Automatic Test Generation for Mutation Testing on Database Applications

Kai Pan Xintao Wu
University of North Carolina at Charlotte
Charlotte, North Carolina, USA
{kpan, xwu}@uncc.edu

Tao Xie
North Carolina State University
Raleigh, North Carolina, USA
xie@csc.ncsu.edu

***Abstract*—To assure high quality of database applications, testing database applications remains the most popularly used approach. In testing database applications, tests consist of both program inputs and database states. Assessing the adequacy of tests allows targeted generation of new tests for improving their adequacy (e.g., fault-detection capabilities). Comparing to code coverage criteria, mutation testing has been a stronger criterion for assessing the adequacy of tests. Mutation testing would produce a set of mutants (each being the software under test systematically seeded with a small fault) and then measure how high percentage of these mutants are killed (i.e., detected) by the tests under assessment. However, existing test-generation approaches for database applications do not provide sufficient support for killing mutants in database applications (in either program code or its embedded or resulted SQL queries). To address such issues, in this paper, we propose an approach called *MutaGen* that conducts test generation for mutation testing on database applications. In our approach, we first apply an existing approach that correlates various constraints within a database application through constructing synthesized database interactions and transforming the constraints from SQL queries into normal program code. Based on the transformed code, we generate program-code mutants and SQL-query mutants, and then derive and incorporate query-mutant-killing constraints into the transformed code. Then, we generate tests to satisfy query-mutant-killing constraints. Evaluation results show that *MutaGen* can effectively kill mutants in database applications, and *MutaGen* outperforms existing test-generation approaches for database applications in terms of strong mutant killing.**

I. Introduction

To assure high quality of database applications, testing database applications remains the most popularly used approach. In testing database applications, tests consist of both program inputs and database states. Assessing the adequacy of tests allows targeted generation of new tests for improving their adequacy (e.g., fault-detection capabilities). In particular, assessing the adequacy of tests could indicate the weakness of tests in terms of satisfying the target testing requirements. Comparing to code coverage criteria (a popular type of testing requirements), mutation testing has been a stronger criterion for assessing the adequacy of tests. Mutation testing would produce a set of mutants (each being the software under test systematically seeded with a small fault) and then measure how high percentage of these mutants are killed (i.e., detected) by the tests under assessment. Other than traditional mutation testing where mutants exist in normal program code, Tuya et al. [13], [14] proposed a set of mutation operators for SQL queries and developed a tool called SQLMutation that implements these mutation operators to generate SQL-query mutants. To assess the adequacy of tests for Java database applications, Zhou and Frankl [18] developed a tool called JDAMA based on the mutation operators for SQL queries [14].

To kill generated mutants, test generation for mutation testing has been addressed [1], [17]. However, for mutation testing on database applications, tests consist of both program inputs and database states. Thus, these approaches become inapplicable for database applications because sufficient and supportive back-end database states are required for generated tests. Focusing on test generation for testing database applications, some recent approaches [2], [8], [11] have been proposed to automatically generate database states and program inputs to achieve various testing requirements such as high code coverage. However, these approaches do not consider mutation testing as the main goal and cannot provide effective support for killing mutants in database applications.

For a database application, a mutant may occur in either normal program code or SQL queries. Generating appropriate program inputs and sufficient database states to kill a mutant requires collecting and satisfying constraints for killing that mutant. Typically, within a database application, a mutant in normal program code can affect the query-construction constraints (where constraints come from the sub-paths explored before the query execution) and query-result-manipulation constraints (where constraints come from the sub-paths explored for iterating through the query result), while a mutant in SQL queries can affect the query constraints (where constraints come from conditions in a query's WHERE clause). Test generation by applying a constraint solver on the collected constraints faces great challenges because a constraint solver can deal with *program-execution constraints* (e.g., query-construction constraints and query-result-manipulation constraints) but cannot directly handle *environment constraints* (e.g., query constraints).

Existing test-generation approaches [2], [11] for database applications choose to consider *program-execution constraints* and *environment constraints* separately. Thus, when applying existing approaches [2], [11] for mutation testing on database applications, the design decision of these approaches requires a whole constraint system for each mutant's killing, making the whole process costly or even infeasible [9]. On the other

hand, although a recent approach called PexMutator [17] incorporates all the mutant-killing constraints into the program under test, the approach still cannot directly correlate *program-execution constraints* and *environment constraints* for database applications, thus not being able to generate sufficient database states.

