Guided Test Generation for Database Applications via Synthesized Database Interactions

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

Testing database applications typically requires the generation of tests consisting of both program inputs and database states. Recently, a testing technique called Dynamic Symbolic Execution (DSE) has been proposed to reduce manual effort in test generation for software applications. However, applying DSE to generate tests for database applications faces various technical challenges. For example, the database application under test needs to physically connect to the associated database, which may not be available for various reasons. The program inputs whose values are used to form the executed queries are not treated symbolically, posing difficulties for generating valid database states or appropriate database states for achieving high coverage of query-result-manipulation code. To address these challenges, in this paper, we propose an approach called SynDB that synthesizes new database interactions to replace the original ones from the database application under test. In this way, we bridge various constraints within a database application: query-construction constraints, query constraints, database schema constraints, and query-result-manipulation constraints. We then apply a state-of-the-art DSE engine called Pex for .NET from Microsoft Research to generate both program inputs and database states. The evaluation results show that tests generated by our approach can achieve higher code coverage than existing test generation approaches for database applications.

1. INTRODUCTION

For quality assurance of database applications, testing is essential before the applications are deployed. Among different types of testing, functional testing focuses on functional correctness. There, covering a branch is necessary to expose a potential fault within that branch. To cover specific branches, it is crucial to generate appropriate tests, including both appropriate program inputs (i.e., input arguments) and database states. However, manually producing these tests could be tedious and even infeasible. To reduce manual effort in test generation, a testing technique called Dynamic Symbolic Execution (DSE) has been proposed [10, 15]. DSE extends the traditional symbolic execution by running a program with concrete inputs while collecting both concrete and symbolic information at runtime, making the analysis more precise [10]. DSE first starts with default or random inputs and executes the program concretely. Along the execution, DSE simultaneously performs symbolic execution to collect symbolic constraints on the inputs obtained from predicates in branch conditions. DSE flips a branch condition and conjuncts the negated branch condition with constraints from the prefix of the path before the branch condition. DSE then hands the conjuncted conditions to a constraint solver to generate new inputs to explore not-yet-covered paths. The whole process terminates when all the feasible program paths have been explored or the number of explored paths has reached the predefined upper bound.

Recently, some approaches [12, 16] adapt this technique to generate tests, including both program inputs and database states, for achieving high structural coverage of database applications. Emmi et al. [12] proposed an approach that runs the program simultaneously on concrete program inputs as well as on symbolic inputs and a symbolic database. In the first run, it uses random concrete values for the program inputs, collects path constraints over the symbolic program inputs along the execution path, and generates database records such that the program execution with the concrete SQL queries (issued to the database during the concrete execution) can cover the current path. To explore a new path, the approach flips a branch condition and generates new program inputs and corresponding database records. However, sometimes the associated database is not available for various reasons. For example, the associated real database could be confidential, or setting up a local test database may be difficult or infeasible. Moreover, interactions between the application and real database could be intensive and time-consuming. To test an application under such scenario, the MODA framework [16] replaces the real database with a mock database. The approach simulates the operations on the mock database and transforms the code into a form that interacts with the mock database. Then, based on the transformed code, the approach applies a DSE-based test generation tool called Pex [3] for .NET to collect constraints of both program inputs and the associated database state, in order to generate tests that can achieve high code coverage. Thus, it is able to iteratively generate tests to cover feasible paths. The approach also inserts the generated records back to the mock database, so that the query execution on the mock database could return appropriate results.

In general, for database applications, constraints used to generate effective program inputs and sufficient database states often come from four parts: (1) query-construction constraints, where constraints come from the sub-paths being explored before the query-issuing location; (2) query constraints, where constraints come from conditions in the query’s WHERE clause; (3) database schema constraints, where constraints are predefined for attributes in the database...
schema; (4) query-result-manipulation constraints, where constraints come from the sub-paths being explored for iterating through the query result. Basically, query-construction constraints and query-result-manipulation constraints are program-execution constraints while query constraints and database schema constraints are environment constraints. Typically, program-execution constraints are solved with a constraint solver for test generation, but a constraint solver could not directly handle environment constraints.

