Query-Aware Shrinking Test Databases

Javier Tuya, Mª José Suárez-Cabal, Claudio de la Riva
Universidad de Oviedo, Departamento de Informática
Campus Universitario de Gijón (SPAIN)
+34 985 182 049
{tuya, cabal, claudio} @uniovi.es

ABSTRACT
Keeping the test databases as small as possible leads to faster execution of tests and facilitates the task of completing the test cases and evaluating the actual outputs against the expected. In this paper we present an automated approach to database reduction that considers an initial database that may be a copy of a production database and the set of queries that are executed against it. The database is reduced in order to preserve the coverage of the data with respect to the queries attaining large reductions with very similar fault detection ability.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging - Testing tools

General Terms
Reliability, Experimentation, Languages, Verification.

Keywords
Software testing; Database testing; MC/DC; Test-Suite Reduction; Data Reduction; SQL Coverage.

1. INTRODUCTION
During the maintenance of software applications, two main kinds of functional testing activities are performed: (1) development or maintenance of test cases for new and updated features, and (2) regression testing. Existing research in testing for database applications focuses mainly on either the test data selection criteria by defining a number of coverage criteria [9,11,12,15,16] or the automatic generation of test cases [1-4,21]. There also exists some work on regression testing that focuses on the selection of test cases [10,20] or in the order of execution and the number of resets of the test database [7,8].

On the other hand, test-suite reduction aims to find a representative set of test cases to provide the same test coverage as an original test suite [14], with a trade-off between reduction and effectiveness in fault detection. In the context of database applications, this may be applied to the number of test cases or to the size of the test inputs.

Keeping test databases as small as possible entails a number of benefits for both activities. In regression testing, the test cases are executed faster because both the time of loading the test database and executing each test is shorter. When maintaining test cases, a small test database facilitates the design of new test inputs that consider the new situations required by the changed functionalities and the checking of the expected results against the actual results. If the tests are designed for a completely new feature, previous test databases may be reused and completed with meaningful test data to consider the new features under test.

The issue addressed by this paper is the reduction of the data that is present in a database in order be used as a basis for further testing. The reduction takes as input a previously developed test database or even a complete production database. In order to include in the reduced database only a small set of suitable data, discarding redundant data, the reduction is made for preserving the coverage of the original database against the queries that are executed. These queries can be taken from the execution of test cases or from the real usage of the application. This process is fully automated. A case study using a production database and real queries shows a good trade-off between reduction and fault detection ability. It attains large reduction of the number of rows in the database while maintaining very similar fault detection ability in relation with the original database.

The rest of the paper is organized as follows: Section 2 presents an overview of the coverage criterion and the general approach to database reduction along with an example. Section 3 details the procedure for performing the reduction and Section 4 presents the tool support available to automate this task. In Section 5 a case study is presented and finally, Section 6 concludes.

2. INTRODUCTORY APPROACH AND EXAMPLE
In this section we present the concept of coverage rules and outline how these rules are used to reduce a database taking into account a set of queries.

2.1 Coverage Rules
Consider the following SQL query:

```
SELECT * FROM departments D LEFT JOIN
  job_history H ON h.department_id=d.department_id
WHERE department_name NOT LIKE 'IT%'
  AND location_id<>1700
```

This query is executed against the example database HR (Human Resources) which is bundled with the Oracle Database

DBTest'09, June 29, 2009, Providence, Rhode Island, USA.

Management System. This database contains information about the jobs performed by the employees that belong to a department, plus some other master tables such as jobs, locations, regions and countries.

An approach to designing tests for SQL queries based on the MC/DC coverage criterion has been suggested previously in [17]. Using this approach, the tester should design a test database for covering the following situations:

- Include rows such that the where conditions on *department_name* and *location_id* are: (C1) both true, (C2) true and false and (C3) false and true, respectively.
- As *location_id* may be null (as indicated in the database schema), include rows such that (N1) *location_id* is null and the condition on *department_name* is true
- As there is a join, include rows such that (J1) there exists a master (*departments*) without detail (*job_history*), and (J2) there exist a detail without master (this situation is possible because the foreign key column *department_id* in *job_history* is nullable).