To address these issues, in this paper, we propose a new approach called *MutaGen* for killing mutants in database applications based on a newly developed framework [9] called SynDB. The SynDB framework is built on Dynamic Symbolic Execution (DSE) [3], [12] and correlates program-execution constraints and environment constraints in a database application. It constructs synthesized database interactions and transforms the original program under test into another form that the synthesized database interactions can operate on. Meanwhile, a synthesized object is constructed to replace the physical database state and the query constraints are transformed into normal program code. The framework focuses on generating program inputs and database states to achieve high program code coverage. In *MutaGen*, we leverage SynDB as a supporting mechanism for mutation testing on database applications.

To generate mutants that occur in the program code, we apply an existing code-mutation tool [17] on the code transformed with the SynDB framework. To generate SQL-query mutants, we apply an existing SQL-query-mutation tool [13] to generate SQL-query mutants at query-issuing points. We then derive query-mutant-killing constraints by considering both the original query and its mutants. We finally incorporate the derived constraints into the transformed code. Specifically, solving these query-mutant-killing constraints helps produce a database state on which running the original query and its mutants can cause different query results, thus killing the corresponding SQL-query mutants. The transformed code is able to guide DSE to collect constraints for both program inputs and database states. By applying a constraint solver on the collected constraints, we generate effective tests for killing both program-code mutants and SQL-query mutants.

II. Background

In this section, we present some technical background about mutation testing. We leave the discussion of the SynDB framework in Section III-B.

Mutation testing is a fault-based software-testing technique that has been intensively studied for evaluating the adequacy of tests. The original program under test is mutated into a set of new programs, called *mutants*, caused by syntactic changes following a set of rules. The mutants are *(strongly) killed* if running the mutants against given tests produces different results than the results of the original program. Killing more mutants reflects better adequacy of the tests under assessment.

However, automatically producing tests that can kill mutants could be very time-consuming and even intractable [1], because there can be a large number of mutants produced for a short program. To deal with the expensiveness of mutation testing, Howden et al. [4] proposed *weak mutation testing* that focuses on intermediate results or outputs from components of the program under test. Instead of checking mutants after the execution of the entire program, weak mutation testing checks the mutants immediately after the mutated components. Researchers also selected a subset of mutation operators [6] or mutants [16] to reduce time or space resources exhausted by a large number of mutants. For example, Offutt et al. [6] reported that 5 mutation operators could perform as effectively as all 22 mutation operators.

Mutation testing was also applied to detect faults in SQL queries. Tuya et al. [14] proposed a set of mutation operators and developed SQLMutation [13] that implements this set of mutation operators to generate SQL-query mutants. These mutation operators are organized into four categories:
SC - SQL clause mutation operators: mutation of the most distinctive features of SQL (e.g., clauses, aggregate functions).
OR - Operator replacement mutation operators: extension of the expression modification operators.
NL - NULL mutation operators: mutation related to incorrect treatment of NULL values.
IR - Identifier replacement mutation operators: replacement of operands and operators (e.g., replacement of columns or constants).

In our approach, for database applications, SQL queries are considered as components of the program under test. Thus, applying *weak mutation testing* by seeding faults [14] to the queries can reflect the adequacy of the associated test database states.

III. Approach

In this section, we present details of the *MutaGen* approach. We first give a motivating example to illustrate the necessity of generating sufficient database states for mutation testing on database applications.

A. A Motivating Example

The example code in Figure 1 is a portion of C# code from a database application that calculates some statistics related to customers' mortgages. The schema of the associated database is shown in Table I. The method `calcStat` sets up database connection (Lines 03-05), constructs a query (Line 06), and executes the query (Lines 07-08). The query contains two program variables: a local variable `zip` and a program-input parameter `inputAge`. The returned records are then iterated (Lines 09-14). For each record, a variable `diff` is calculated from the values of the columns `C.income` and `M.balance`. If `diff` is greater than `50000`, a counter variable `count` is increased (Line 14). The method finally returns the value of `count` (Line 15).