Considering the preceding four parts of constraints, applying DSE on testing database applications faces great challenges for generating both effective program inputs and sufficient database states. For existing DSE-based approaches of testing database applications, it is difficult to correlate program-execution constraints and environment constraints. Performing symbolic execution of database interaction API methods would face a significant problem: these API methods are often implemented in either native code or unmanaged code, and even when they are implemented in managed code, their implementations are of high complexity; existing DSE engines have difficulty in exploring these API methods. In practice, existing approaches [12, 16] would replace symbolic inputs involved in a query with concrete values observed at runtime. Then, to allow concrete execution to iterate through a non-empty query result, existing approaches generate database records using constraints from conditions in the WHERE clause of the concrete query and insert the records back to the database (either real database [12] or mock database [16]) so that it returns a non-empty query result for query-result-manipulation code to iterate through.

A problem of such design decision made in existing approaches is that values for variables involved in the query issued to the database system could be prematurely concretized. Such premature concretization could pose barriers for achieving structural coverage because query constraints (constraints from the conditions in the WHERE clause of the prematurely concretized query) may conflict with later constraints, such as database schema constraints and query-result-manipulation constraints. In particular, the violation of database schema constraints could cause the generation of invalid database states, thus causing low code coverage of database application code in general. On the other hand, the violation of query-result-manipulation constraints could cause low code coverage of query-result manipulation code. Basically, there exists a gap between program-execution constraints and environment constraints, caused by the complex black-box query-execution engine. Treating the connected database (either real or mock) as an external component isolates the query constraints with later constraints such as database schema constraints and query-result-manipulation constraints.

In this paper, we propose a DSE-based test generation approach called SynDB to deal with preceding challenges. In our approach, we treat symbolically both the embedded query and the associated database state by constructing synthesized database interactions. We transform the original code under test into another form that the synthesized database interactions can operate on. To force DSE to actively track the associated database state in a symbolic way, we treat the associated database state as a synthesized object, add it as an input to the program under test, and pass it among the generated database interactions. The synthesized database interactions integrate the query constraints into normal program code. We also check whether the database state is valid by incorporating the database schema constraints into normal program code. Through this way, we correlate aforementioned four parts of constraints within a database application, and bridge the gap of program-execution constraints and environment constraints. Then, based on the transformed code, we guide DSE’s exploration through the operation.

This paper makes the following main contributions:

- An automatic test generation approach to solve significant challenges of existing test generation approaches for testing database applications, even when the associated physical database is not available.
- A novel test generation technique based on DSE through code transformation for correlating various parts of constraints in database applications, bridging query construction, query execution, and query-result manipulation.
- A prototype implemented for the proposed approach and evaluations on real database applications to assess the effectiveness of our approach. Empirical evaluations show that our approach can generate effective program inputs and sufficient database states that achieve higher code coverage than existing DSE-based test generation approaches for database applications.

2. ILLUSTRATIVE EXAMPLE

In this section, we first use an example to intuitively introduce aforementioned problems of existing test generation approaches. Then we apply our SynDB approach on the example code to illustrate how our approach works.

The code snippet in Figure 1 includes a portion of C# code from a database application that calculates some statistics related to customers’ mortgages. The schema-level descriptions and constraints of the associated database are given in Table 1. The method calcStat first sets up database connection (Lines 03-05). It then constructs
### Table 1: Database schema

