURE WORK

To finish, a number of conclusions are enumerated below. Firstly, the most important aim considered: two different coverage measures for the coverage of SQL queries have been established, specifically for the case of the SELECT query, that are automatically calculated taking into consideration the information of database, the schema constraints and the SQL query. Like the measurement of coverage for imperative and structured languages, this is an indicator that helps improve designed test cases with the purpose of detecting faults in SELECT queries. In the case study, it was detected that although we have lots of real data in production, the total possible coverage is not achieved due to incomplete test information.

Furthermore, the number of tuples in the database may be simplified by tracing the coverage tree. This is useful for creating smaller databases and for detecting information that would have to be inserted to make them complete. Reducing the number of tuples in the tables leads to the detection of database design problems and helps to complete the test cases.

We next cite some points that have been established and which will influence current and future work.

On the one hand, the SQL queries analyzed until now are simple and isolated, although we plan to increase the complexity of the queries and carry out more studies that will confirm the obtained results, as well as including parameters in the query. Besides, greater complexity implies the evaluation of sets of queries, for example those found in a stored procedure. It will be necessary to integrate SQL coverage criteria with other criteria for imperative languages that would permit complete transactions or store procedures to be tested.

On the other hand, all the information on the database structure is not exploited. This could be used to improve the information supplied to the user with output data and to detect the specific test cases needed in a more automatic way, as well as those test cases that, even though an attempt was made to add them to the database, cannot be incorporated due to constraints in the tables.

As the number of nodes of the coverage tree might be large depending on the number of conditions in the query, another point to be studied is that of new ways of measuring coverage, also possibly based on other traditional measurements of coverage in imperative languages, such as decision/condition or full predicate where the combinations will not be so numerous, or incorporating different concepts.

6. ACKNOWLEDGMENTS

This work is supported by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, project TIC2001-1143-C03-03 (ARGO).

The case study data is based on information obtained by the project entitled AITOR, which was funded by the European Coal and Steel Community (ECSC) (7210-PR-148) and *Aceralia Corporación Siderúrgica* (CN-99-287-B1).

7. REFERENCES

[1] ANSI/ISO/IEC International Standard (IS). *Database Language SQL—Part 2: Foundation (SQL/Foundation). "Part 2"*. 1999

[2] Chan, M.Y. and Cheung, S.C. *Applying white box testing to database applications*. CSTR, Hong Kong University of Science and Technology, HKUST-CS99-01. 1999

[3] Chays D., Deng, Y., Frankl, P.G., Dan S., Vokolos, F.I. and Weyuker, E.J. *An AGENDA for testing relational database applications*. Software Testing, Verification and Reliability. 14 17-44. 2004

[4] Davies, R.A., Beynon, R.J.A. and Jones, B.F. *Automating the testing of databases*. 1st International Workshop of Automated Program Analysis, Testing and Verification. 2000

[5] Daou, B., Haraty, R.A. and Mansour, N. *Regression testing of database applications*. Symposium of Applied Computing. ACM. 2001

[6] Encontre, V. *Empowering the developer to be a tester tool.* Int. Symposium on Software Testing and Analysis, Industry panel. ACM SIGSOFT Software Engineering Notes. 2002

[7] Hartman, A. *Is ISSTA research relevant to industry?* Int. Symposium on Software Testing and Analysis, Industry panel. ACM SIGSOFT Software Engineering Notes. 2002

[8] Kapfhammer, G.M. and Soffa, M.L. *A family of test adequacy criteria for database-driven applications*. ESEC/FSE'03. ACM SIGSOFT Software Engineering Notes. 2003

[9] Slutz, D. *Massive Stochastic Testing of SQL.* 24th Very Large Data Base Conference. 1998

[10] SQLUnit Project. http://sqlunit.sourceforge.net

[11] Tassey, G. *The economic impacts of inadequate infrastructure for software testing.* National Institute of Standards and Technology. Planning Report 02-3. 2002

[12] Zang, J., Xu, C. and Cheung, S. C. *Automatic generation of database instances for white -box testing.* 25th International Computer Software and Applications Conference. 2001

[13] Zhu, H., Hall, P. A. V., May, J. H. R. *Software Unit Test Coverage and Adequacy*. ACM Computing Surveys, 49(4) 366-427. 1997