To test the preceding method in Figure 1 for achieving high code coverage, existing test-generation approaches [2], [11] can generate both program inputs and database states to cover feasible paths. For example, the generated values for input `inputAge` and corresponding database records shown in Table II could achieve full code coverage: a default value `inputAge = 0` and an empty database state covers the path where `Line`

```
01:public int calcStat(int inputAge) {
02:  int zip = 28223, count = 0;
03:  SqlConnection sc = new SqlConnection();
04:  sc.ConnectionString = "..";
05:  sc.Open();
06:  string query = "SELECT C.SSN, C.income,"
   +" M.balance FROM customer C, mortgage M"
   +" WHERE C.age='" + inputAge +"' AND"
   +" C.zipcode='"+ zip + "' AND C.SSN = M.SSN";
07:  SqlCommand cmd = new SqlCommand(query, sc);
08:  SqlDataReader results = cmd.ExecuteReader();
09:  while (results.Read()){
10:     int income = results.GetInt(1);
11:     int balance = results.GetInt(2);
12:     int diff = income - balance;
13:     if (diff > 50000){
14:        count++;}}
15:  return count;}
```

Fig. 1. Example code from a database application in C#

TABLE I
DATABASE SCHEMA

customer table			mortgage table		
Attribute	Type	Constraint	Attribute	Type	Constraint
SSN	Int	Primary Key	SSN	Int	Primary Key
name	String	Not null			Foreign Key
gender	String	$\in$ {F, M}	year	Int	$\in$ {10, 20, 30}
zipcode	Int	[00001, 99999]			
age	Int	[0, 100]	balance	Int	[2000, Max)
income	Int	[10000, Max)			

`09 = false`; `inputAge = 30` and the record whose column `SSN = 001` covers the path where `Line 09 = true, Line 13 = false`; `inputAge = 40` and the record whose column `SSN = 002` covers the path where `Line 09 = true, Line 13 = true`.

However, in terms of mutation testing, tests in Table II are not sufficient. Killing mutants in database applications requires more program inputs and multiple database records so that executing the program and its mutants with these inputs against the database could produce different results. For example, in Figure 1, for a mutant in Line 13 where `diff > 50000` is mutated to `diff >= 50000`, none of the values for `inputAge` in Table II could kill this mutant because the original program's output and the mutant's output are the same. Similarly, for a mutant of the query in Line 06 where the condition `C.age = 'inputAge'` is mutated to `C.age <= 'inputAge'`, none of the values for `inputAge` could kill this SQL-query mutant[1]. Hence, for database applications, achieving mutant killing requires both effective program inputs and sufficient database states.

B. SynDB Framework Revisited

MutaGen is based on the newly developed SynDB framework [9]. SynDB transforms the original program under test into another form to correlate program-execution constraints and environment constraints. It constructs new synthesized database interactions to replace the original ones for the program under test. Figure 2 shows the transformed code of the example code in Figure 1.

[1]Indeed, for `inputAge = 40`, the mutant `C.age <= 'inputAge'` is weakly killed because executions of the original query and this mutant on Table II produce different result sets.

TABLE II
PROGRAM INPUTS AND DATABASE STATES TO COVER PATHS FOR PROGRAM CODE IN FIGURE 1

input	customer table						mortgage table		
inputAge	SSN	zipcode	name	gender	age	income	SSN	year	balance
30	001	28223	Alice	F	30	70000	001	20	30000
40	002	28223	Bob	M	40	90000	002	20	30000

```
01:public int calcStat(int inputAge,
                       DatabaseState dbState) {
02:  int zip = 28223, count = 0;
03:  SynSqlConnection sc = new SynSqlConnection(dbState);
04:  sc.ConnectionString = "..";
05:  sc.Open();
06:  string query = "SELECT C.SSN, C.income,"
   +" M.balance FROM customer C, mortgage M"
   +" WHERE C.age='" + inputAge + "' AND"
   +" C.zipcode='" + zip + "' AND C.SSN = M.SSN";
07:  SynSqlCommand cmd = new SynSqlCommand(query, sc);
08:  SynSqlDataReader results = cmd.ExecuteReader();
09:  while (results.Read()){
10:     int income = results.GetInt(1);
11:     int balance = results.GetInt(2);
12:     int diff = income - balance;
13:     if (diff > 50000){
14:        count++;}}
15:  return count;}
```

Fig. 2. Transformed code for the example code in Figure 1

SynDB identifies and replaces the original database interactions with renamed API methods (e.g., by adding "Syn" before each method name). SynDB then constructs a synthesized database state to replace the physical one according to the given database schema. It defines tables and attributes within the synthesized database state and uses auxiliary methods to enforce schema constraints. It treats the synthesized database state as an object and adds it as an input to the program under test. For example, according to the schema in Table I, SynDB constructs a synthesized database state shown in Figure 3. In Figure 2, SynDB adds a new input `dbState` with the type `DatabaseState` to the program. It then passes the synthesized database state within the synthesized database interactions. For each database-interacting interface (e.g., database connection, query construction, and query execution), SynDB adds a new field to represent the synthesized database state and uses auxiliary methods to pass it. The synthesized database interfaces help implement basic interacting functionalities with the synthesized database state. For example, the interface `SynSqlCommand` integrates a query to be executed and uses its method `ExecuteReader()` to implement database operations. SynDB incorporates the query constraints as program-execution constraints in normal program code by parsing the symbolic query and transforming the constraints from conditions in the WHERE clause into normal program code.