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Constraint</th>
<th>Attribute</th>
<th>Type</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>Int</td>
<td>Primary Key</td>
<td>SSN</td>
<td>Int</td>
<td>Primary Key</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>∈ [F, M]</td>
<td>year</td>
<td>Int</td>
<td>∈ [10, 15, 30]</td>
</tr>
<tr>
<td>gender</td>
<td>String</td>
<td>∈ {0, 1}</td>
<td>balance</td>
<td>Int</td>
<td>∈ [100000, Max]</td>
</tr>
<tr>
<td>zipcode</td>
<td>Int</td>
<td>∈ [00001, 99999]</td>
<td>income</td>
<td>Int</td>
<td>∈ [0, 100]</td>
</tr>
</tbody>
</table>

```java
01: public int calcStat(int inputYear, DatabaseState dbState) {
02:     int zip = 28223, count = 0;
03:     SqlConnection sc = new SynSqlConnection(dbState);
04:     sc.ConnectionString = "..";
05:     sc.Open();
06:     string query = buildQuery(zip, inputYear);
07:     SynSqlCommand cmd = new SynSqlCommand(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 year = results.GetInt(3);
13:         int diff = (income - 1.5 * balance) * year;
14:         if (diff > 100000)
15:             count++;
16:     }
17:     return count;
```

```java
06a: public string buildQuery(int x, int y) {
06b:     string query = "SELECT C.SSN, C.income,"
06c:         +"M.balance, M.year FROM customer C, mortgage M"
06d:         +" WHERE C.SSN = M.SSN AND C.zipcode =" + x + "}"
06e:         +" AND M.year =" + y + "}";
06c:     return query;
```

**Figure 2:** Transformed code produced by SynDB for the code in Figure 1

A query by calling another method buildQuery (Lines 06, 06a, 06b, and 06c) and executes the query (Lines 07-08). Note that the query is built with two program variables: a local variable zip and a program-input argument inputYear. The returned result records are then iterated (Lines 09-15). For each record, a variable diff is calculated from the values of the fields C.income, M.balance, and M.year. If diff is greater than 100000, a counter variable count is increased (Line 15). The method then returns the final result (Line 16). To achieve high structural coverage of this program, we need appropriate combinations of database states and program inputs.

Typically, a database application communicates with the associated database through four steps. First, the application sets up a connection with the database (e.g., construct a SqlConnection object). Second, it constructs a query to be executed and combines the query into the connection (e.g., construct a SqlCommand object using the database connection and the string value of the query). Third, if the query’s execution yields an output, the result is returned (e.g., construct a SqlDataReader object by calling the API method ExecuteReader()). Fourth, the returned query result is manipulated for further execution.

To test the preceding code, for existing DSE-based test generation approaches [12, 16] where the application interacts with either real database or mock database, in the first run, DSE chooses random or default values for inputYear (e.g., inputYear = 0 or inputYear = 1). For generation of a database state, the con-
into normal program code (e.g., whose exploration helps derive path conditions). The query result is then assigned to the variable `results` with the synthesized type `SynSqlDataReader`. The query result eventually becomes an output of the operation on the symbolic database state.

We then apply a DSE engine on the transformed code to conduct test generation. In the first run, DSE chooses random or default values for `inputYear` and `dbState` (e.g., `inputYear = 0`, `dbState = null`). The value of `dbState` is passed through `sc` and `cmd`. Note that for the database connection in Line 05, DSE’s exploration is guided to check database schema constraints for each table (e.g., `Mortgage.year ∈ [10, 15, 30]`). Then, in Line 08, DSE’s exploration is guided to collect query constraints from the symbolic query. In Line 09, because the query result is empty, DSE stops and tries to generate new inputs. To cover the new path where `Line 09 == true`, the DSE engine generates appropriate values for both `inputYear` and `dbState` using a constraint solver based on the collected constraints. The generated program input and database records are shown in Table 2 (e.g., `inputYear = 15` and the record with `C.SSN = 001`). In the next run, the execution of the query whose WHERE clause has been updated as `C.SSN = M.SSN AND C.zipcode = 28223 AND M.year = 15` yields a record so that DSE’s exploration enters the `while` loop (Lines 09-15). Straightforwardly, the transformed code can also guide DSE’s exploration to collect later constraints (Line 14) from sub-paths for manipulating the query result to generate new inputs. For example, to cover `Line 14 == true`, the collected new constraint `(income - 1.5 * balance) * year` is combined with previous constraints to generate new inputs (e.g., the input `inputYear = 15` and the record with `C.SSN = 002` as shown in Table 2).

3. APPROACH

Our approach relates the query construction, query execution, and query result manipulation in one seamless framework. We conduct code transformation on the original code under test by constructing synthesized database interactions. We treat the database state symbolically and add it as an input to the program. In the transformed code, the database state is passed through synthesized database interactions. At the beginning of the synthesized database connection, we enforce database schema constraints via checking code. The synthesized database interactions also incorporate query constraints from conditions in the WHERE clause of the symbolic query into normal program code. Then, when a DSE engine is applied on the transformed code, DSE’s exploration is guided to collect constraints for both program inputs and database states. In this way, we generate sufficient database states as well as effective program inputs.

