Improvement of test data by measuring SQL statement coverage

Mª José Suárez Cabal, Javier Tuya

Departamento de Informática. Universidad de Oviedo
cabal@correo.uniovi.es, tuya@lsi.uniovi.es

Abstract. Many software applications produced today have a component, of lesser or greater importance to the structure, that is based on database management systems. What is more, this information is generally handled through SQL queries embedded in the application code. On the other hand, automatic software testing is normally associated with the testing of programs implemented in imperative and structured languages. The problem arises when it comes to unifying software tests in programs that manage databases using SQL. The aim of this paper is to get closer to a measurement of the coverage of SQL statements and to show how, using this measurement, we might change the testing databases by means of completing or deleting information which provides improvements to the measurement, in order to achieve the highest possible percentage of coverage of the statements which have access to the database.

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

1. Introduction

Software Testing is one part of the processes known as Verification and Validation (V&V) [20] in software engineering. These processes determine whether the products developed in a particular activity meet their requirements and whether the software satisfies the user's needs and carries out the functions for which it was designed [8]. The aim of Software Testing is to discover the hidden faults that may have occurred during the tasks of specification, design and programming, to ensure that these faults do not occur and to reduce the cost caused by the software throughout the life of the product [16].

V&V are expensive processes; it is estimated that they can cost at least 50% of the total cost of the software developed. The automation of testing process enables the test cases designed to be more efficiently managed and helps to carry out regression tests, though always using test cases designed manually.

In the case of programs written in imperative languages, the automation of Software Testing is under study from various angles, depending on the tests that they are going to carry out: white box or black box [16]. The techniques for designing test data can be divided into four categories:

- Random techniques [17].
- Static techniques, applicable to both white [6] and black box tests. [14][15]. These techniques can use heuristics [2], which can optimize the generation of test data.
- Dynamic techniques, used for white box tests [10]. In general, these try to test the coverage of the code. Metaheuristic techniques are applied [4]: genetic algorithms applicable to branch coverage [9] [13] [19] and to path coverage [11], and simulated annealing [22].
- Another type of technique is the combination of static and dynamic techniques [18] [7].

However, it is common for a software application written in an imperative language to have components with access to databases through SQL statements embedded in the code. The tests to be carried out on these types of components become more complicated, mainly due to the information contained in the databases and to the SQL code itself. The database instances that are used to perform the tests on the software will determine their quality, as the situations set out in the queries must be covered, while at the same time avoiding producing undesired results [12]. When we talk about the contents of the database, we are not just referring to the fact that the combination of the values of tuples, according to the SQL statement, can cover all cases of execution of the query, i.e. it has the maximum coverage. There are few studies in the literature that focus on software testing together with databases, and the approaches are very different in each case. These are described in Section 2.1. "Work relating to databases and Software Testing".

The main objectives of the present paper are to:

- Present an algorithm that aids the carrying-out of software testing of applications with access to databases and SQL queries. It aims to obtain a measurement of the percentage of SELECT coverage by means of a database loaded with test data.
- Extract a subset of the tuples that allows the same result as the original information to be obtained.
- Guide the expert as to the database values needed to increase said percentage of coverage.

The problems encountered when carrying out tests with databases and SQL statements are set out in more detail in Section 2. Additionally, some of the published studies related to this topic are briefly described in subsection 2.1. Following this, a way of measuring the coverage of SELECT statements is established in Section 3 along with how it is applied. Subsequently, the results of the algorithm on the SELECT statements with a real database are described

in Section 4. Finally, in Section 5, we present the conclusions we have reached, together with future lines of research and development.

2. Databases and testing

When we perform tests on parts of software that contain access to databases with SQL queries incorporated, we can come up against problems or obstacles which may have various causes:

- The first task, as well as the first obstacle, is to design the initial loading of instances for the test database. The selection of this information is one of the most important steps in obtaining a good set of unit tests.
- Another problem when designing the loading database is to ensure that the data contained is useful for the greatest possible number of statements.
- The information contained in a database will not be static, but will modify itself through the very same SQL statements incorporated in the application code. Therefore, at each stage of carrying out the testing plan, the previous one will have an effect as to which data exists at any given moment.
- As in the case of imperative languages, SQL statements may be given parameters and contain variables and constants which intervene in the query itself. Hence, when designing the testing plan, this input must also be considered, and test data provided for it.
- Databases are usually large in size, as they may have many tables and views, and a high number of tuples. All this makes it difficult to design a complete set of instances for them. Furthermore, sets of SQL statements can be established which act upon specific parts of the information. Because of this, test data for subsets of the database can be generated which take into account only the set of statements to be tested and the data which is consistent with the structure of the tables.
- Another of the obstacles overcome lies in judging the efficiency of the unitary 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.