Then SynDB applies DSE [3], [12] on the transformed code to collect constraints of the associated database and generate tests. DSE is an automatic test-generation technique that extends traditional symbolic execution [5] by executing

```
public class customerTable {
    public class customer {//define attributes;}
    public List<customer> customerRecords;
    public void checkConstraints() {
        /*method for checking schema constraints*/;}}
public class mortgageTable {
    public class mortgage {//define attributes;}
    public List<mortgage> mortgageRecords;
    public void checkConstraints() {
        /*method for checking schema constraints*/;}}

public class DatabaseState {
  public customerTable customerT = new customerTable( );
  public mortgageTable mortgageT = new mortgageTable( );
  public void checkConstraints(){
        /*check constraints for each table*/;}}
```

Fig. 3. Synthesized database state

a program under test with concrete inputs and collecting concrete and symbolic information at runtime [3], [12]. In the SynDB framework, DSE's exploration on the transformed code is guided to track the synthesized database state symbolically through synthesized database interactions and collect constraints of the synthesized database state when exploring path conditions from query constraints.

SynDB [9] mainly focused on generating tests to achieve high program code coverage. In *MutaGen*, we leverage SynDB as a supporting mechanism for mutation testing.

C. Mutant Killing

Based on the transformed code produced by the SynDB framework [9], *MutaGen* conducts mutant killing for database applications from two aspects: killing mutants in original normal program code and killing SQL-query mutants. For the transformed code, *MutaGen* seeds code-mutant-killing constraints by applying an existing mutant-generation tool [17]. To kill SQL-query mutants, *MutaGen* invokes a query-mutant-generation tool [13] to generate SQL-query mutants at query-issuing points, derives query-mutant-killing constraints, and inserts the constraints into the transformed code. Thus, applying a DSE engine on the modified transformed code to satisfy the weak-mutant-killing constraints is able to generate both effective program inputs and sufficient database states to weakly kill program-code mutants and SQL-query mutants.

1) Killing Program-Code Mutants: Mutants in original program code may affect test generation of database states because variables in the mutated statements may be data-dependant on the database attributes of the returned query result. For example, in Figure 1, the value of variable `diff` in Line 13 is derived from the values of database attributes `C.income` and `M.balance`. Hence, mutants of the statement in Line 13 would cause changes to the constraints for generating database states.

MutaGen applies a tool called PexMutator [17] on the transformed code of the original program under test. PexMutator is a mutant-generation tool that constructs weak-mutant-killing constraints to guide test generation.

Note that in the transformed code, program-execution constraints affected by mutants of original program code have been correlated with query constraints. Thus, satisfying these generated weak-mutant-killing constraints provides sufficient constraints for generating database states to help kill corresponding program-code mutants. In *MutaGen*, applying PexMutator on the transformed code would not affect the implementations of the constructed synthesized database interactions, because PexMutator focuses on only the specific program (i.e., the program under test) indicated by *MutaGen*. After introducing the weak-mutant-killing constraints, we apply a DSE engine (e.g., Pex [12], [15] for .NET) on the transformed code to generate database records.

```
12:     ...
13a:    if(((diff>50000) && !(diff>=50000)) ||
           (!(diff>50000) && (diff>=50000)));
           //to weakly kill the mutant diff>=50000
13b:    if(((diff>50000) && !(diff==50000)) ||
           (!(diff>50000) && (diff==50000)));
           //to weakly kill the mutant diff==50000
13c:    if(((diff>50000) && !(diff!=50000)) ||
           (!(diff>50000) && (diff!=50000)));
           //to weakly kill the mutant diff!=50000
        ...
13:     if (diff > 50000){
14:        count++;}}
15:  return count;}
```

Fig. 4. Code resulted from applying PexMutator on the transformed code in Figure 2

TABLE III
GENERATED TESTS FOR PROGRAM CODE IN FIGURE 2 TO WEAKLY KILL THE THREE MUTANTS SHOWN IN FIGURE 4

int	DatabaseState dbState								
inputAge	dbState.Customer						dbState.Mortgage		
	SSN	name	gender	zipcode	age	income	SSN	year	balance
50	003	AAA	F	28223	50	100000	003	30	50000
50	004	BBB	M	28223	50	100000	004	30	40000
50	005	CCC	M	28223	50	100000	005	30	60000

