Dynamic Test Input Generation for Database Applications to Achieve High Mutation Score

Tanmoy Sarkar
Ph.D. Advisors: Samik Basu, Johnny S. Wong
Department of Computer Science, Iowa State University
Email: {tanmoy, sbasu, wong}@iastate.edu

***Abstract*—Automatic generation of test cases for database applications has attracted researchers from both academia and industry. Typically in database application, the quality of test cases for the host language (e.g., Java) is evaluated on the basis of the number of lines, statements and blocks covered by the test cases, whereas, the quality of test cases for the embedded language (e.q., SQL) is evaluated using mutation testing. In mutation testing, several mutants or variants of the original SQL query are generated and the mutation score is calculated. It is a metrics which indicates the percentage of mutants that can be identified in terms of their results using the given test cases. Higher mutation score indicates higher quality for the test cases. We present a novel framework for test case generation which ensures high quality of the test cases not only in terms of coverage of code written in the host language, but also in terms of mutant detection of the queries written in the embedded language.**

I. Problem and Motivation

Automated test case generation techniques [1], [2], [3] have been proposed to minimize human effort in testing. Typically these techniques focus on structural (code, branch, block, etc.) coverage of the application program under test. Test suite achieving high structural coverage certainly increases the confidence on the quality of the test cases being used to validate correctness of (or to find bugs in) the application. However, coverage cannot be argued as a sole criterion for effective testing. Mutation testing [4] has been proven effective to assess the quality of test cases in terms of identifying common/typical programming faults. In mutation testing, the program being tested is modified slightly following pre-specified rules (that mimic common programming errors) and it is checked whether the existing test cases can differentiate between the original program and its mutant in terms of the outputs they generate. If the check is successful, the mutants are said to be *killed* by the test cases and the test cases are said to be of *good* quality; otherwise new test cases are considered for killing mutants. The quality of test cases is assessed using a metrics: *mutation score* which is the percentage of mutants among the total number of mutants killed by the test cases.

Driving Problem. With advances in the Internet technology and ubiquity of the Web, applications relying on data/information processing and retrieval from database form the majority of the applications being developed and used in the Software industry. Therefore, it is important that such applications are tested adequately before being deployed. A typical database application consists of two different programming language constructs: the control flow of the application depends on procedural languages, *host language* (e.g., Java); while the interaction between the application and the backend database depends on specialized *query languages* (e.q., SQL) that are constructed and embedded inside the host language. Automatically generating test cases and assessing their quality, therefore, pose an interesting and important challenge.

Problem Statement. How to automatically generate test cases for database applications such that: *test cases not only ensure high coverage of the control flow described in host language, but also allow for adequate testing of the embedded queries by attaining high mutation scores where mutants are generated from embedded queries.*

A. Motivating Example

```
1: procedure CHOOSECOFFEE(x, y)
2:     String q = " ";
3:     if x>10 then
4:         y++;
5:         if y≤ 2 then
6:             q = "SELECT cof_name FROM coffees
                                        WHERE price =" + y + ";";
7:         else
8:             q = "SELECT sup_id, cof_name
               FROM coffees c, suppliers s WHERE c.sup_id = s.sup_id
                                        AND c.price ≤"+ y + ";";
9:         end if
10:    end if
11:    if q != "" then executeQuery(q);
12:    end if
13: end procedure
```

We present here a simple database application to illustrate the problem we are addressing in this paper. Consider the pseudo-code in the above procedure CHOOSECOFFEE. It represents a typical database application; it takes as input two parameters x and y, creates a query string depending on the valuation of the parameters which guides the control path in the application. Assume that, the database table coffees contain the entries shown in Table I.

Pex [2], a dynamic symbolic execution (DSE) engine generates three test cases, (0, 0), (11, 0) and (11, 2) taking into consideration the branch conditions in the application program.

