Automatic Test Generation for Mutation Testing on Database Applications

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

Tao Xie
North Carolina State University
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). Compared 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 check 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 enough support for killing mutants in database applications (in either program code or its embedded or resulted SQL queries). In this paper, we propose an approach called MutaGen that conducts test generation for mutation testing on database applications. In our approach, we first correlate 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.

1. 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. Compared to code coverage criteria (a popular type of testing requirements), mutation testing [1] 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 check 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 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 et al. [16] 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 [2, 15]. However, for mutation testing on database applications, tests consist of both program inputs and database states. Thus, these approaches become inapplicable for database applications since sufficient and supportive back-end database states are required for generated tests. Focusing on test generation for database application testing, some recent approaches [6, 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 [6, 11] for database applications choose to consider program-execution constraints and environment constraints separately. Thus, when applying existing approaches [6, 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 very costly or even infeasible [9]. On the other hand, although a recent approach called PexMutator [15] incorporates all the mutant-
To address these issues, in this paper, we propose a new approach called *MutaGen* (Test Generation for Mutation Testing on Database Applications) for killing mutants in database applications based on our previous SynDB framework [9]. The SynDB framework is based on Dynamic Symbolic Execution (DSE) [4, 10, 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 [15] 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 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.

2. BACKGROUND

In this section, we present some technical background about mutation testing. We present the background of our previous SynDB framework in Section 3.2.

Mutation testing [1] is a fault-based testing technique that is intensively studied for evaluating the adequacy of tests. The original program under test is mutated into a set of new programs, called *mutants*, each caused by applying a small syntactic change on the original program following a set of rules (called *mutation operators*). A mutant is *(strongly)* killed by the given tests if running the mutant against the given tests produces different testing results (passed or failed) than the results of running the original program against the given tests. 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 [2], because there may be a large number of mutants for a short program. To deal with the high cost of mutation testing, Howden et al. [5] proposed weak mutation testing, which focuses on intermediate results or outputs from components of the program under test. Instead of checking the execution results of mutants after the execution of the entire program, the execution results of the mutants need to be checked only immediately after the mutated components. In addition, to reduce time or space resources exhausted by a large number of mutants, instead of using all the mutation operators to generate mutants, a subset of the mutation operators. For example, Offutt et al. [7] proposed that 5 mutation operators (namely, ABS, AOR, ROR, LCR, and UOI) can perform as effectively as all the 22 mutation operators [3].

Mutation testing has also been applied to assess the adequacy of tests in terms of detecting faults in SQL queries. Tuya et al. [13, 14] proposed a set of mutation operators [14] and developed a tool called 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: mutate the most distinctive features of SQL (e.g., clauses, aggregate functions).
- **OR** - Operator replacement mutation operators: extend the expression modification operators.
- **NL** - NULL mutation operators: produce mutants related with incorrect treatment of NULL values.
- **IR** - Identifier replacement mutation operators: replace 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.

3. APPROACH

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

3.1 A Motivating Example

The code snippet 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 1. 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`.
Table 2: Program inputs and database states to cover paths for program code in Figure 1

<table>
<thead>
<tr>
<th>input</th>
<th>customer table</th>
<th>mortgage table</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>zipcode</td>
<td>M.balance</td>
</tr>
<tr>
<td>001</td>
<td>20</td>
<td>50000</td>
</tr>
<tr>
<td>010</td>
<td>25223</td>
<td>105000</td>
</tr>
</tbody>
</table>

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' + inputAge + M.M." + 
          "AND C.zipcode = 'inputAge' + zip + 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;}

Figure 2: Code transformation for example code in Figure 1 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 structural coverage, existing test-generation approaches [6, 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 1 could achieve full code coverage: a default value inputAge = 0 and an empty database state covers the path where Line 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 2 are not sufficient. Killing mutants in database applications requires more program inputs and multiple database records so that executing the program 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 2 could kill this mutant since the final program outputs are 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 satisfactory mutant-killing effectiveness requires both effective program inputs and sufficient database states.