2.1. Work relating to databases and Software Testing

A number of studies have been found in the literature relating to the topic of test data generation for databases and programs or applications with incorporated SQL statements. Some of these articles are described below.

One of the studies [21] was carried out by the Microsoft research team. Its aim was to evaluate database management systems. In another study [5], a set of valid and invalid data was generated from the structure of the database and its constraints at field-level, not the referential integrity An attempt was then made to fill a database with this set of data and automatically obtain an initial database. In a further study [3], the prototype of a tool is presented whose function is to facilitate testing of applications with a database. This consists of various components that will enable the generation of input data to fill a database, respecting integrity constraints and taking into account the SQL queries of the application. Once it has obtained the output, this is compared with the expected results and the final state of the database is checked. According to the indications, so far only the first component has been developed. Finally, another study shows how to generate instances for databases from the semantics of the SQL statements of a program [23].

It can be seen that, perhaps owing to the problems outlined above, there are not many studies related to this topic, and further, that the approach of each of the authors is quite different from the others.

3. Measurement of coverage

As mentioned in the previous section, one of the problems encountered when performing tests of SQL statements and databases is that of judging the efficiency of the plan of the tests proposed. This means that, given an SQL statement with data from the database, is it possible to know whether all the possibilities of the query are covered?

As a solution, we propose a way of measuring the coverage based on the method used traditionally with imperative languages: multiple condition coverage [16], whereby all the combinations of the conditions have to take the values true and false.

Given the variety of SQL statements that we can find in an application, we have restricted the algorithm for the case SELECT queries according to the grammar in BNF notation, a simplification of that specified in SQL3 [1].

The conditions to be evaluated in a SELECT statement are extracted from the expressions in the FROM which included JOIN clauses and WHERE..

When the operands are columns, the problem that arises when evaluating the expressions is that we are not dealing with operations within a single pair of values, but with operations within sets of values. Therefore each value in the column or field of the first term must be compared with each of the values of the field of the second term of the expression.

In these cases, the result of an expression will not be true or false for pairs of values.

Two cases of SELECT queries are shown in Fig. 1. In the example on the left, the evaluation for the expression will be between two columns of two tables; in the example on the right, a SELECT statement with an expression formed by a column of a table and a literal is shown.

```
SELECT <select list>                           SELECT <select list>
FROM table_i LEFT JOIN table_j                 FROM table_i
   ON (table_i.field_i <OP> table_j.field_j)   WHERE table_i.field_i <OP> literal
```

Where <OP> can represent any operator {'=', '!=', '<', '>', '<=', '>='}.

Fig. 1. Examples of SELECT queries.

We suggest the following evaluation in these cases:

1. Some of the tuples of $table_i$ and $table_j$ that make the condition true with its corresponding values in $field_i$ and $field_j$. Therefore in this case the evaluation of the expression is TRUE.
2. A tuple of $table_i$ and any of the tuples of $table_j$, the condition is never made true with the values of $field_i$ and $field_j$ respectively. Therefore, in this case, the evaluation of the expression is FALSE.
3. A tuple of $table_i$, and a literal, so that they make the condition true with the value of $field_i$. In this case, the expression is TRUE.
4. For a tuple of $table_i$, with the value of $field_i$, the condition is never made true for any literal. In this case, the expression is FALSE.

In SELECT, statements can be various expressions, and each one will be evaluated, according to the values of the fields and literals, with the values TRUE and FALSE. For the evaluation of all, a tree is created in which each level represents an expression. Should there be more, each node will have two sub-nodes, one of which will evaluate the following expression when that of the current node is TRUE, and the other when it is FALSE. After evaluating the tree, the coverage may be established, and also therefore the multiple condition coverage taking into account the expressions of the SELECT query. The percentage of coverage is calculated according to the number of possible combinations of values in the expressions and the number of combinations found in the evaluation, using the following formula:

$$\% coverage = \frac{e}{(2^n - 1) * p} * 100 ,$$

where:

n: number of levels of the evaluation tree; this is the number of expressions of the query.

p: number of possible values that a expression can adopt once it is evaluated, which in the present measurement will have two values.

e: number of cases (elements of a node) that it has been possible to evaluate.

