International Journal of Information Technology and Knowledge Management
January June 2009, Volume 2, No. 1, pp. 217-222

MUTATION TESTING IN DATABASE SYSTEMS

Rakesh Kumar[*], Surjeet Singh[**], & Priyanka Gupta[***]

In competitive and fast moving world of systems, reliability for services is must required attribute. Systems that fail on regular basis are not at all accepted. Such level of reliability is provided by dependable systems.

Mutation testing can be used to enhance the reliability of a system by keeping a check on completeness and effectiveness of test cases. This paper discusses need of dependable systems followed by discussions of database systems as dependable. After that use of mutation operators in procedural, Object Oriented and declarative systems is discussed. At the end some mutations for SQL Queries are suggested.

Keywords: Dependable systems, Software Testing, Test case generation, Mutation Testing, Mutation operators

1. Introduction

The ever accelerating trend towards sophisticated computing systems pervades application areas where computing services need to function with great reliance. In this over whelmed world of services & computing, only those systems are sustainable & worth which can imprint their images on the hearts of users with the reliability of services they provide. As general expectations of quality in all types of systems have increased, it is not at all acceptable to have a system which fails on regular basis. Failure of systems leads to high costs for end users & trouncing of business opportunities for the developers. Such high level of reliance in all application domains can be provided by Dependable systems. A dependable system is always Available (ready for use when needed), Reliable (able to provide continuity of service while it is used), Safe (does not have a catastrophic environment consequence on the environment), Secure(able to preserve confidentiality) [Gill (2002)]

Database systems are an important asset for almost all businesses. DBMS often persists mission critical data which is updated by many applications & accessed by thousands of end users. Same data is used by all levels of management & it also supports critical business processes & decision making. Therefore these systems need to be safe, secure & reliable all of the times & failure is consequential to big loss of data & business. So dependability is an essential attribute of such systems.

Different approaches to achieve dependability (Fault Avoidance, Fault Removal, Fault Tolerance, & Fault Evasion) have been suggested in literature. Fault Avoidance employs various tools & techniques to design the system in such a manner that the introduction of faults is minimized. An avoided fault is never need to be dealt with at later times. Fault tolerance capability in a system manages to keep it operating, perhaps at a degraded level in the presence of faults. Practice of Fault evasion keeps a check on the behavior of system. When a system deviates from its normal behavior, system is reconfigured to reduce the stress on a component with a high failure potential. Fault removal exploits verification & testing techniques to locate faults. Removal of a fault is much more expensive than its avoidance. Testing the systems rigorously to locate & remove faults can be a handy solution to achieve system dependability.

2. Rigorous Testing & Adequacy of Test Cases

System failure occurs due to the existence of faults in the system. System reliability is a function of the number of failures experienced by a particular user of that system. System dependability can be enhanced by making system fault free. Testing is a mechanism which shows the presence of faults in the system. Rigorous testing of the system ensures quality & testing Quality depends upon Test cases. Test case can be defined as a set of input values, execution preconditions, expected results & execution post conditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement [Glossary(2006)].

To make a system fault free it is mandatory to have adequate test cases so that all faults existing in systems can be revealed. Different test adequacy criteria exists like statement coverage, branch coverage, path coverage, all uses & all definitions as found in Patrick, Hall & May (1997). Basili & Selby (1987), Roper, Miller, Brooks & Wood (1993), Lott, & Rombach, (1997), Kamsties & Lott (1995)

[*] Department of Computer Science and Applications,K.U., Kurukshetra, Haryana, India. *E-mail: rsagwal@rediffmail.com*

[**] Department of Computer Science, G.M.N. College, Ambala Cantt, Haryana, India. *E-mail: surjeetsagwal@gmail.com*

[***] Department of Computer Applications, MAIMT, Jagadhri, Haryana, India. *E-mail: gupta800@gmail.com*

have done statistical & experimental studies to compare the effectiveness of different testing strategies. It has been observed that irrespective of the technique used, there are some residual faults in the software which directly relates to the completeness & effectiveness of test cases selected. Therefore there is a strong requirement on the study of techniques using which the completeness & effectiveness of test cases can be verified in order to make systems dependable.