COF_NAME	SUP_ID	PRICE	TOTAL
Colombian	101	1	0
French_Roast	49	2	0
Espresso	150	10	0

Table I: coffees Table in the database

The first and the second values in the tuple represent the valuations of x and of y respectively. These test cases cover all branches present in the program. However, as the database is not taken into consideration for the test case generation, the test cases are unlikely to kill all mutants corresponding to the query being executed. For instance, the test case (11, 0) results in the execution of the query generated at line 6. The executed query

SELECT cof_name FROM coffees WHERE price = 1

generates the result `Colombian` using the *coffees* table. A mutant of this query

SELECT cof_name FROM coffees WHERE price ≤ 1

is generated by slightly modifying the "WHERE" condition in the query (mimicking typical programming errors). The result of the mutant is also `Colombian`. That is, if the programmer makes a typical error of using less-than-equal-to-operator in the "WHERE" condition instead of the intended equal-operator, then that error will go un-noticed if test case $(11, 0)$ is used. Note that, there exists a test case $(11, 1)$, which can distinguish both the mutants from the original query without compromising branch coverage. We will show that our framework successfully identifies such test cases automatically.

II. Related Work

Database application programs play a central role in operation of almost every modern organization. There are two main approaches to generate test cases for such applications:(1) generating database states from scratch [3], [5], (2) using existing database states [6]. Both of these approaches try to achieve a common goal, *high branch coverage*. Therefore automatic generation of test inputs has been regarded as the main issue in database application testing. Along with high branch coverage, assessing the goodness of test data has not been considered as a criteria while generating test inputs. Mutation testing has been proven to be a powerful method in this regard. It [7] is a fault-based testing approach and has been shown to be an effective indicator for the quality of test inputs [8]. Mutation testing was primarily developed for programming languages like Fortran, Ada [9]. For database application, SQL mutation operators have been developed [10] and coverage criteria of isolated SQL statements [11] have been defined separately. In our approach, we want to combine them together and guide test case generation technique using SQL mutation analysis, so that the test results achieve high coverage metrics like high quality and high structural coverage.

Test case generation for database applications primarily depends on the current database state. Before generating test inputs for database application, testers need to generate sufficient number of entries for the tables present in the database. Therefore, generating test database in an optimized/sufficient manner, for a given application, is a challenging problem which has concentrated some research efforts [12], [13], [14], [15]. Our work is orthogonal to these works and we focus on generating test cases (test inputs) achieving high coverage metrics (as mentioned before) of a database application, given an existing database state.

III. Approach and Uniqueness

Testing database applications has two important challenges:

- Generate test cases to validate correctness or find bugs by improving structural coverage (statement, block or branch coverage) of the program.
- Identify or generate sufficient and necessary database states which help test cases to improve coverage metrics.

We propose and develop a framework which comprehensively addresses the first challenge by incorporating mutation analysis in coverage based automatic test case generation. We show that the test cases generated in our framework are superior both in terms of coverage and in terms of mutation score. The framework also provides a roadmap to address the second challenge regarding sufficient and necessary database states and opens new avenues of research in the area of testing database application.

A. Approach Overview–ConSMutate: A New Framework for Database Application Testing

Figure 1 shows the details and the salient features of our framework, ConSMutate. It combines *Con*crete, *S*ymbolic execution and *Mutation* analysis to generate test cases. It has two main parts, *Application Branch Analyzer* and *Mutation Analyzer*. Application Branch Analyzer takes the program under test and sample database as inputs, and generates test cases and the corresponding path constraints. It uses Pex [2], a dynamic symbolic execution engine (other engines like concolic testing tool [1] can also be used), to generate test cases by carefully comparing the concrete and symbolic execution of the program. After exploring each path, mutation analyzer performs quality analysis using mutation testing. If the quality (mutation score) is low, mutation analyzer generates a new test case, for the same path, whose *quality* is likely to be high. The steps followed in our framework for generating test cases are as follows:

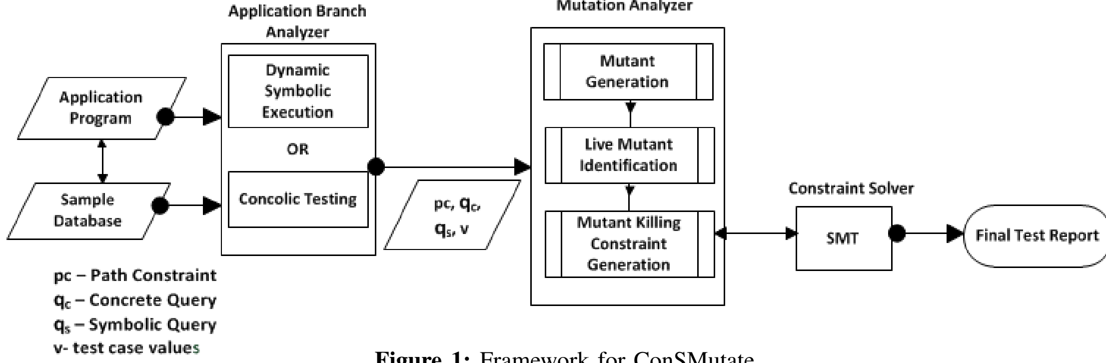

Figure 1: Framework for ConSMutate

Step 1: Generate Test Case and associated Path Constraints using Application Branch Analyzer. In the first step, the framework uses *Application Branch Analyzer* module to generate a test case value v and the associated path constraints. It results in a specific execution path constraint (say, pc) of the application, which in turn results in an execution of database query (if the path includes some query). The executed query is referred to as the concrete query q_c and the same without the concrete values is referred to as the symbolic query q_s. The path constraints refer to the conditions which must be satisfied for exploring the execution path in the application.

Going back to the example in Section I, in Step 1, Application Branch Analyzer (Pex in our case) generates a test case $v = (11, 0)$, i.e., $x = 11$ and $y = 0$. This results in an execution path with path constraints $pc = (x > 10) \wedge (y + 1 \leq 2)$. It also results in a symbolic query and a corresponding concrete query:

q_s:SELECT cof_name FROM coffees WHERE price = y_s
q_c:SELECT cof_name FROM coffees WHERE price = 1

y_s is the symbolic state of the program input y, which is y+1 in this case, at line 6(see program in I-A).

Step 2: Execute Mutation Analyzer. After exploring a path of the program under test, ConSMutate forwards pc, q_c, q_s and v to *Mutation Analyzer* to evaluate the quality of the generated test case in terms of mutation score.

Step 2.1: Generate Mutant Queries. In Mutation Analyzer, the obtained q_c in step one is mutated to generate several mutants. The mutations are done using pre-specified mutation functions in the *Mutant Generation* module.

We have identified six rules which we call *sufficient set of SQL mutation generation rules* [10], [16] to identify logical errors present in the *WHERE* and *HAVING* clauses. The first three columns of the Table II illustrate some of the rule names, rule conditions and the descriptions. For instance, one of the mutants of the above query q_s is

q_m: SELECT cof_name FROM coffees
WHERE price $\leq y_s$

α is "=" (equality relational operator) and β is "$\leq$" (less-than-equal-to relational operator) as per the rule in the first row, second and third columns of Table II.

Mutation Rule	Original	Mutant	Mutant Killing Constraint
Relational Operator Replacement (*ROR*) $\alpha, \beta \in$ ROR and $\alpha \neq \beta$	$C_1\ \alpha\ C_2$	$C_1\ \beta\ C_2$	$((C_1\ \alpha\ C_2) \wedge \neg(C_1\ \beta\ C_2))$ $\|\|$ $(\neg(C_1\ \alpha\ C_2) \wedge (C_1\ \beta\ C_2)))$
Logical Connector Operator Replacement (*LCR*) $\alpha, \beta \in$ LOR and $\alpha \neq \beta$	$C_1\ \alpha\ C_2$	$C_1\ \beta\ C_2$	$(C_1\ \alpha\ C_2) \neq (C_1\ \beta\ C_2)$
Arithmetic Operator Replacement (*AOR*) $\alpha, \beta \in$ AOR and $\alpha \neq \beta$	$C_1\ \alpha\ C_2$	$C_1\ \beta\ C_2$	$(C_1\ \alpha\ C_2) \neq (C_1\ \beta\ C_2)$

Table II: Partial Table for Mutant Generation and Mutant Killing Constraints Generation Rules

Step 2.2: Identify Live Mutants. Using the test case under consideration, the live mutants are identified. Live mutants are the ones whose results do not differ from that of the concrete query in the context of the given database table.