3.1 Code Transformation

For the code transformation, we transform the code under test into another form upon which our synthesized database interactions can execute. Basically, we replace the standard database interactions with renamed API methods. We mainly deal with the statements or stored procedures to execute against a SQL Server database [2]. We identify relevant method calls including the standard database API methods. We replace the original database API methods with new names (e.g., we add “Syn” before each method name). Note that replacing the original database API methods is a large body of work. Even a single class could contain many methods and their relationships could be very complex. In our `SynDB` framework, we mainly focus on the classes and methods that are commonly used and can achieve the basic functionalities of database applications. Table 3 gives a summary of the code transformation part.

We construct a synthesized object to represent the whole database state, according to the given database schema. Within the synthesized database state, we define tables and attributes. For example, for the `Customer` table in Table 1, the corresponding synthesized database state is shown in Figure 3. Meanwhile, we check database schema constraints for each table and each attribute by transforming the database schema constraints into normal program code for checking these constraints. Note that we are also able to capture complex constraints at the schema level such as constraints across multiple tables and multiple attributes. We then add the synthesized database state as an input to the transformed code. Through this way, we force DSE to track the associated database state symbolically and guide DSE’s exploration to collect constraints of the database state.

3.2 Database Interface Synthesization

We use synthesized database interactions to pass the synthesized database state, which has been added as a new input to the program. 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. Thus, DSE’s exploration on the transformed code is guided to track the synthesized database state symbolically through these database interactions. For example, as listed in Table 3, for the interactions `SqlConnection` and `SqlCommand`, we add new fields and new methods.

For the synthesized database connection, at the beginning, we enforce the checking of database schema constraints by calling auxiliary methods predefined in the passed synthesized database state. In this way, we guarantee that the passed database state is valid. It is also guaranteed that the further operations issued by queries (e.g., `SELECT` and `INSERT`) on this database state would yield valid results. Figure 4 gives the details of the synthesized database connection. For example, in `SqlConnection`, we rewrite the method `Open()` by calling the method `checkConstraints()` predefined in the passed synthesized database state. Then, we synthesize new API methods to execute the query and synthesize a new data type to represent the query result. For example, we rewrite API methods to execute a query against the synthesized database state, according to various kinds of queries (e.g., queries to select database records, and queries to modify the database state). Figure 5 gives the details of `SqlCommand` whose methods `ExecuteReader()` and `ExecuteNonQuery()` are used to execute queries. The details of algorithms for `ExecuteReader()` and `ExecuteNonQuery()` are discussed later in Section 3.3 (Algorithms 1 and 2, respectively).

```java
public class SqlConnection{
  ...
  public DatabaseState dbStateConn; // new field
  public void Open()
  {
    (dbStateConn.checkConstraints())
    // new methods
    public SynSqlConnection(DatabaseState dbStatePara)
    {
      (dbStateConn = dbStatePara); // modified method
    ...
  }
}
```

![Figure 4: Synthesized SqlConnection](image-url)
3.3 Database Operation Synthesization

In this section, we illustrate how to use the preceding synthesized database interactions to implement database operations. A database state is read or modified by executing queries issued from a database application. In our SynDB framework, we parse the symbolic query and transform the constraints from conditions in the WHERE clause into normal program code (e.g., whose exploration helps derive path conditions).

3.3.1 Select Operation

We first discuss how to deal with the SELECT statement for a simple query. A simple query (shown in Figure 6) consists of three parts. In the FROM clause, there is a from-list that consists of a list of tables. In the SELECT clause, there is a list of column names of tables named in the FROM clause. In the WHERE clause, there is a qualification that is a boolean combination of conditions connected by logical connectives (e.g., AND, OR, and NOT). A condition is of the form expression op expression, where op is a comparison operator (=, <>, >, >=, <, <=) or a membership operator (IN, NOT IN) and expression is a column name, a constant, or an arithmetic or string expression. We leave discussion for complex queries in Section 3.4.