3. MUTATION TESTING & COMPLETENESS OF TEST CASES

According to Grigorjev, F., Lascano, N., Staude, J. (Motorola Global Software Group) Fault seeding can be used as a valid methodology for predicting the amount of residual errors. The methodology relies on the assumption that if we insert a known and controlled number of seeded errors and measure the proportion of them discovered by the test process, that proportion could be used to predict the number of real errors yet to be discovered. Fault insertion can give insight as to where testing should be concentrated and how much should be done. It might also provide insights about the real error density distribution. One of the widely used error seeding method is mutation testing, a mechanism to determine test set thoroughness by measuring the extent to which a test set can discriminate the program from slight variations of the program. Demillo, Lipton, & Sayward (1978) & Budd, Lipton, Demillo, & Sayward, (1980) have discussed Mutation as a fault-based testing technique. According to Morell (1990) Fault-based testing aims at demonstrating the absence of pre specified faults in a program. Therefore, performing mutation-based testing helps an implementation to be free from specific faults.

The mutation method is a fault-based testing strategy that measures the quality/adequacy of testing by examining whether the test set used in testing can reveal certain types of faults. Mutation Testing is based on two basic Assumptions:

(a) *The Competent Programmer Hypothesis*: In general programmers are competent. i.e., the programs they write are nearly correct. The program differs from a correct version in only a few small ways.

(b) *The Coupling Effect Hypothesis*: Large program faults, particularly those of a semantic nature are coupled with smaller syntactic faults that can be detected with mutation testing.

The core of a mutation-based testing is a set of operators, each of which modifies the source code to inject a fault. The modified program is known as a mutant. A mutant is said to be killed relative to a test data set, if at least one test case generates different results between the mutant & the implementation. Otherwise, the mutant is live. If no test case can kill a mutant, then it is either equivalent of the original implementation or a new test case needs to be generated to kill the live mutant, a method of enhancing a test data set. The adequacy of a test data set is measured by a *Mutation Score (MS) (MS = 100 × D/ (N – E) where D* = Dead mutants, *N* = Number of mutants, & *E* = Number of equivalent mutants). A set of test cases is *mutation adequate* if its *MS* is 100%. Thus mutation helps in verification of the completeness as well as effectiveness of test cases, which will further help in making systems dependable.

4. MUTATION OPERATORS IN PROCEDURAL & OBJECT ORIENTED PARADIGMS

Mutation operators lie at the heart of mutation testing. Different mutation operators for different languages have been suggested & worked upon by many researches. Mothra mutation operators are the most prominent among procedural languages. Mothra has used 22 mutation operators for FORTRAN & 70 for C language [Demillo & Orutt (1991)]. These operators are designed to represent common mistakes that a programmer might make. These particular mutation operators represent more than 10 years of refinement through several mutation systems. These operators explicitly require that the test data meet statement & branch coverage criteria, extreme values criteria, domain perturbation, & that they also directly model many types of faults. A comprehensive list of Mothra operators can be found in Offut, Rothermel, Untch & Zapf (1996) & King & Offutt (1991). Traditional mutation operators are not sufficient for Object Oriented Systems. In addition to these operators a list of class mutant operators can be found at Kim, Clark & Mcdermit (2000). But these were sufficient to deal with errors within a class only. But there is a need to emphasize on errors across the class boundaries. For that purpose again some mutant operators are suggested in Offutt, Ma & Kwon, (2000) & Ma, Kwon, Offutt (2002).

5. MUTATION OPERATORS IN SQL

To make DBMS dependable, mutation can play a big role as completeness of test cases can be ensured by mutation. Different mutation operators for SQL has been proposed by Tuya et al.(2007). These operators cover a wide spectrum of SQL features. They are organized in four categories: Mutations for the main SQL clauses (*SC*), Mutations for the operators that are present in conditions & expressions (*OR*), Mutations related to the handling of NULL values (*NL*), Replacement of identifiers: column references, constants & parameters (*IR*).Details of operators are given in Table 1. A set of SQL mutants based on features present in a conceptual model of the database schema has been presented by Chan & Cheung (2005).