Once it has been established how to measure the coverage of a SELECT query, the algorithm is applied to a group of queries on a database used to perform the tests for a real application. The algorithm will take as inputs:

- Expressions that form part of the SELECT. The evaluation tree will be formed on the basis of these expressions.
- Database structure: tables and columns that appear in the query; these are necessary in order to take into account constraints such as null values.
- Information or tuples from the tables; these will be the values used for the evaluation of the expressions.

The outputs automatically obtained by the process are:

1. Consulting at each node the values of the expressions (TRUE and FALSE), the percentage of the SELECT coverage can be determined, achieving 100% coverage if all the possible values have been verified.
2. Bearing in mind the evaluation tree, nodes can be determined for which no values have been obtained. By observing the expression associated with the node, their parents information, the database structure and the tuples, the expert can be guided in finding the information missing from the test database to cover all cases.
3. During the running of the algorithm, an outline is generated of those tuples evaluated that generate new values for the expressions. By revising these annotations, a subset of tuples can be obtained which supply the same results as the original information, and which can drastically reduce the size of the test database.

4. Case study

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

4.1. Description

One of the company's departments deals with managing the lamination rolls used in the rolling mills for the manufacture of sheet steel. Each roll always works in the same mill, where the rolls are arranged in boxes, so that a mill may consist of one or more boxes, in which there are various rolls. 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.

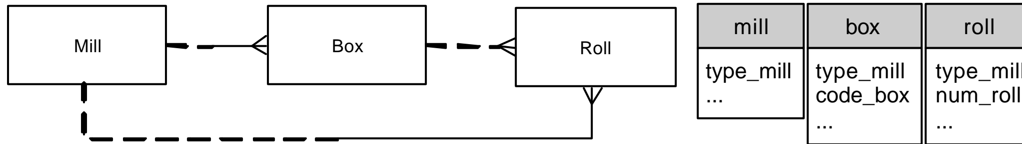

Fig. 2. E-R model and primary keys.

Fig. 2 shows the E-R model in which information on the mills, boxes and rolls are related, along with the tables associated with each entity and their fields representing the primary keys. The SQL query programmed to obtain the information about all of the mills, with their respective boxes and rolls is

"SELECT * FROM (mill LEFT JOIN box ON mill.type_mill=box.type_mill)
LEFT JOIN roll ON (box.type_mill=roll.type_mill) AND (box.code_box=roll.cod_box)"

As regards the database used for loading, it is worth highlighting the large amount of information, i.e. the number of registers it contains: 20 mills, 21 boxes and 1170 rolls. On running the query, the number of tuples resulting from the SELECT is 1186.

4.2. Measure of the coverage

The algorithm is applied to this query. The expressions considered are those within the two "joins", so an evaluation tree will be created with three levels, one for each condition.

The most important is how the evaluation of the query is performed. It is carried out by going over the tuples in the tables referred to in the expressions at each level of the tree. The evaluation will end when all the states of the tree have been verified, i.e. when 100% coverage has been achieved or when there are no more values to be compared. For a given node of the tree, the tuples of the first term are taken one after another, each one taking the tuples of the second term one at time, or else the literal value is taken. The expression evaluated is the one with the pair of corresponding values:

- If the result turns out to be TRUE, those tuples, from both the first and second terms, are fixed in order to evaluate the expressions of lower levels of the tree, by the branch at which the condition is fulfilled.
- If the result is NOT TRUE the following tuple of the second term is selected, and if all have already been evaluated and the result was in each case NOT TRUE, the condition works out to be FALSE. In this case, only the tuple from the first term is fixed in order to evaluate the expressions of the lower levels of the tree, by the branch at which the condition is not fulfilled.

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

In the test performed, the first result obtained is that 100% coverage is not achieved, only 86%.

The evaluation tree generated in this case is shown in Fig. 3.

As can be observed in the evaluation tree, the expression "box.code_box=roll.code_box", in two nodes of the evaluation tree, is never FALSE. This might be interpreted as all boxes and rolls that, despite having a different "mill type", have the same "box type" as any rolls, though boxes and mills might be not laminating and thus rolls not working.

With the information handled, all situations are not covered, so it is necessary to add new test cases to the database: rolls without an associated box or boxes with "mill code" and "box code" which do not correspond to any rolls.