The above mutant q_m is live under the test case $v = (11, 0)$ as q_c and q_m produces the same result for the given database table (Table I).

Step 2.3: Generate Mutant Killing Contraints. A new set of constraints, θ is generated in *Mutant Killing Constraint Generation* module from the

- the symbolic query q_s and its concrete version q_c
- the live mutants (q_m's) computed in the previous step
- the path constraints of the execution (pc) obtained in the step 1

θ includes conditions on the inputs to the application. Due to expensiveness of mutation testing, we adopt the concept of weak mutation testing [17]. Therefore, the generated constraint will guide to weakly kill the mutant, i.e., the

generated test cases do not guarantee to kill the live mutants, but improve the probability to kill them.

The mutant killing constraint θ is generated as follows. The mutant q_m is live because the WHERE clause $price = y_s$ and $price \leq y_s$ do not generate two different result-sets, when the input y is set to 0 to the symbolic variable y_s(since y_s = $y+1$). In other words, as price is set to the variable $y+1$ (where y is equal to 0), q_m is live because $y+1=1$ and $y+1 \leq 1$ do not produce two different result-sets. In order to generate a different value of y which is likely to kill the mutant q_m, we need to choose a value for y such that $(y+1=1 \wedge y+1 \not\leq 1) \vee (y+1 \neq 1 \wedge y+1 \leq 1)$. The last column of the Table II demonstrates the general rules for generating these constraints. In our example, α is "=", β is $\leq$, C_1 is $y+1$ and C_2 is 1.

This constraint, in conjunction with the path constraint (since the new test case should satisfy the executed path constraint), results in θ, the constraint which when satisfied is likely to generate a test case that can kill the mutant q_1.

$$\theta: \quad (x>10) \wedge (y+1 \leq 2) \wedge \ [(y+1=1 \wedge y+1 \not\leq 1) \\ \vee (y+1 \neq 1 \wedge y+1 \leq 1)]$$

Step 2.4: Find Satisfiable Assignment for Constraint θ. The constraint θ is checked for satisfiability to generate a new test case. If θ is satisfied then certain valuations of the inputs to the application are identified, which is the new test case v'. This new test case v' is guaranteed to explore the same execution path as explored due to test case v (see Step 1). Furthermore, some mutants that were left "live" by v are *likely to be* "killed" by v'. Therefore, it is necessary to check whether v' indeed kills the live mutants; if not, SMT solver is used again to generate new satisfiable assignment for θ, which results in a new test case v''. This iteration is terminated after certain pre-specified times (e.g., 10) or after all live mutants are killed (whichever happens earlier).

For instance, if the SMT solver generates a satisfiable assignment $x=11, y=1$ for the mutant killing constraint θ (see above), then the new test case $v'=(11,1)$ successfully kills the live mutant q_m as shown in Table III.

Query	Concrete Query	Result
q_c	SELECT cof_name FROM coffees WHERE price = 2	French_Roast
Mutant q_m	SELECT cof_name FROM coffees WHERE price $\leq$ 2	Colombian, French_Roast

Table III: Mutants and Results for test case (11, 1)

Finally **Step 1** is iterated to generate new test cases that explore different execution paths of the program. This iteration continues until all possible branches are covered following the method used by Pex.

B. Uniqueness

Several features of ConSMutate framework sets apart our approach and attributes to its uniqueness in solving a very important problem of testing database applications.

Quality. Our approach combines coverage constraints and mutation analysis to automatically generate high quality test cases for database applications.

Applicability. Being based on constraint-satisfaction, our approach does not rely on the usage of any specific application language or query language. In other words, it is applicable to any database applications.

Extensibility. Our approach is implemented in a highly modular fashion which makes it possible to include different (and newly developed) techniques in plug-and-play basis for generating path and mutation killing constraints. This makes our approach and framework relevant and applicable even when new languages and technologies are developed for realizing and testing database applications.