Automatic Generation of Mutants of SQL Queries is also possible with the help of a tool named SQL Mutation

Table 1
SQL Mutation Operators

Operators	*Description*
SC Category	
SEL (select clause)	Replacement of SELECT/ SELECT DISTINCT
JOI (join clause)	Replacement of JOIN type keywords
SUB (sub query predicates)	Replacement of predicates in eRp(Q)
GRU (groupings)	Removal of group-by-expressions
AGR (aggregate functions)	Replacement of aggregate functions
UNI (Query concatenation)	Replacement of UNION keywords and removal of participating queries in UNION
ORD (ordering of the result set)	Change in direction and removal of order-by-expressions.
OR Category	
ROR(relational op. replacement)	Replacement of relational operators
LCR (logical connector operator)	Replacement of logical operators
UOI (unary operator insertion)	Replacement in arithmetic expression or reference to a number e by $-e$, $e+1$, $e-1$.
ABS(absolute value insertion)	Replacement in arithmetic expression or reference to a number e by abs(e) and -abs(e)
AOR(arithmetic op. replacement)	Replacement of arithmetic operators
BTW(between predicate)	Replacement in conditions like a BETWEEN *x* AND *y*
LKE (like predicate)	Replacement and removal of wildcard characters in a LIKE *s*
NL Category	
NLF (null check predicates)	Replacement of NULL keyword
NLS(null in select list)	Replacement of column reference *c* by function ifnull(*c*, *r*)
NLI (include nulls)	Forces the true value of the condition when value is null
NLO (other nulls)	Completes the other combinations of nulls
IR Category	
IRC (column replacement)	Replacement of column references by other column references, constants, parameters
IRT (constant replacement)	Replacement of constant by other constants, columns and parameters
IRP (parameter replacement)	Replacement of query parameter reference by other parameters, constants and columns
IRH (hidden column replacement)	Replacement of column attribute reference by other columns

as proposed by Tuya, Suarez-Cabal, Riva (2006). This tool is publicly available on the Web & it can be accessed using two different interfaces: A Web application to interactively generate the mutants & a Web service that allows it to be integrated with other applications developed using different platforms.

An SQL injection attack consists of insertion or injection of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system & in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands. Mutation testing has been applied to test such SQL injection vulnerabilities (SQLIVs) by Shahriar (2008). Nine mutation operators divided into two categories have been proposed. The first category consists of four operators that inject faults into where conditions (WC) of SQL queries. The second category consists of five operators that inject faults in database API method calls (AMC).Details are in Table 2.

Table 2
Operators for SQL Injection Vulnerabilities

Category	*Operators*	*Description*
WC	RMWH	Remove WHERE keywords & conditions.
	NEGC	Negate each of the unit expression inside where conditions.
	FADP	Prepend "*FALSE AND*" after the WHERE keyword.
	UNPR	Unbalance parentheses of where condition expressions.
AMC	MQFT	Set multiple query execution flags to true.
	OVCR	Override commit & rollback options.
	SMRZ	Set the maximum number of record returned by a result set to infinite.
	SQDZ	Set query execution delay to infinite.
	OVEP	Override the escape character processing flags.

All of the above operators are suggested for mutations in SQL. But the list is not complete. Many more mutations can be suggested in this field. Some of them are as follows:

(a) Many operators are suggested by tuya *et al.* (2007) for *select* command, but there is no suggestion for *insert* commands in SQL. The syntax of *insert* command is: *"Insert into <table name> (<column name 1>, <column name 2>) values (<expression 1>, <expression 2>)"*. Two mutations can be

suggested here regarding the number of expressions to be used that will check for the data integrity. (1) Reducing the number of expressions, (2) Increasing the number of Expressions.

(b) Another mutation can be for the *create table* command when a table need to be created from the existing table. Command syntax is: *Create table <table name> (< column name>, < column name>) as select < column name>, < column name> from <table name>*. Mutation suggestions are as follows: (1) Add where clause if it is not there in from part of command. (2) Remove Where clause if it exists & further change in conditions in where clause can be applied. Different mutations for where clause used in select, delete & update operations are suggested by Shariar (2008).

(c) Mutation operators for searching range has been suggested by Tuya *et al*. (2007). Some additions to that can be: (1) Replace *between* by *not between* & vice versa. (2) Replace *in* by *not in* & vice versa for comparing purpose.

(d) Some mutations for scalar functions in addition to suggestions by Tuya *et al.* (2007) can be: mutations in numeric functions (1) *greatest* function can be replaced by *least* & vice versa. (2) *Truncate, Round* & *floor* can be interchanged with each other.

(e) Some suggested mutations for string functions are: (1) *lower, upper* & *initcap* can be interchanged with each other. (2) replace *compose* & *decompose* with each other. (3) Interchange *trim, ltrim* & *rtrim* with each other. (4) replace *lpad* & *rpad* with each other.

(f) Some suggested mutations for conversion functions are: (1) replace *to-number* function with *to-char* function & vice versa.