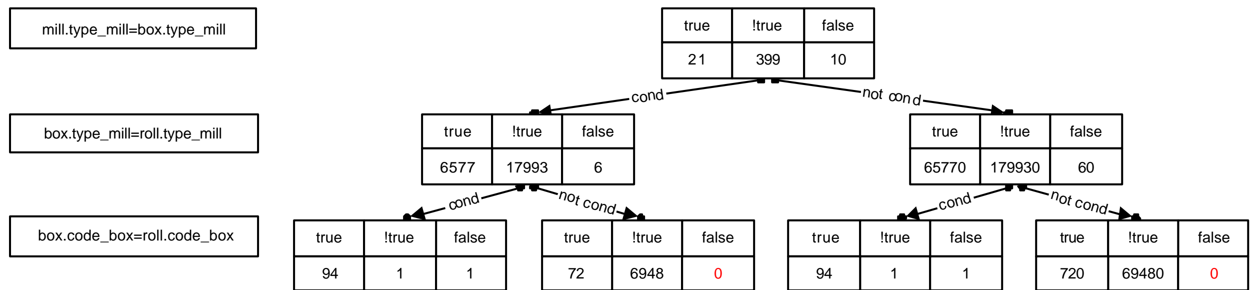

Fig. 3. Evaluation tree.

4.3. Simplification of tuples

With the number of records being handled, and the number of comparisons that need to be performed for the evaluation, it is extremely complicated to determine new tuples that avoid producing this result. To simplify matters, we can make use of the outline of the algorithm where the values of the tables that have been significant in the evaluation are indicated; the same result of coverage percentage would be obtained with these.

For the tables “mill”, “roll” and “box”, only the following entries will be necessary:

MILL		BOX			ROLL		
name_mill	type_mill	code_box	name_box	type_mill	code_box	num_roll	type_mill
AP. TANDEM 1	TAN1A	F0	CAJA F0	ACTF0	F0	F211	ACTF0
DESCASCAR.	DES42	F1	CAJA F1	ACTF6	F3	1569	ACTF6
ACABADOR F0	ACTF0	1	DESC VERT	DES42	1	K001	SKINP
ACA F1/F6	ACTF6						
SKINPASS	SKINP[1]						

Obviously, the number of records for each table is much lower than in the original; the number of rows returned by the query is only five, and the execution of the algorithm with this data is much quicker.

4.4. Completing test data

As we mentioned earlier, after simplify the data, the situation is still the same: in principle, the test database would appear to be incomplete because total coverage is not achieved. But now, the volume of information is more manageable and in view of the results and the database information, it can be clearly observed that all the “box code” (code_box) in the “box” table are associated with a “box code” in the “roll” table, even though the “mill type” (type_mill) is not the same. To achieve 100% coverage, it would be necessary to add a tuple to the “box” table which had, in “mill type” and “box code”, values that were different to any of those present in the “roll” table. For example, adding a box whose “code_box” is “F5”, “name_box” is “CAJA F5” and “type_mill” is “ACAAP”.

To maintain the constraints, it is necessary to manually include a record in the “mill” where “type mill” (type_mill) is the same as “type mill” in the new box added before.

With these insertions, the evaluation tree would evaluate the expressions to all possible values and with all the combinations, and the coverage of the query would be 100%.

Another fault that shows up is that all the rolls have a “box code”, even though this code does not figure in the table “box”. This means that the information will not be valid or fulfil the specifications. This would be a problem of database design and would need to be solved by means of including the field “box code” as a foreign key of the table “box” in the table “roll”, or by controlling these situations by code, with triggers or stored procedures.

5. Conclusions and future work

To end this article, we will point out various conclusions we have reached. Firstly, due to its being the most important objective we had to consider, we have established a measurement for the coverage of SQL queries, specifically, for the case of SELECT, which is an indicator that helps to improve designed test data, similarly the measurement for coverage of imperative languages.

In the tests performed, it was found that even with a lot of real data at our disposal, 100% coverage is not achieved, due to incomplete information. Therefore, having lots of information does not imply having a complete database.

[1] This record is included manually to maintain referential integrity with the table “roll”.

Furthermore, by tracing the evaluation tree the databases become simplified, which is useful for creating smaller databases and for discovering which information would have to be added to make them complete, given that it is easier with a few items of data to find which are missing than with a large amount of information. Decreasing the number of records in the tables leads to the detection of database design problems.