IV. Results and Contributions

A. Evaluation Criteria

We evaluate the benefits of our approach from the following two perspectives:

1) What is the percentage increase in code coverage by the test cases generated by Pex compared to the test cases generated by ConSMutate in testing database applications?
2) What is the percentage increase in mutation score of test cases generated by Pex compared to the ones generated by ConSMutate in testing database applications?

To set up the evaluation, we choose methods from two database applications that have parameterized embedded SQL queries and the program inputs are directly or indirectly used in those queries. We first run Pex, which is a dynamic symbolic execution based unit testing tool for .NET applications, to generate test cases (different valuations for program inputs) for those methods and then record the mutation score and code coverage percentage achieved by them. Next we apply ConSMutate to generate test cases for the same methods and record the corresponding mutation score and code coverage statistics. The experiments are conducted on a PC with 2GHz Intel Pentium CPU and 2GB memory running Windows XP operating system.

B. Evaluation test-bed

Our empirical evaluations are performed on two open source database applications: *UnixUsage*[1] and *RiskIt*[2]. UnixUsage is an application which interacts with a database and the queries are written against the database to display information about how users (students) who are registered in different courses, interact with the Unix systems using different commands. The database contains 8 tables, 31

[1]http://sourceforge.net/projects/se549unixusage

[2]https://riskitinsurance.svn.sourceforge.net

Method(s)	Parameter Name	Block Covered by Pex	Block Covered by ConSMutate	SQL Mutation Score of test results by Pex	SQL Mutation Score of test results by ConSMutate
courseIdExists	courseId	95.92%	100%	90.90%	100%
getCourseNameByID	courseId	100%	100%	72.72%	**100%**
getCourseIDByName	courseName	96%	100%	45.45%	**100%**
courseNameExists	courseName	95.9%	100%	57.14%	**100%**
doesUserIdExist	userId	90%	90%	57.14%	**85.71%**
isDepartmentIdValid	departmentId	90%	90%	90%	100%
isRaceIdValid	raceId	90%	90%	54.54%	**90.90%**
getDeptInfo	deptId	84.90%	90.56%	57.14%	**100%**
deptIDExists	deptId	92.59%	92.59%	72.72%	**95.45%**

Table IV: UnixUsage Evaluation

Method(s)	Parameter Name	Block Covered by Pex	Block Covered by ConSMutate	SQL Mutation Score of test results by Pex	SQL Mutation Score of test results by ConSMutate
getValues	ssn	74.24%	**92.42%**	28.57%	**92.8%**
filterZipcode	zip	75.34%	86.30%	57.14%	**85.71%**
filterEducation	edu	50%	**100%**	20.00%	**76.66%**
filterMaritalStatus	status	95.24%	95.24%	85.71%	**100%**

Table V: RiskIt Evaluation

attributes, and has over a quarter million records. RiskIt is an insurance quote application which makes estimation based on users' personal information, such as zipcode. It has a database containing 13 tables, 57 attributes, and over 1.2 million records. Both applications were written in Java with backend Derby. To test them in the Pex environment, we convert the Java source code into C# code using a tool called Java2CSharpTranslator[3]. Since Derby is a special database management system for Java and does not adequately support C#, we retrieve all the database records from Derby and populate them into Microsoft Access 2010. We also manually translate those original JDBC drivers and connection settings into C# code.

C. Summary of Evaluation

Tables IV and V show the results of our evaluation. Columns 1 and 2 for each of these tables show the methods under test and the input parameters to those methods respectively.

Evaluation Criteria 1: Coverage benefit. Columns 3 and 4 in both tables demonstrate the block coverage achieved by Pex and ConSMutate respectively (more than 10% gains in coverage are presented in bold). They show that, given a database state, Pex cannot generate sufficient program inputs to achieve higher code coverage especially when program inputs are directly or indirectly involved in embedded SQL statements. ConSMutate does not suffer from this drawback as it considers database states and the results of generated queries and their execution results.