Figure 4 shows the code resulted from applying PexMutator on the transformed code shown in Figure 2. At the mutation point in Line 13, the generated weak-mutant-killing constraints (we list three of them) for the statement `if(diff>50000)` are inserted before Line 13. The value of variable `diff` is calculated from the values of attributes `C.income` and `M.balance`. Then, applying a DSE engine on the modified transformed code generates appropriate values for program inputs `inputAge` and `dbState` to cover the true branches of Lines 13a, 13b, and 13c, weakly killing the corresponding three mutants `diff>=50000`, `diff==50000`, and `diff!=50000`. We show tests to kill the three mutants in Table III.

Note that although PexMutator provides a general way of inserting weak-mutant-killing constraints into the program code, combining PexMutator with existing test-generation approaches [2], [11] cannot help directly generate tests to kill program-code mutants in database applications. Program-execution constraints and query constraints are still not correlated, causing that a whole constraint system is needed for each mutant's killing.

2) Killing SQL-Query Mutants: Mutants occurring in SQL queries directly affect constraints for generating database states. To weakly kill a SQL-query mutant, *MutaGen* generates database records to expose the difference between the original query and the mutant so that their executions produce different results.

In *MutaGen*, the transformed code has incorporated the query constraints into normal program code. We first identify query-issuing points by finding corresponding method signatures (e.g., `SynSqlCommand.ExecuteReader()`). Then, at each query-issuing point, we get the symbolic query and invoke the tool SQLMutation [13] to generate its mutants. SQLMutation automatically generates SQL-query mutants (providing each mutant's form, type, and generation rule) based on a set of mutation operators [14] for SQL queries. As mentioned in Section II, the mutation operators are organized into four categories of which the SC operators mainly focus on the main clauses (e.g., SELECT clause) and the other operators (OR, NL, and IR) focus on the conditions in the WHERE clause. For example, one of the mutants generated by the OR operators using SQLMutation for the query in Figure 2 is shown in Figure 5, where the condition `C.age = 'inputAge'` is mutated to `C.age >= 'inputAge'`.

Next, we derive query-mutant-killing constraints based on the original query and its mutants, and insert these constraints into the transformed code. To avoid causing syntactic errors, in the transformed code, we insert these constraints before the original query. Algorithm 1 gives details of how to derive the query-mutant-killing constraints. The algorithm mainly deals with mutants generated by OR, NL, and IR operators (e.g., mutating operators or column names in the WHERE clause). In Algorithm 1, the inputs consist of a constructed synthesized database state $SynDB$ and a symbolic query Q, and the output is a set of program statements that contain conditions whose exploration helps derive constraints for killing mutants of the given query. In Algorithm 1, we construct an empty statement set S (Line 1) and a SQL-query mutant set Q_m by calling $SQLMutation(Q)$ (Line 2). We retrieve Q's WHERE clause $s1$ using a SQL parser (Line 4). In Lines 5-17, for each mutant q in Q_m, if q is generated by the mutation operators OR, NL, or IR, we retrieve its WHERE clause $s2$ and construct a query-mutant-killing constraint s = (!$s1$ AND $s2$) OR ($s1$ AND !$s2$). Note that if a record r satisfies conditions in s, then r can satisfy only either $s1$ or $s2$, causing different execution results when executing Q and q against r. We then check the expressions in s and replace the columns in s with their corresponding names from the constructed synthesized database state $SynDB$. We add the query-mutant-killing constraint s to the set S (Line 15). After dealing with all the mutants in Q_m, the algorithm finally returns the set S (Line 18). For example, to weakly kill the mutant shown in the upper part of Figure 5, the constructed query-mutant-killing constraints based on the query's WHERE clause are shown in the lower part of Figure 5.

To deal with SQL-query mutants generated by the SC operators, we mainly focus on cardinality constraints as killing

Algorithm 1 *QMutantGen*: Generate query-mutant-killing constraints

Input: Synthesized database state $SynDB$, a symbolic query Q
Output: A set of program statements S