Each of these situations specifies a test requirement that can be expressed in SQL and constitutes a *Coverage Rule*. For instance, three of the coverage rules for the above query are:

```
(C1): SELECT * FROM departments D INNER JOIN
job_history H ON h.department_id = d.department_id
WHERE (department_name NOT LIKE 'IT%') AND
(location_id <> 1700)
```

```
(N1): SELECT * FROM departments D INNER JOIN
job_history H ON h.department_id = d.department_id
WHERE (location_id IS NULL)
AND (department_name NOT LIKE 'IT%')
```

```
(J1): SELECT * FROM departments D LEFT JOIN
job_history H ON h.department_id = d.department_id
WHERE ((H.DEPARTMENT_ID IS NULL)
AND (D.DEPARTMENT_ID IS NOT NULL))
AND (department_name NOT LIKE 'IT%' AND
location_id <> 1700)
```

Given a previously populated test database, the execution of the SQL statement that expresses a coverage rule will determine whether this situation is covered if the output produced by the rule contains at least one row.

The coverage rules used in this paper are based on Masking MC/DC [5] or Full Predicate Coverage [13] and allow to measure the coverage of a test database against a set of queries or be used as a test input selection criterion. A complete description of the coverage rules and the procedure for automatically obtaining them is detailed elsewhere [19] . The scope of this paper is the use of the coverage rules to reduce a previously populated database with the goal of preserving the same coverage as the original.

2.2 Database Reduction

Let us illustrate the approach with an example. The output produced by the coverage rule C1 when executed against the HR database is included in Table 1 (only relevant columns are shown, primary keys are in bold).

In order to have a reduced database that covers the rule we need only to consider the rows of tables *departments* and *job_history* that correspond to only one of the rows in the output presented in Table 1. For instance, the first row is kept, the reduced database will be composed by a department with *ID=20* and a job history record with *ID=20* and *start_date=17/02/96* which fulfills the rule C1. All other rows may be discarded.

Table 1. Output produced by rule C1

DEPARTMENTS			**JOB_HISTORY**	
ID	NAME	LOCATION	**START_DATE**	**ID**
20	Marketing	1800	02/17/96	20
50	Shipping	1500	03/24/98	50
50	Shipping	1500	01/01/99	50
80	Sales	2500	03/24/98	80
80	Sales	2500	01/01/99	80

In short, the approach is the following: We collect a set of SQL queries that are executed against the original database. Next, for each query we generate and execute each coverage rule and process the output produced by browsing each output row and selecting only one. The constituent rows will be added to the reduced database. For each rule the criterion for selecting an output row will be that of minimizing the cost of adding new rows to the reduced database, measured in terms of the number of additional rows that have to be added to the reduced database. Finally, the set of rows that have been kept will constitute the reduced database.

3. FINDING REDUCED ROWS

In this section we provide the internal details of the procedure for obtaining a reduced database using as the only source of information the database and a set of queries.

3.1 Data Structures

Figure 1 depicts the main data structures used in the process of finding the reduced rows.

During the reduction process, the information of the rows that are to be added to the reduced database is maintained in memory (only the values of the primary keys of the rows are stored). The *PhysicalDatabase* class stores the information about the tables and rows that will compose the final reduced database. The *QueryDatabase* class stores the same information, but related to the tables used by a single query. Additionally, the metadata for tables and the database itself are stored and linked to all databases and tables. The *Database* class is an abstract class that provides the common methods to locate and process tables. At this moment the *AliasDatabase* class may be ignored.

During the reduction process a single instance of *PhysicalDatabase* is created (*reducedDB*) and for each rule two instances of *QueryDatabase* (*bestDB* and *currentDB*) are created.

3.2 Reduction Procedure and Costs

The reduction algorithm will process each query and will add to the *reducedDB* the minimum set of rows that cover all rules. Given a set of queries, the algorithm to select the rows that are stored in the *reducedDB* is presented in Figure 2 and described below.

Firstly, the set of tables that take part in the query and their associated metadata are loaded both in *bestDB* and *currentDB* by calling *getTable()* and *getNewTable()* methods.