In our approach, we rewrite the method ExecuteReader() (shown in Figure 5) to deal with the SELECT statement. The return value of the method is a SynSqlDataReader object. Recall that the SynSqlCommand object contains a field dbStateComm to represent the symbolic database state. We evaluate the SELECT statement on this symbolic database state in three steps. The details of executing the SELECT statement is shown in Algorithm 1. First, we compute a cross-product of related tables to get all rows based on the FROM clause (Lines 1-14). A cross-product operation computes a relation instance that contains all the fields of one table followed by all fields of another table. One tuple in a cross-product is a concatenation of two tuples coming from the two tables. To realize cross-product computation, we update the columns of the field
We compute the cross-product by copying all the rows to `DataTable resultSet`. Second, from the cross-product, we select rows that satisfy the conditions specified in the `WHERE` clause (Lines 15-22). For each row `r`, if it satisfies the conditions, we move to check the next row; otherwise, we remove `r`. Note that, as aforementioned, we deal with the `SELECT` statement for a simple query whose `WHERE` clause contains a qualification that is a boolean combination of conditions connected by logical connectives (e.g., AND, OR, and NOT). In this step, we transform the evaluation of the conditions specified in the `WHERE` clause into normal program code in the following way. From the `WHERE` clause, we replace the database attributes in the conditions with their corresponding column names in `DataTable resultSet`. We also map those SQL logical connectives (e.g., AND, OR, and NOT) to program logical operators (e.g., &&, ||, and !), thus keeping the original logical relations unchanged. After these transformations, we push the transformed logical conditions into parts of a path condition (e.g., realized as an assumption recognized by the DSE engine).

Third, after scanning all the rows, we remove unnecessary columns from `DataTable resultSet` based on the `SELECT` clause (Lines 23-28). For each column `c` in `DataTable resultSet`, if it appears in the `SELECT` clause, we keep this column; otherwise, we remove `c`. After the preceding three steps, the field `DataTable resultSet` contains all rows with qualified values that the `SELECT` statement should return.

Through this way, we construct a `SynSqlDataReader` object to relate the previous query execution and the path conditions later executed in the program. We transform the later manipulations on the `SynSqlDataReader` object to be indirect operations on the initial symbolic database state. To let this `SynSqlDataReader` object satisfy the later path conditions, the test generation problem is therefore transformed to generating a sufficient database state against which the query execution can yield an appropriate returned result.

### 3.3.2 Modify Operation

To deal with queries that modify database states, we rewrite the method `ExecuteNonQuery()` (shown in Figure 5). The pseudocode is shown in Algorithm 2. The method also operates on the field `dbStateComm` that represents the symbolic database state. We first check the modification type of the query (e.g., INSERT, UPDATE, and DELETE). For the `INSERT` statement (Lines 1-8), from the table in the `INSERT INTO` clause, we find the corresponding table in `dbStateComm`. From the `VALUES` clause, we then check whether the values of the new row to be inserted satisfy database schema constraints. We also check after this insertion, whether the whole database state still satisfy database schema constraints. If both yes, we add this new row to the target table in `dbStateComm`, by mapping the attributes from the `INSERT` query to their corresponding fields. For the `UPDATE` statement (Lines 9-19), from the `UPDATE` clause, we find the corresponding table in `dbStateComm`. We scan the table with the conditions from the `WHERE` clause and locate target rows. For each row, we also check whether the specified values satisfy the schema constraints. If qualified, we set the new values to their corresponding columns based on the `SET` clause. For the `DELETE` statement (Lines 20-30), from the `DELETE FROM` clause, we find the corresponding table in `dbStateComm`. We locate the target rows using conditions from the `WHERE` clause. We then check whether this deletion would violate the schema constraints; otherwise, we remove these rows.