1: Statement set $S = \varnothing$;
2: Query mutant set $Q_m = SQLMutation(Q)$;
3: Mutation operator set OP = {OR, NL, IR};
4: String $s1 = Q.whereClause$;
5: **for** each query q in Q_m **do**
6: **if** $q.type \in OP$ **then**
7: String $s2 = q.whereClause$;
8: String s = (!$s1$ AND $s2$) OR ($s1$ AND !$s2$);
9: **for** each expression e in s **do**
10: **for** each column c in e **do**
11: Variable $v = findColumn(c, SynDB)$;
12: $Replace(c, v)$;
13: **end for**
14: **end for**
15: $S = S \bigcup s$;
16: **end if**
17: **end for**
18: **return** S;

```
A mutant generated by OR operators using SQLMutation:
OR(query) = "SELECT C.SSN, C.income, M.balance
          FROM customer C, mortgage M
          WHERE C.age >= 'inputAge'
          AND C.zipcode = 'zip' AND C.SSN = M.SSN

Constructed query-mutant-killing constraints:
((SynDB.customerTable.age == 'inputAge' AND
SynDB.customerTable.zipcode == 'zip'
AND SynDB.customerTable.SSN == SynDB.mortgageTable.SSN) &&
!(SynDB.customerTable.age >= 'inputAge' AND
SynDB.customerTable.zipcode == 'zip'
AND SynDB.customerTable.SSN == SynDB.mortgageTable.SSN))||
((!(SynDB.customerTable.age == 'inputAge' AND
SynDB.customerTable.zipcode == 'zip'
AND SynDB.customerTable.SSN == SynDB.mortgageTable.SSN)) &&
(SynDB.customerTable.age >= 'inputAge' AND
SynDB.customerTable.zipcode == 'zip'
AND SynDB.customerTable.SSN == SynDB.mortgageTable.SSN))
```

Fig. 5. Query-mutant-killing constraints generated for the query shown in Figure 2

mutants generated by the SC operators requires different sizes of qualified records. For example, a LEFT OUTER JOIN keyword requires the two joined tables contain different numbers of qualified records for conditions in the WHERE clause. To kill such mutants, we specify different cardinality constraints in the transformed code for the query results.

IV. Evaluation

In our evaluation, we seek to evaluate the effectiveness of *MutaGen* by investigating the following research questions:

RQ1: What is the effectiveness of *MutaGen* in generating tests to kill mutants in database applications?
RQ2: What is the effectiveness of *MutaGen* compared with two existing test-generation approaches [2], [11] in terms of mutant killing for testing database applications?

A. Subject Applications and Setup

We conduct the empirical evaluation on two open-source database applications `RiskIt` (https://riskitinsurance.svn.sourceforge.net) and `UnixUsage` (http://sourceforge.net/projects/se549unixusage). `RiskIt` is an insurance-quote application that makes estimation based on users' personal information (e.g., age, income). Its database contains 13 tables and 57 attributes. `UnixUsage` is an application to obtain statistics about how users interact with the Unix systems using different commands. Its database contains 8 tables and 31 attributes. Both applications contain existing records in their databases but we do not use them because our approach is able to conduct test generation from scratch.

For the DSE engine, we use Pex [12], [15], a state-of-the-art tool for .NET programs from Microsoft Research. To test the subject applications in the Pex environment, we convert the original Java code into C# code using a tool called Java2CSharpTranslator (http://sourceforge.net/projects/j2cstranslator/).

The experimental procedure is as follows. To evaluate how effectively *MutaGen* performs in killing program-code mutants, we generate a compiled file for the program under test and apply the tool PexMutator [17] on the compiled file to generate a meta-program that has incorporated weak-mutant-killing constraints. We then generate compiled files for the other programs (e.g., synthesized database interfaces constructed by the SynDB framework). We send these compiled files together with the meta-program of the program under test to Pex for test generation. We insert the generated database records back to the real database, run the original program under test using the generated program inputs, and record the number of weakly killed program-code mutants at each mutation point.

To evaluate how effectively *MutaGen* performs in killing SQL-query mutants, we invoke the tool SQLMutation [13] to generate SQL-query mutants at each query-issuing point and use *MutaGen* to generate tests. We insert the generated database records back to the real database and run the original program with our generated program inputs. To measure the number of weakly killed SQL-query mutants, we compare the returned result sets from executions of the original query and its mutants by checking the metadata (e.g., the numbers of rows, columns, and contents). For both kinds of mutants, we also record the numbers of strongly killed mutants by comparing final results of the program and its mutants.

To compare *MutaGen* with existing test-generation approaches [2], [11], we simulate these approaches using the SynDB framework [9], by not incorporating either program-mutant-killing constraints or query-mutant-killing constraints into the transformed code. We insert the database records generated in this step back to the real database and run the program under test with the generated program inputs to measure the numbers of weakly and strongly killed mutants and the code coverage.