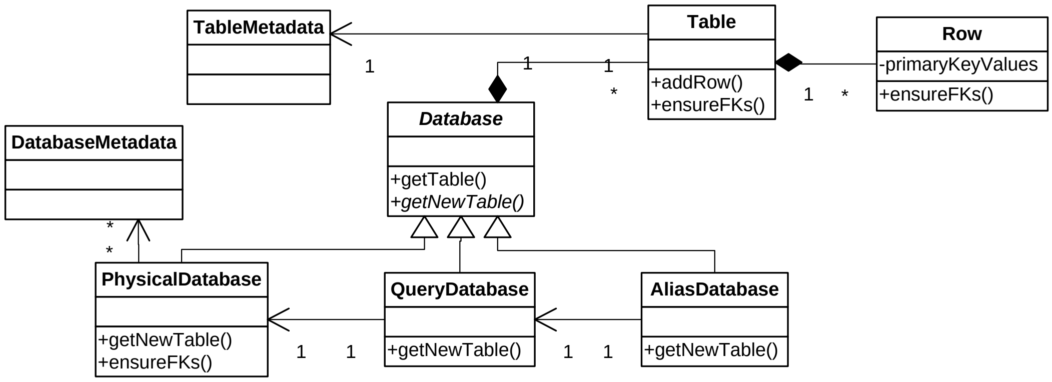

Figure 1. Main Data Structures used in the Reduction

Let reducedDB=new PhysicalDatabase()
***For each** query Q*
    ***For each** coverage rule of Q*
        Let bestDB=new QueryDatabase()
        Let currentDB=new QueryDatabase()
        Obtain and load in bestDB and currentDB all tables involved in the query (without rows)
        Execute the coverage rule against the original database
        ***For each** output row*
            Add the constituent rows to currentDB
            ***If** cost of inserting currentDB in reducedDB < cost of inserting bestDB in reducedDB OR the first row is being processed*
                Let bestDB=currentDB
        Add rows of bestDB to reducedDB

Figure 2. Algorithm to select the reduced rows

After all tables for a query have been loaded, each coverage rule is executed. The *bestDB* and *currentDB* store a solution which is constituted by a set of rows. The first one stores the best solution found up to now and the other stores the current solution. Then each output row is traversed. For each one, the primary keys of the constituent tables are used to create *Row* instances which are stored in *currentDB*.

Now the cost of inserting the rows of *currentDB* into the *reducedDB* is calculated as the number of rows of *currentDB* that are not present in *reducedDB*. So as, if the cost is lower than the cost of inserting *bestDB* into *reducedDB*, there is a new solution that replaces *bestDB* (with the exception of the first output row).

At the end, *bestDB* contains a set of rows that cover the rule and that are added to the *reducedDB*. The procedure continues for the next rule of each query.

3.3 Aliased Tables

Tables in an SQL query may be referenced using an alias so as that the same table may appear more than once in the query and handled as if it were a different table. However both refer to the same set of rows.

The *AliasDatabase* class is designed to handle this situation. All aforementioned operations on the *QueryDatabase* class are performed on *AliasDatabase* instead (*bestDB* and *currentDB* are instances of *AliasDatabase*) which forwards its calls another pair of instances of *QueryDatabase*. The structure of each instance of *AliasDatabase* is the same, but the only difference is that it does not store new rows, but a reference to the rows contained in the *QueryDatabase* instances instead. In this way a row added to a table using a given alias is visible when using any other alias that refers to the same table.

3.4 Populating the Reduced Database

After the reduction process, *reducedDB* contains in memory all the rows (primary keys) of the final database that preserve the coverage. However, some additional rows that refer to master tables must be added in order not to violate referential integrity. The procedure is handled by the *ensureFKs()* method which proceeds recursively for each table and for each row in *reducedDB*. At each row it looks for each foreign key and checks whether the corresponding referenced row has been loaded previously. If not, a recursive call to load each referenced row is made to ensure referential integrity. A flag is maintained for each visited row in order to perform the recursive calls only once per row.