The points that we have established and which will influence future work are:

On the one hand, increasing the complexity of the statements and carrying out more studies that would confirm the results obtained so far. The tests performed are not complete, as they take isolated SQL statements, which of course do not constitute an application even though they form part of one. The complexity of the queries also implies the inclusion of parameters in the WHERE clause.

On the other hand, although the structure of the database is taken into account, not all its information is exploited. It could be used to improve the information supplied as output data, as well as equipping the tool with an intelligent algorithm that would enable the detection of the specific cases needed, as well as those cases which, even if we tried to add them to the database, would not be incorporated due to constraints in the tables.

Another point to be studied is a new way of measuring coverage, possibly also based on other traditional measurements of coverage in imperative languages, such as decision/condition coverage or incorporating different concepts.

6. Acknowledgements

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 study case data is based on information obtained by the project entitled AITOR, which was funded by the European Coal and Steel Community (ECSC) (CN-CECA-99-7210PR148) and *Aceralia Corporación Siderúrgica* (CN-99-287-B1).

References

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

[2] Burton, S., Clark, J., Galloway A. and McDermid, J. Automated V&V for High Integrity Systems: A Targeted Formal Methods Approach, In the Proceedings of the 5th NASA Langley Formal Methods Workshop. June 2000.

[3] Chays D., Dan S., Frankl, P.G., Vokolos, F.I. and Weyuker, E.J. A frame work for testing database applications. ISSTA, 2000

[4] Clarke, J., Dolado, J.J., Harman, M., Hierons, R., Jones, B., Lumkin, M., Rees, K. and Roper, M. Can software engineering be reformulated as search problem?. 2001

[5] Davies, R.A., Beynon, R.J.A. and Jones, B.F. Automating the testing of databases. Procedings of the first international workshop of automated program analysis, testing and verification, 2000

[6] DeMillo, R. A. and Offutt, A. J. Constraint-based automatic test data generation, IEEE Transactions on Software Engineering, 17(9). 1991.

[7] Ferguson, R. and Korel, B. The Chaining Approach for Software Test Data Generation, ACM Transactions on Software Engineering and Methodology, 5(1). 1996.

[8] IEEE Standards Software Engineering. Volumen One. 1999

[9] Jones, B., Eyres, D. and Sthamer, H. Generating test data for ADA procedures using genetic algorithms. 1st. IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems (GALESIA’95). 1995.

[10] Korel, B. Automated software test data generation, IEEE Transactions on Software Engineering, 16(8). 1990.

[11] Lin, J. and Yeh, P. Automatic test data generation for path testing using GAs. Information Sciences 131. 2001.

[12] Mannila, H., Räihä, K-J. Test Data for Relational Queries. ACM SIGACT-SIGMD. Symp. On Principles of database systems. 1986.

[13] McGraw, G., Michael C. and Schatz, M. Generating Software Test Data by Evolution. Technical Report RSTR-018-97-01, RST Corporation. 1998.

[14] Meudec, C. Automatic Generation of Software Tests From Formal Specifications. Doctoral Thesis, Faculty of Science of The Queen’s University of Belfast. 1997.

[15] Meudec, C. ATGen: Automatic Test data Generation using constraint logic programming and symbolic execution. Software Testing, Verification & Reliability 11(2). 2001.

[16] Myers, G. The Art of Software Testing. 1977.

[17] Ntafos, S. On Random and Partition Testing, Intl. Symp. on Software Testing and Analysis. 1998.

[18] Offutt, A. J., Jin, Z. and Pan, J. The Dynamic Domain Reduction Procedure for Test Data Generation, Software-Practice and Experience, 29(2). 1999.

[19] Pargas, R.P., Harrold, M.J. and Peck, R.R. Test data generation using genetic algorithms. The journal of software testing, verification and reliability, 9. 1999.

[20] Pressman, R.S. Ingeniería del Software. Un enfoque práctico. 4ª Edición Ed. McGraw Hill. 1997.

[21] Slutz, D. Massive Stochastic Testing of SQL. VLDB. 1998.

[22] Tracey, N., Clark, J. and Mander, K. Automated program flaw finding using simulated annealing. International Symposium on software testing and analysis. ACM/SIGSOFT. 1998.

[23] Zang, J., Xu, C. and Cheung, S. C. Automatic generation of database instances for white-box testing. COMPSAC. 2001.