B. Results

Table IV shows detailed evaluation results. In the table, Column 1 lists the subject applications and Column 2 lists method names; the remaining columns give comparisons of effectiveness using tests generated by *MutaGen* and existing approaches [2], [11] from three perspectives: killing program-code mutants (Columns 3-9), killing SQL-query mutants (Columns 10-16), and code coverage (Columns 17-20), respectively. For mutant killing (Columns 3-9 and 10-16), we list the total number of mutants, the number of weakly killed mutants, the number of strongly killed mutants, and percentage increase. For code coverage (Columns 17-20), we list the number of total blocks, covered blocks, and percentage increase. For example, for the first method "filterZipcode" in `RiskIt`, there are 24 program-code mutants in total, of which our *MutaGen* approach weakly kills 22 and strongly kills 16, achieving better mutant-killing ratio (16.7% and 12.5% increase, respectively) than existing approaches. For the total 14 SQL-query mutants, our *MutaGen* approach also achieves better mutant-killing effectiveness (35.7% increase for weak killing and 14.3% increase for strong killing).

In summary, to answer RQ1, *MutaGen* can effectively kill a large portion of both program-code mutants and SQL-query mutants for database applications, leaving a few hard-to-kill mutants. To answer RQ2, *MutaGen* outperforms existing test-generation approaches in terms of mutant killing. For example, for `RiskIt`, *MutaGen* achieves a 16.3% percentage increase on average in weakly killing program-code mutants and a 28.9% percentage increase on average in weakly killing SQL-query mutants, while the average increases are 14.7% and 16.7% in strong mutant killing for the aforementioned two kinds of mutants, respectively. Meanwhile, we report the comparisons of corresponding code coverage (21.3% higher for `RiskIt` and 33.5% higher for `UnixUsage`), of which the increase comes from the merit of the SynDB framework [9].

During the evaluation, we notice that for existing approaches, the not-killed mutants mainly come from the not-covered blocks. Such phenomenon is reasonable because to kill a mutant, at least the mutant must be reachable. We also notice that among the not-killed mutants for *MutaGen*, some are equivalent mutants (mutants that do not change the semantics of the program and cannot be killed by any test), which are impossible to kill. Some other mutants are hard to kill because of the characteristic of the program outputs. For example, quite a few methods return the number of specific records from the query result. Although *MutaGen* is able to generate different database records to weakly kill the mutants, such differences cannot be expressed in producing different final outputs, and thus *MutaGen* is not able to strongly kill the mutants.

V. Related Work

Based on the DSE technique, Zhang et al. [17] developed the PexMutator approach that automatically generates tests to kill mutants, by constructing weak-mutant-killing constraints to guide test generation. However, their approach cannot directly

TABLE IV
EVALUATION RESULTS (NOM: NUMBER OF MUTANTS, MG: *MutaGen*, EA: EXISTING APPROACHES, INC%: PERCENTAGE INCREASE)

Subjects	Methods	Program-Code Mutants							SQL-Query Mutants							Coverage (covered blocks)			
			Weakly Killed			Strongly Killed				Weakly Killed			Strongly Killed						
		NOM	MG	EA	Inc%	MG	EA	Inc%	NOM	MG	EA	Inc%	MG	EA	Inc%	Total	MG	EA	Inc%
RiskIt	filterZipcode	24	22	18	16.7	16	13	12.5	14	11	6	35.7	8	6	14.3	42	38	28	23.8
	filterEducation	20	18	14	20.0	12	9	15.0	62	43	31	19.4	32	23	14.5	41	37	27	24.4
	filterMaritalStatus	20	18	14	20.0	12	9	15.0	14	11	6	35.7	8	6	14.3	41	37	27	24.4
	getAllZipcode	27	23	19	14.8	17	14	11.1	85	69	45	28.2	51	33	21.2	39	37	17	51.3
	filterEstimatedIncome	24	22	18	16.7	16	13	12.5	122	98	68	24.6	70	53	13.9	58	54	44	17.2
	getOneZipcode	32	28	25	9.4	21	17	12.5	14	11	6	35.7	8	6	14.3	34	32	23	26.5
	getValues	44	38	33	11.4	29	22	15.9	56	44	28	28.6	32	22	22.7	107	99	68	29.0
	userinformation	37	33	26	18.9	27	21	16.2	70	58	39	27.1	41	29	17.1	61	57	51	9.8
	updatestability	42	37	29	19.0	30	21	21.4	64	50	34	25.0	35	24	17.2	79	67	75	10.1
all methods (total)		270	239	196	16.3	180	140	14.7	501	395	263	28.9	285	202	16.7	502	458	360	21.3
Unix-Usage	raceExists	12	9	7	16.7	6	5	8.3	16	13	8	31.3	9	6	18.8	11	11	7	36.4
	transcriptExist	12	9	7	16.7	6	5	8.3	16	13	8	31.3	9	6	18.8	11	11	7	36.4
	retrieveMaxLineNo	9	9	8	11.1	7	5	22.2	14	11	7	28.6	5	3	14.3	10	10	7	30.0
	getUserInfoBy	14	11	9	14.3	7	6	7.1	16	13	8	31.3	9	6	18.8	47	47	15	68.1
	doesUserIdExist	12	9	7	16.7	6	5	8.3	12	10	7	25.0	7	4	25.0	10	10	9	10.0
	getPrinterUsage	16	14	11	18.8	9	5	25.0	36	32	22	27.8	28	17	30.6	34	34	27	20.1
all methods (total)		75	61	49	15.7	41	31	13.2	110	92	60	29.2	67	42	22.5	123	123	75	33.5