Now all rows that are present in *reducedDB* ensure the referential integrity. The next task is to create the final reduced database. The approach taken is to perform a copy of the rows in *reducedDB* from the original database to the reduced database (which must have the same schema) as indicated in the steps below:

1. Temporally disable all foreign keys (this step is required if there are recursive relations between tables).
2. Perform a delete of all destination tables from the bottom to the top of the database schema (to perform a clean insert).
3. Execute the SQL commands that will copy each row in each table from the top to the bottom. An SQL query in the form INSERT INTO DestTable (columns) SELECT (columns) FROM OrigTable WHERE key columns are equal to the values in reducedDB.
4. Enable referential integrity

As an example, the following script would be generated for the introductory example. In order to keep it short, only two queries for each step are presented:

```
ALTER TABLE HR2.JOB_HISTORY DISABLE CONSTRAINT
  JHIST_DEPT_FK;
ALTER TABLE HR2.DEPARTMENTS DISABLE CONSTRAINT
  DEPT_LOC_FK;
DELETE FROM HR2.JOB_HISTORY;
```

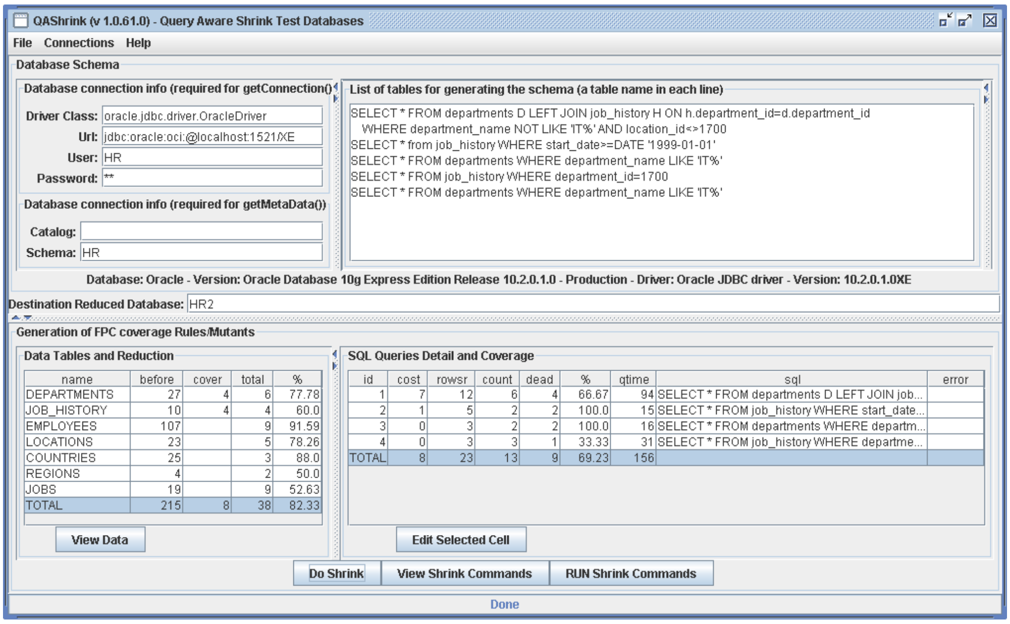

Figure 3. User Interface of the QAShrink Tool

```
DELETE FROM HR2.DEPARTMENTS;
INSERT INTO HR2.DEPARTMENTS (DEPARTMENT_ID,
  DEPARTMENT_NAME, MANAGER_ID, LOCATION_ID) SELECT
  DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID,
  LOCATION_ID FROM HR.DEPARTMENTS WHERE
  DEPARTMENTS.DEPARTMENT_ID=10;
INSERT INTO HR2.JOB_HISTORY (EMPLOYEE_ID,
  START_DATE, END_DATE, JOB_ID, DEPARTMENT_ID)
  SELECT EMPLOYEE_ID, START_DATE, END_DATE,
  JOB_ID, DEPARTMENT_ID FROM HR.JOB_HISTORY WHERE
  JOB_HISTORY.EMPLOYEE_ID=122 AND
  JOB_HISTORY.START_DATE=DATE '1999-01-01';
ALTER TABLE HR2.DEPARTMENTS ENABLE CONSTRAINT
  DEPT_LOC_FK;
ALTER TABLE HR2.JOB_HISTORY ENABLE CONSTRAINT
  JHIST_DEPT_FK;
```

4. TOOL SUPPORT

The *QAShrink* Tool (Query-Aware Shrink) has been implemented in order to automate the whole process. It can be downloaded from http://in2test.lsi.uniovi.es/sqltools/qashrink. Currently it has been tested with the Oracle and SQLServer database management systems.