#### Algorithm 2: `ModifyExec`: Evaluate a modification statement on a symbolic database state

**Input:** DatabaseState `dbStateComm`, a modification query `Q`  

1. if `Q` is an `INSERT` statement then  
2. : Get table `T` from `Q`'s `INSERT INTO` clause;  
3. : Find `T`'s corresponding table `T'` in `dbStateComm`;  
4. : Construct a new row `r` based on `VALUES` clause;  
5. : if `T'`.check(`r`) == true && `dbStateComm.T'`.afterInsert(`r`) == true then  
6. : `dbStateComm.T'`.Add(`r`);  
7. : end if  
8. : end if  
9. : if `Q` is an `UPDATE` statement then  
10. : Get table `T` from `Q`'s `UPDATE` clause;  
11. : Find `T`'s corresponding table `T'` in `dbStateComm`;  
12. : for each row `r` in `T` do  
13. : if `r` satisfies the conditions in `Q`'s `WHERE` clause then  
14. : if `dbStateComm.T'`.afterUpdate(`r`) == true then  
15. : Set `r` with the specified values;  
16. : end if  
17. : end if  
18. : end for  
19. : end if  
20. : if `Q` is a `DELETE` statement then  
21. : Get table `T` from `Q`'s `DELETE FROM` clause;  
22. : Find `T`'s corresponding table `T'` in `dbStateComm`;  
23. : for each row `r` in `T` do  
24. : if `r` satisfies the conditions in `Q`'s `WHERE` clause then  
25. : if `dbStateComm.T'`.afterDelete(`r`) == true then  
26. : `dbStateComm.T'`.Remove(`r`);  
27. : end if  
28. : end if  
29. : end for  
30. : end if

SELECT `C1`, `C2`, ..., `Ch` FROM `from-list` WHERE (`All AND ... AND A1n`) OR ... OR (`A1m AND ... AND Ann`)  

---

3.4 Discussion

In this section, we present some complex cases that often occur in database applications. We introduce how our approach can deal with these cases, such as complex queries, aggregate functions, and cardinality constraints.

#### 3.4.1 Dealing with Complex Queries

Note that SQL queries embedded in the program code could be very complex. For example, they may involve nested sub-queries with aggregation functions, union, distinct, and group-by views, etc. The syntax of SQL queries is defined in the ISO standardization\(^2\). The fundamental structure of a SQL query is a query block, which consists of `SELECT`, `FROM`, `WHERE`, GROUP BY, and `HAVING` clauses. If a predicate or some predicates in the `WHERE` or `HAVING` clause are of the form \([C_k \lor Q]\) where `Q` is also a query block, the query is a nested query. A large body of work [4, 7] on query transformation in databases has been explored to unnest complex queries into equivalent single level canonical queries. Researchers showed that almost all types of sub-queries can be unnested except those that are correlated to non-parents, whose correlations appear in disjunction, or some ALL subqueries with multi-item connecting condition containing null-valued columns.

---

Generally, there are two types of canonical queries: DPNF with the WHERE clause consisting of a disjunction of conjunctions as shown in Figure 7, and CPNF with the WHERE clause consisting of a conjunction of disjunctions (such as \((A1 \text{ OR}... \text{ OR} A1n)\) AND \(\ldots\) AND \((A1 \text{ OR}... \text{ OR} A1m))\). Note that DPNF and CPNF can be transformed mutually using DeMorgan's rules\(^1\). For a canonical query in DPNF or CPNF, SynDB can handle it well because we have mapped the logical relations between the predicates in the WHERE clause to normal program code. We are thus able to correctly express the original logical conditions from the WHERE clause using program logical connectives.

### 3.4.2 Dealing with Aggregate Functions

An SQL aggregate function returns a single value, calculated from values in a column (e.g., AVG(), MAX(), MIN(), COUNT(), and SUM()). It often comes in conjunction with a GROUP BY clause that groups the result set by one or more columns.

In general, we map these aggregate functions to be calculations on the SynSqlDataReader object. Recall that for the SynSqlDataReader object that represents a query’s returned result set, its field DataTable resultSet contains qualified rows selected by a SELECT statement. From these rows, we form groups according to the GROUP BY clause. We form the groups by sorting the rows in DataTable resultSet based on the attributes indicated in the GROUP BY clause. We discard all groups that do not satisfy the conditions in the HAVING clause. We then apply the aggregate functions to each group and retrieve values for the aggregations listed in the SELECT clause.