deal with database applications. Mutation testing was also applied in testing database applications. Tuya et al. [13], [14] proposed a set of mutation operators for SQL queries and integrated these operators into a tool called SQLMutation that generates SQL-query mutants automatically. Shah et al. [10] proposed an approach that focuses in particular on a class of join/outer-join mutations, comparison operator mutations, and aggregation operation mutations. Their approach generates tests for killing a predefined subclass of mutations. Zhou and Frankl [18] developed the JDAMA approach that leverages a set of mutation operators for SQL queries to evaluate the quality of database states generated by existing test-generation techniques. However, these previous approaches have difficulty in handling complex program contexts in database applications.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have proposed an approach called *MutaGen* that generates tests for mutation testing on database applications. In our approach, we leverage the newly developed SynDB framework [9] that relates program-execution constraints and query constraints within a database application. We incorporate weak-mutant-killing constraints for the original program code and query-mutant-killing constraints for the SQL queries into the transformed code, guiding DSE to generate both effective program inputs and sufficient database states to kill mutants. Evaluation results show that *MutaGen* achieves high effectiveness and outperforms existing test-generation approaches in killing both program-code mutants and SQL-query mutants.

In future work, we plan to investigate how to generate program inputs based on a given database state for mutation testing. We also plan to investigate techniques of augmenting existing tests to detect logical faults [7] in database applications.

ACKNOWLEDGMENT

This work was supported in part by U.S. National Science Foundation under CCF-0915059 for Kai Pan and Xintao Wu, and under CCF-0845272, CCF-0915400, CNS-0958235, and CNS-1160603 for Tao Xie.

REFERENCES

[1] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Software Eng.*, 17(9):900–910, 1991.
[2] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proc. ISSTA*, pages 151–162, 2007.
[3] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.
[4] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. Software Eng.*, 8(4):371–379, 1982.
[5] J. C. King. Symbolic execution and program testing. In *Commun. ACM, 19(7):385394*, 1976.
[6] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, 1996.
[7] K. Pan, X. Wu, and T. Xie. Database state generation via dynamic symbolic execution for coverage criteria. In *Proc. DBTest*, pages 4–9, 2011.
[8] K. Pan, X. Wu, and T. Xie. Generating program inputs for database application testing. In *Proc. ASE*, pages 73–82, 2011.
[9] K. Pan, X. Wu, and T. Xie. Guided test generation for database applications via synthesized database interactions. Technical report, DPL-2012-6-1,UNC Charlotte, 2012.
[10] S. Shah, S. Sudarshan, S. Kajbaje, S. Patidar, B. P. Gupta, and D. Vira. Generating test data for killing SQL mutants: A constraint-based approach. In *Proc. ICDE*, pages 1175–1186, 2011.
[11] K. Taneja, Y. Zhang, and T. Xie. MODA: Automated test generation for database applications via mock objects. In *Proc. ASE*, pages 289–292, 2010.
[12] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.
[13] J. Tuya, M. J. S. Cabal, and C. de la Riva. SQLMutation: A tool to generate mutants of SQL database queries. In *Proc. Mutation*, pages 1–5, 2006.
[14] J. Tuya, M. J. S. Cabal, and C. de la Riva. Mutating database queries. *Information & Software Technology*, 49(4):398–417, 2007.
[15] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. DSN*, pages 359–368, 2009.
[16] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proc. ICSE*, pages 435–444, 2010.
[17] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proc. ICSM*, pages 1–10, 2010.
[18] C. Zhou and P. G. Frankl. Mutation testing for Java database applications. In *Proc. ICST*, pages 396–405, 2009.