The user interface of *QAShrink* is depicted in Figure 3. The user has to specify the connection information of the database and the set of queries that are to be used for the reduction. Additionally, the name of the destination database where the selected rows will be copied has to be specified. Firstly, it generates all coverage rules for the queries and then these rules are used to perform the reduction as explained in previous sections.

The figure presents the information after shrinking the example database taking into account five queries, the first one being the same as has been used in the examples. First the queries are analyzed to remove duplicates. After running the reduction (*Do Shrink* command) the size of the database is presented on the left side of the screen, with information about the original number of rows for each table, the number of rows that are loaded to satisfy the coverage rules, the total number of rows of the reduced database (which also includes the rows that are needed to ensure referential integrity) and the percent reduction (percentage of rows that are eliminated).

On the right side of the screen, information of each query is presented. The cost is the number of rows that have been added to ensure the coverage. It can be shown that in the figure, queries 3 and 4 have a zero cost, because all rows added for covering query 1 will also cover them. Also, there are only 4 queries because the last one is duplicated and then removed.

Once the shrinking has been done, the user may view the commands that will create the reduced database or execute them in order to populate the reduced database.

5. CASE STUDY

In this section we present a case study and show the results of the reduction. We use a real-life helpdesk application and a copy of the production database. The helpdesk application manages change and support requests (*Ticket*), which may be transferred to other technicians or change their state using history records (*TicketHistory*) associated to each ticket. The application has other capabilities such as keeping bookmarks, attachments, and a complete security subsystem that issues queries to determine whether it allows or denies access to the stored information on the basis of special privileges and groups for each type of transaction (create, update, read) into each object (ticket, history record, ticket list).

The production database used is implemented in SQL Server and has 22,387 tickets with 103,553 history records and 279 users. In total the database contains 137,490 rows spread over 31 tables.

Four main questions arise in relation to the feasibility of the approach presented in this paper. With regard to the efficiency:

a) What is the degree of reduction that can be attained?
b) What is the performance of the approach as automated by QAShrink?

With regard to the effectiveness:

c) Is the coverage measured in the original database preserved by the reduced database?
d) Is the fault detection ability of the original database preserved by the reduced database?

5.1 Efficiency Considerations

We recorded the queries that are being executed against the database during the use of the application in a number of different user sessions. In total, 988 queries were recorded which, after removing duplicates, led to 198 different queries used for the reduction process.

In relation with question a), Table 2 displays the size of each table before and after the reduction (CovAfter shows the number of rows after reduction, excluding the rows needed to preserve the referential integrity). The size of the database drops from 137,490 to 223 rows. The reduction is 99.84% measured as the percentage of rows eliminated. Although we can not claim that this is the optimum (because it would depend on the order of the queries), the magnitude of the reduction is very large, especially for the tables that have many rows.

Table 2. Size of the database for each table (before and after)

Table Name	Before	CovAfter	After	Red (%)
DBPermission	309	11	11	96.44
UserPreference	332	11	11	96.69
DataBase	5	2	4	20.00
SQLSELECTType	5	2	2	60.00
User	279	21	25	91.04
SQLTypeCriterion	52	7	9	82.69
Ticket	22,387	27	33	99.85
SQLCategoryCriterion	23	3	4	82.61
SQLStatusCriterion	37	5	6	83.78
SQLScopeCriterion	5	3	3	40.00
SQLOrderCriterion	13	4	5	61.54
SQLConjunction	2	1	1	50.00
SQLWhereFind	11	4	4	63.64
SQLOtherCriterion	5	2	2	60.00
Category	24	8	11	54.17
Status	32	16	18	43.75
Priority	3	2	3	0.00
OrganizationType	10	4	7	30.00
Type	66	7	14	78.79
Attachment	3,013	4	4	99.87
TicketAttachment	2,898	3	3	99.90
TicketHierarchy	3,880	4	4	99.90
Bookmark	321	6	6	98.13
TicketHistory	103,553	8	8	99.99
NextStatus	45	6	6	86.67
NextStatusUser	70	6	6	91.43
NextStatusUserAct	36	1	1	97.22
PermissionOnType	33	2	2	93.94
PermissionOnGroup	34	4	4	88.24
Organization	3		3	0.00
TypeClass	4		3	25.00
TOTAL	**137,490**	**184**	**223**	**99.84**