(g) Manipulation of date can be done by to-char function in which there can be use of *TH, SP, SPTH*. Some suggested mutations for date functions: (1) each of these *TH, SP, SPTH* can be replaced by each other.

(h) Suggested mutations for miscellaneous functions: (1) replace *UID* by user & vice versa.

(i) In addition to mutations suggested by Tuya *et al*. (2007) for Group By clause following can be done: (1) *Having* clause can be altogether removed from group by clause & all other variations on *having* clause can be done as suggested by Tuya *et al*. (2007). (2) Replacement of group functions by other group functions in having clause of group. (3) Replacement of *having* with *distinct* & vice versa. (4) *Rollup* & *cube* operators can be replaced by each other. (5) Replacement of union, intersection & minus clauses on multiple Queries. When union converted to intersection or minus, distinct keyword should be added.

(j) Indexes are used in SQL for fast access. Mutations on create Index command can be as follows: (1) Add *Unique* keyword if it doesn't exists & remove if it exists. (2) Add *reverse* keyword if it doesn't exists & remove if it exists. 3) Add *bitmap* keyword if it doesn't exists & remove if it exists.

(k) SQL grants privileges to users for security purpose with the help of Grant statement. Suggested mutations are as follows: (1) type of privileges comes from the set {Alter, delete, index, insert, select, update}. Each of these can be replaced with other in object privileges part of the command. (2) Number of privileges can be increased for some user & for some can be decreased.

(l) The Data Dictionary is a series of tables & views that contain information about structures & users in the database. The data dictionary can be queried for information about database objects that are owned or on which access rights have been granted as said by Bayross (2005). To get the structure information for object relational as well as object oriented database tables sample queries can be as: Select column_name, data_type From USER_TAB_COLUMNS Where Table_Name = 'Customer'; & Select attr_name, length, attr_type_name From USER_TYPE_ATTRS Where Type_name = 'Person_ty'; suggested mutations are as follows: (1) USER_TAB_COLUMNS & USER_TYPE_ATTRS can be replaced by ALL_TAB_COLUMNS & ALL_ TYPE_ATTRS & vice versa.

Proposed Mutations operators are summarized in the following table.

6. CONCLUSION

Database systems are very crucial operative systems. Loss of data can lead to the loss of whole business; therefore they need to be dependable. Dependability of systems can be taken care of by number of techniques. Mutation testing can provide a good support by keeping a check on completeness & effectiveness of test cases. It is a fault-based testing strategy that measures the quality/adequacy of testing by examining whether the test set used in testing can reveal certain types of faults. Unlike other fault-based strategies that directly inject artificial faults into the program, the mutation method generates simple syntactic deviations (mutants) of the original program, representing 'typical' programming errors. Many mutation operators have proved their worth in procedural & Object oriented systems. In this paper the use of mutation operators in procedural, object

Table 3
Suggested Mutation Operators in SQL

Category	*Op.*	*Description*
II(insert into)	RNE	Reducing number of expressions
	INE	Increasing number of expressions
CT(create table)	AWC	Add *where* clause if it doesn't exists in from part of command
	RWC	Remove *Where* clause if it exists & further change in conditions in *where* clause can be applied
SR(searching range)	RNB	Replace *not between* by *between* & vice versa.
	RNI	Replace *not in* by *in* & vice versa.
NF(scalar functions)	RGF	Greatest numeric function can be replaced by least & vice versa.
	TRF	*Truncate, Round* & *floor* can be interchanged with each other
SF(string functions)	LUI	*lower, upper* & *initcap* can be interchanged with each other
	RCD	Replace *compose* & *decompose* with each other.
	TLR	Interchange *trim, ltrim* & *rtrim* with each other.
	RLR	Replace *lpad* & *rpad* with each other
CF (conversion func.)	RNC	replace *to-number* function with *to-char* function & vice versa.
DF (date functions)	RDF	Replace each of TH, SP, & SPTH with each other.
MF (misc. functions)	RUU	Replace UID by user & vice versa.
GC(group-by clause)	RHC	*Having* clause can be altogether removed from *group by* clause & all other variations on *having* clause can be done
	RGF	Replacement of group functions by other group functions in having clause of group
	RHD	Replacement of *having* with *distinct* & vice versa
	RRC	Rollup & cube operators can be replaced by each other
	UIM	Replacement of union, intersection & minus clauses on multiple Queries
CI(create index)	ARU	Add *Unique* keyword if it doesn't exists & remove if it exists.
	ARR	Add *reverse* keyword if it doesn't exists & remove if it exists
	ARB	Add *bitmap* keyword if it doesn't exists & remove if it exists.
GP(grant privileges)	IFS	Type of privileges comes from the set {Alter, delete, index, insert, select, update}. Each of these can be replaced with other in object privileges part of the command
	IDP	Number of privileges can be increased for some users & for some can be decreased.
DQ(data dictionary query)		RDK USER_TAB_COLUMNS & USER_TYPE_ATTRS can be replaced by ALL_TAB_COLUMNS & ALL_TYPE_ATTRS & vice versa.