3.2 SynDB Framework Revisited

MutaGen is based on our previous SynDB framework [9]. The framework 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. For example,

1Although for inputAge = 40, the mutant C.age <= 'inputAge' is weakly killed because executions of the original query and this mutant on Table 2 produce different result sets.

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<customer> mortgageRecords;
    public void checkConstraints() {
        /method for checking schema constraints*/}
}

Figure 3: Synthesized database state

The transformed code of the example code in Figure 1 is shown in Figure 2.

In the framework, we identify and replace the original database interactions with renamed API methods (e.g., we add “Syn” before each method name). We also construct a synthesized database state to replace the physical one according to the given database schema. We define tables and attributes within the synthesized database state and use auxiliary methods to enforce schema constraints. We treat the synthesized database state as an object and add it as an input to the program under test. For example, according to the schema in Table 1, we construct a synthesized database state shown in Table 3. In Figure 2, we add a new input dbState with the type DatabaseState to the program. We then pass the synthesized database state within the synthesized database interactions. For each database interacting interface (e.g., database connection, query construction, and query execution), we add a new field to represent the synthesized database state and use 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. We incorporate 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 we apply Dynamic Symbolic Execution (DSE) [4, 10, 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 by executing a program under test with concrete inputs and collecting concrete and symbolic information at runtime [4, 10, 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.

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

3.3 Mutant Killing

Base on the transformed code with the SynDB framework [9], MutaGen conducts mutant killing for database applications from two aspects: killing mutants in the original normal program code and killing SQL-query mutants. Based on the transformed code, MutaGen seeds code-mutant-killing constraints by applying an existing mutant-generation tool [15]. To kill SQL-query mutants, Mu-
### 3.3.1 Killing Program-Code Mutants

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

In our approach, MutaGen applies a tool called PexMutator [15] 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 using sufficient mutation operators. In the literature, Offutt et al. [7] proposed that 5 mutation operators (called sufficient mutation operators: ABS, AOR, ROR, LCR, and UOI) could achieve as effective performance as all the 22 mutation operators [3].

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

Figure 4 shows a code snippet of applying PexMutator on the transformed code shown in Figure 2. For example, 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 variable diff is calculated from the 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. For example, tests to kill the three mutants are shown in Table 3.

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 [6, 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 killing.

### 3.3.2 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 call the tool SQLMutation [13] to generate its mutants. SQLMutation is developed by Tuya et al. [13] that 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 aforementioned in Section 2, 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. Algorithm 1 shows 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 a given query. In Algorithm 1, we construct an empty statement set S (Line 1) and a SQL-query mutant set Qm 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 Qm, if q is generated by the mutation operators OR, NL, or IR, we retrieve its WHERE clause s2 and construct a weak-mutant-killing constraint s = (s1 AND s2) OR (s1 AND s2). Note that if a record r satisfies conditions in s, r can only satisfy 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. After dealing with all the mutants in Qm, the algorithm finally returns the set S.

To avoid causing syntactic errors, in the transformed code, we insert these constraints before the original query.
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 existing test-generation approaches [6, 11] in terms of mutant killing and code coverage for database application testing?

### 4. EVALUATION

In our evaluation, we conduct the empirical evaluation on two open source database applications RiskIt\(^1\) and UnixUsage\(^2\). RiskIt is an insurance quote application that makes estimation based on users’ personal information (e.g., age, income). Its database contains 13 tables, 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, 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. We use Pex [12], a state-of-the-art tool, for .NET from Microsoft Research as the DSE engine. To test the subject applications in the Pex environment, we convert the original Java code into C# code using a tool called Java2CSharpTranslator\(^3\). The detailed evaluation subjects and results can be found on our project website\(^4\).