With regard to question b), the time needed to select all rows to be kept in the reduced database was 86.6 seconds, with a total elapsed time of 125.2 seconds plus 3.7 seconds in executing the SQL commands to copy the rows from the original database to the reduced one. An Intel Core 2 Duo 2.2 GHz processor with a local database has been used for the experiments. In total 335,191 output rows were read during the evaluation of all coverage rules.

That figures show a very good performance in the reduction of a production database, both in the size of the reduced database and in the time spent for this task.

5.2 Efficacy Considerations

In order to check the question c), the coverage rules were executed against both the original database and the reduced database and the percent coverage measured. This information is displayed in the first row in Table 3.

Table 3. Coverage and Mutation Score before and after the reduction

		Before Reduction		After Reduction	
	Number Rules/ Mutants	Covered/ Dead	Coverage/ Mutation Score	Covered/ Dead	Coverage/ Mutation Score
Coverage	1,869	889	47.57%	959	51.31%
Mutation Score	78,844	50,014	63.43%	49,703	63.04%

At first glance, the expected result would be to achieve a lower or equal coverage than using the original database, but the actual result is that coverage increases in the reduced database. After examining the cause of that increase, we found that this is caused by the rules intended to measure the coverage of joins. The original database is a copy of a production database that has been used for several years. Therefore, as all rows in most master tables have at least one related row in their detail table, these rules are not covered. However, as the reduction process removes many rows, the final database contains the situations in which there is some row in a master table without any related row in the detail, leading to a coverage increase.

In relation with question d), we generated a set of mutants for each query. These mutants include conventional mutants and others specifically designed for SQL [18]. As in the previous case the mutation score was measured against both the original and the reduced database and the results given in the second row in Table 3.

In this case the mutation score decreases, but less than 0.5%, which is a very good score, taking into account that the reduced database has far fewer rows. Although not directly comparable, these results show lower effectiveness loses than other results for non database applications [14]. That implies that the reduced database contains a very diverse set of rows that are good enough to be used for testing purposes in the sense that they have approximately the same fault detection ability measured in terms of mutants.

6. CONCLUSION

We presented an automated approach for the reduction of test databases which attains a good trade-off between the percentage of reduction and the fault detection ability. Given a set of queries and an original test database we select a minimum set of rows that satisfy the coverage of the database with respect to the queries and populate a new reduced test database which only contains the selected rows. As shown in the case study, it performs efficiently and the resulting reduced database has very similar fault detection ability to the original one. The case study shows the results obtained from a production database which has been reduced by 99.84% (measured in the number of rows that are eliminated), while the mutation score only decreases by less than 0.5%.

This work may be completed in two main different directions. First, this approach takes isolated queries and does not consider the effects of updates which may have influence on the subsequent queries. These effects and the possible combination into a complete framework of test suite reduction (both reducing the number of tests and the data) may be a point for future research. On the other hand, currently queries with joins and alias are considered, but additional work must be done to incorporate some other features such as views, subqueries or groupings.

Nevertheless, the approach is potentially useful in helping the tester in the elaboration of new test cases or complete existing ones by starting from a previously populated database which includes a suitable set of test data taken from other tests or from a production database.

7. ACKNOWLEDGMENTS

This work has been funded by the Department of Science and Innovation (Spain) and ERDF Funds (TIN2007-67843-C06-01), the Government of the Principality of Asturias (CN-07-168) and the Government of Castilla-La Mancha (PAC-08-0121-1374).