oriented, & declarative languages is reviewed. Some new mutations operators in SQL have been identified. As a further work, the research is to be carried out in identification of mutation operators in PL/SQL & orthogonality of mutation operators in declarative paradigm & in PL/SQL.

References

[1] Budd, T., Lipton, R., Demillo, R., and Sayward, F. Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs In *Proceedings of the Seventh Conference on Principles of Programming Languages*. Las Vegas: *ACM Press*, (1980), 220–233.

[2] Basilli, V. R. and Selby, R. W. Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering,* **SE-13**, (12), (1987).

[3] Chan, W. K., Cheung, S. C. and Tse, T. H. Fault-Based Testing of Database Application Programs with Conceptual Data Model In Proc. of the *Fifth International Conference on Quality Software,2005*. IEEE Computer Society Press, Los Alamitos, California, (2005), 187–196.

[4] Demillo, R., Lipton, R. and Sayward, F. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer Magazine*, **11**, (4), (1978), pp. 34-41.

[5] DeMillo, R. A., and Orutt, A. J. (1991): Constraint-based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, **17**, (9), 900–910.

[6] Gill, N. S. Software Engineering. New Delhi: Khanna Book Publishing Co.(P) Ltd. 818732517-8. (2002), 425–430.

[7] Grigorjev, F., Lascano, N., Staude, J.: A Fault Seeding Experience. Argentina: *Motorola Global Software Group*.

[8] Glossary Working Party, *Standard Glossary of Terms used in Software Testing*. The Netherlands: Erick Van Veenendaal, Version 1.2, (2006).

[9] King, K. N. and Offutt, A. J. A Fortran Language System for Mutation-based Software Testing. *Software-practice and Experience*, **21**, (7), (1991), 685–718.

[10] Kamsties, E. and Lott, C. M. An Empirical Evaluation of Three Defect-Detection Techniques in *Proceedings of Fifth European Software Engineering Conference,* (1995).

[11] Kim, S., Clark, J. A. and Mcdermit, J. A. Class Mutation: Mutation Testing for Object Oriented Programs In Proceedings of the FMES (2000).

[12] Morell L. A Theory of Fault-based Testing. *IEEE Transactions on Software Engineering*, **16**, (8), (1990), 844–857.

[13] Ma, Y., Kwon, Y. and Offutt, J. Interclass Mutation Operators for Java. *In Proceedings of International Symp. On Software Reliability Engineering*, (2002), 352–363.

[14] Offut, A.J., Rothermel, G., Untch, R. H. and Zapf,C. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering Methodology*, **5**, (2), (1996), 99–118.

[15] Offutt, J., Ma, Y. and Kwon, Y. (2004): An Experimental Mutation System for Java. *ACM SIGSOFT Software Engineering Notes, ACM Press*, **29**, (5), 1–4.

[16] Patrick, H. Z., Hall, A. V. and May, John H. R. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, **29**, (4), (1997).

[17] Roper, M., Miller, J., Brooks, A. and Wood M. Towards the Experimental Evaluation of Software Testing Techniques. *Technical Report*, version 1.0., (1993)

[18] Shahriar, H. Mutation-Based Testing of Buffer Overflows, SQL Injections, and Format String Bugs. *Masters of Science Thesis*, School of Computing, Queen's University, Kingston, Ontario, Canada, (2008).

[19] Tuya, Suarez-Cabal, M. J., Riva, C. D. L. SQL Mutation A Tool to Generate Mutants of SQL Database Queries In *Second Workshop on Mutation Analysis*, (Nov 2006), 1.

[20] Tuya, Suarez-Cabal, M. J., Riva, C. D. L. Mutating Database Queries. *Information and Software Technology,* **49**, (4), (2007), 398-41.