Evaluation Criteria 2: Mutation Score benefit. Columns 5 and 6 in both tables display the mutation score achieved by Pex and ConSMutate respectively (more than 10% gains in mutation score are presented in bold). The mutation score of test cases generated by ConSMutate is always higher than the mutation score of test cases generated by Pex; in some cases the increase in the mutation score is *four*-fold.

The mutation scores achieved by ConSMutate are sometimes less than 100% because the database tables provided may not be sufficient to kill all the mutants.

D. Contributions

Our approach for test case generation has several significant impacts in the domain of Software Engineering.

1) To the best of our knowledge, this is the first approach that combines coverage analysis and mutation analysis in automatic test case generation for database applications. Our initial experiments show the effectiveness and practical applicability of the approach.
2) The framework ConSMutate is generic and therefore, new technologies of coverage-based techniques and mutation generations can be easily incorporated and evaluated in the framework.
3) The framework relies on identifying important control-path constraints of the application and the constraints for killing mutants. These constraints provide valuable insights which help in obtaining the necessary and sufficient database states for generating test cases with pre-specified (even 100%) coverage and mutation scores. This leads to new research avenues involving concolic testing, model checking and constraint solving to investigate the adequacy and limitations of database states for generating high quality test cases.

[3]http://sourceforge.net/projects/j2cstranslator/

REFERENCES

[1] K. Sen, "Concolic testing," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, (New York, NY, USA), pp. 571–572, ACM, 2007.

[2] N. Tillmann and J. De Halleux, "Pex: white box test generation for .net," in *Proceedings of the 2nd international conference on Tests and proofs*, TAP'08, (Berlin, Heidelberg), pp. 134–153, Springer-Verlag, 2008.

[3] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, (New York, NY, USA), pp. 151–162, ACM, 2007.

[4] A. J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction procedure for test data generation," *Softw. Pract. Exper.*, vol. 29, pp. 167–193, Feb. 1999.

[5] K. Taneja, Y. Zhang, and T. Xie, "Moda: automated test generation for database applications via mock objects," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, (New York, NY, USA), pp. 289–292, ACM, 2010.

[6] C. Li and C. Csallner, "Dynamic symbolic database application testing," in *Proceedings of the Third International Workshop on Testing Database Systems*, DBTest '10, (New York, NY, USA), pp. 7:1–7:6, ACM, 2010.

[7] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 900–910, Sept. 1991.

[8] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *Proceedings of the 27th international conference on Software engineering*, ICSE '05, (New York, NY, USA), pp. 402–411, ACM, 2005.

[9] J. Voas, "Software fault injection: growing 'safer' systems," in *Aerospace Conference, 1997. Proceedings., IEEE*, vol. 2, pp. 551 –561 vol.2, feb 1997.

[10] J. Tuya, M. J. Suárez-Cabal, and C. d. la Riva, "Mutating database queries," *Inf. Softw. Technol.*, vol. 49, pp. 398–417, Apr. 2007.

[11] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Full predicate coverage for testing sql database queries," *Softw. Test. Verif. Reliab.*, vol. 20, pp. 237–288, Sept. 2010.

[12] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker, "An agenda for testing relational database applications: Research articles," *Softw. Test. Verif. Reliab.*, vol. 14, pp. 17–44, Mar. 2004.

[13] D. Chays, J. Shahid, and P. G. Frankl, "Query-based test generation for database applications," in *Proceedings of the 1st international workshop on Testing database systems*, DBTest '08, (New York, NY, USA), pp. 6:1–6:6, ACM, 2008.

[14] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid, "Query-aware test generation using a relational constraint solver," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, (Washington, DC, USA), pp. 238–247, IEEE Computer Society, 2008.

[15] C. Binnig, D. Kossmann, and E. Lo, "Reverse query processing," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pp. 506 –515, april 2007.

[16] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of the 15th international conference on Software Engineering*, ICSE '93, (Los Alamitos, CA, USA), pp. 100–107, IEEE Computer Society Press, 1993.

[17] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Softw. Eng.*, vol. 8, pp. 371–379, July 1982.