8. REFERENCES

[1] Binnig, C., Kossmann, D. and Lom, E. MultiRQP - Generating Test Databases for the Functional Testing of OLTP Applications. *Proceedings of the 1st international workshop on Testing database systems*. ACM New York, NY, USA, 2008.

[2] Binnig, C., Kossmann, D. and Lo E. Reverse Query Processing. *Proceedings of the 23rd International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, 2007; 506-515.

[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* 2004; 14 (1): 17-44.

[4] Chays, D., Shahid, J. and Frankl, P.G.. Query-based Test Generation for Database Applications. *Proceedings of the 1st international workshop on Testing database systems*. ACM New York, NY, USA, 2008.

[5] Chilenski, J.J. *An investigation of three forms of the modified condition decision coverage (MC/DC) criterion*. Technical Report DOT/FAA/AR-01/18, U.S. Department of Transportation, Federal Aviation Administration, April 2001.

[6] Emmi, M., Majumdar, R. and Sen, K. Dynamic Test Input Generation of Database Applications. *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ACM, New York, NY, 2007; 151-162.

[7] Haftmann, F., Kossmann, D. and Kreutz, A,. Efficient Regression Tests for Database Applications. *Proceedings of the 2nd Conference on Innovative Data Systems Research*, 2005: 95-106.

[8] Haftmann, F., Kossmann, D. and Lo, E. A framework for efficient regression tests on database applications. *The VLDB Journal* 2007; 16 (1): 145-164.

[9] Halfond, W.G.J. and Orso, A. Command-Form Coverage for Testing Database Applications. *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, Washington, DC, 2006; 69-80.

[10] Haraty, R.A., Mansour, N., and Daou, B. Regression test selection for database applications. In Siau K (Ed.), *Advanced Topics in Database Research*, vol. 3, Idea Group, 2004; 141-165.

[11] Kapfhammer, G.M. and Soffa, M.L. A Family of Test Adequacy Criteria for Database-Driven Applications. *Proceedings of the 9th European software engineering conference /11th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM Press, New York, NY, 2003; 98–107.

[12] Kapfhammer, G.M. and Soffa, M.L. Database-Aware Test Coverage Monitoring. *Proceedings of the 1st conference on India software engineering conference*. ACM, New York, NY, 2008; 77-86.

[13] Offutt, J., Liu, S., Abdurazik, A. and Ammann, P. Generating test data from state-based specifications. *Software Testing, Verification and Reliability* 2003; 13(1): 25–53.

[14] Rothermel, G., Harrold, M.J., von Ronne, J. and Hong, C. Empirical studies of test-suite reduction. Software Testing, Verification and Reliability 2002; 12(4): 219-249.

[15] Suárez-Cabal, M.J. and Tuya J. Using an SQL Coverage Measurement for Testing Database Applications. *Proceedings of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM, New York, NY, 2004; 253 – 262.

[16] Suárez-Cabal, M.J. and Tuya J. Structural Coverage Criteria for Testing SQL Queries. *Journal of Universal Computer Science 2009*; 15(3): 584-619.

[17] Tuya, J., Suárez-Cabal, M.J. and de la Riva, C. A practical guide to SQL white-box testing. *ACM SIGPLAN Notices* 2006; 41(1).

[18] Tuya, J., Suárez-Cabal, M.J. and de la Riva, C. Mutating Database Queries. Information and Software Technology 2007; 49 (4): 398-417.

[19] Tuya, J., Suárez-Cabal, M.J. and de la Riva, C. Full Predicate Coverage for Testing SQL Database Queries. Submitted, February 2009.

[20] Willmor, D. and Embury, S.M. A safe regression test selection technique for database–driven applications. *Proceedings of the 21st IEEE International Conference on Software Maintenance*. IEEE Computer Society, Washington, DC, 2005; 421-430.

[21] Willmor, D. and Embury, S.M. An Intensional Approach to the Specification of Test Cases for Database Applications. *Proceedings of the 28th international conference on Software engineering*. ACM New York, NY, USA, 2006; 102-111.