The experimental procedure is as follows. To evaluate how MutaGen performs in killing program-code mutants, we generate compiled file for the program under test and apply the tool PexMutator [15] 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 our 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 MutaGen performs in killing SQL-query mutants, we call 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., number of rows, columns, and contents). For both two kinds of mutants, we also record the numbers of strongly killed mutants by comparing final results of the program.

To compare MutaGen with existing test-generation approaches [6, 11], we simulate these approaches using our 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 generated program inputs to measure the numbers of weakly and strongly killed mutants and the code coverage.

### 4.1 Subject Applications and Setup

We conduct the empirical evaluation on two open source database applications RiskIt\(^1\) and UnixUsage\(^2\). RiskIt is an insurance quote application that makes estimation based on users’ personal information (e.g., age, income). Its database contains 13 tables, 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, 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. We use Pex [12], a state-of-the-art tool, for .NET from Microsoft Research as the DSE engine. To test the subject applications in the Pex environment, we convert the original Java code into C# code using a tool called Java2CSharpTranslator\(^3\). The detailed evaluation subjects and results can be found on our project website\(^4\).

The experimental procedure is as follows. To evaluate how MutaGen performs in killing program-code mutants, we generate compiled file for the program under test and apply the tool PexMutator [15] 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 our 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 MutaGen performs in killing SQL-query mutants, we call 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., number of rows, columns, and contents). For both two kinds of mutants, we also record the numbers of strongly killed mutants by comparing final results of the program.

To compare MutaGen with existing test-generation approaches [6, 11], we simulate these approaches using our 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 generated program inputs to measure the numbers of weakly and strongly killed mutants and the code coverage.

### 4.2 Results

Table 4 shows detailed evaluation results. In the table, Column 1 lists the subject applications and Column 2 lists method names; the remaining columns show comparisons of effectiveness using tests generated by MutaGen and existing approaches [6, 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 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, we also achieve better mutant-killing effectiveness (35.7% increase for weak-killing and 14.3% increase for strong-killing). Meanwhile, for the total 42 blocks, MutaGen achieves better code coverage (23.8% increase) than existing approaches.

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, MutaGen achieves higher code coverage (21.3% increase for RiskIt and 33.5% increase for UnixUsage).

5. 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 our previous 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 in database applications.

6. REFERENCES


Table 4: Evaluation Results(NOM: Number of Mutants, MG: MutaGen, EA: Existing Approaches, Inc%: Percentage Increase)

<table>
<thead>
<tr>
<th>Subjects</th>
<th>Methods</th>
<th>Program-Code Mutants</th>
<th>SQL-Query Mutants</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Weakly Killed</td>
<td>Strongly Killed</td>
<td>Weakly Killed</td>
</tr>
<tr>
<td></td>
<td>NOM</td>
<td>MG</td>
<td>EA Inc%</td>
<td>NOM</td>
</tr>
<tr>
<td>RiskIt</td>
<td>filter/Zipcode</td>
<td>24</td>
<td>22</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>filterEducation</td>
<td>20</td>
<td>18</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>filterMaritalStatus</td>
<td>20</td>
<td>18</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>getAllZipcode</td>
<td>27</td>
<td>23</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>filterEstimatedIncome</td>
<td>24</td>
<td>22</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>getOneZipcode</td>
<td>32</td>
<td>28</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>getValues</td>
<td>44</td>
<td>38</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>userinformation</td>
<td>37</td>
<td>33</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>updatestability</td>
<td>42</td>
<td>37</td>
<td>29</td>
</tr>
<tr>
<td>UnixUsage</td>
<td>raceExists</td>
<td>12</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>transcriptExist</td>
<td>12</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>retrieveMaxLineNo</td>
<td>9</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>getUserName</td>
<td>12</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>doesUserIDExist</td>
<td>12</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>getPrinterUsage</td>
<td>16</td>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>all methods (total)</td>
<td>75</td>
<td>61</td>
<td>49</td>
<td>13.7</td>
</tr>
</tbody>
</table>