Another special case that we would like to point out is, in the SELECT clause, it is permitted to contain calculations among multiple database attributes. For example, suppose that there are two new attributes checkingBalance and savingBalance in the mortgage table. In the SELECT clause, we have a selected item calculated as mortgage.checkingBalance + mortgage.savingBalance. In our approach, dealing with such a complex case is still consistent with how to deal with the aforementioned SELECT statement. From the field DataTable resultSet in the SynSqlDataReader object, we merge the columns involving in this selected item using the indicated calculation. For example, we get a merged column by making an “add” calculation on the two related columns mortgage.checkingBalance and mortgage.savingBalance. We also set the data type of the merged column as the calculation result’s data type.

### 3.4.3 Dealing with Cardinality Constraints

Program logic could be far more complex than our illustrative example. Cardinality constraints for generating a sufficient database state may come from the query-result-manipulation code. Since SynDB is a DSE-based test generation approach, the space-explosion issue in path exploration still exists, especially after the query result is returned.

Consider the example code in Figure 8. Inside the while loop after the result set is returned, a variable count is updated every time when a condition balance > 50000 is satisfied. Then, outside the while loop, branch conditions in Lines 10a and 10c depend on the values of count. Manually, we can observe that the value of count depends on how many records satisfy the branch condition in Line 09b. We may generate enough database records so that branches in Lines 10a and 10c could be entered. However, since there is a while loop, applying DSE to hunt for enough database records (thus covering Lines 10a) faces significant challenges: the size of results can range to a very large number, of which perhaps only

```
09: while (results.Read()) {
09a:   int balance = results.GetInt(1);
09b:   if (balance > 50000)
10:      count++;
10a:   if (count > 10)
10b:      return count;
10c:   else
11:      return 10;
```

Figure 8: An example where cardinality constraints come from the query result manipulation

A small number of records can satisfy the condition in Line 09b. Hence, this problem is reduced to a traditional issue [18]: to explore a program that contains one or more branches with relational conditions (here, we have \((\text{count} > 10))\) where the operands are scalar values (integers or floating-point numbers) computed based on control-flow decisions connected to program inputs through data flow (here, we have \(\text{if} (\text{balance} > 50000) \text{ count}++\)).

In the literature, Xie et al. proposed an approach Fitnex [18] that uses a fitness function to measure how close an already discovered feasible path is to a particular test target. Each already explored path is assigned with a fitness value. Then a fitness gain for each branch is computed and the approach gives higher priority to flipping a branching node with a better fitness gain. The fitness function measures how close the evaluation at runtime is to covering a target predicate.

Under the scenario of our approach, since we have built the consistency between the database state and the returned result set, we can capture the relationship between the database state and the target conditions (such as Lines 10a and 10c) depending on the returned result set. We apply the search strategy that integrates the Fitnex approach [18], so that generating enough database records with high efficiency becomes feasible. For the example code in Figure 8, we detect that covering the path condition in Line 10a is dependant on covering the path condition in Line 09b. To satisfy the target predicate in Line 10a, the search strategy would give priority to flip the branching node in Line 09b. This step therefore helps achieve generating a sufficient database state with high efficiency.

### 4. EVALUATION

Our approach replaces the original database API methods with synthesized database interactions. We also treat the associated database state as a program input to guide DSE to collect constraints for generating both program inputs and corresponding database records. Through this way, tests generated by our approach are able to achieve high code coverage for testing database applications. In our evaluation, we seek to evaluate the performance of our approach from the following perspectives:

**RQ1**: What is the percentage increase in code coverage by the tests generated by our approach compared to the tests generated by existing approaches [12, 16] in testing database applications?

**RQ2**: What is the analysis cost for our approach to generate tests?

### 4.1 Subject Applications

We conduct an empirical evaluation on two open source database applications: RiskIt\(^3\) and UnixUsage\(^4\). RiskIt is an insurance quote application that makes estimation based on users’ personal

\(^1\)http://en.wikipedia.org/wiki/DeMorgan’s_laws
\(^3\)https://riskitinsurance.svn.sourceforge.net
\(^4\)http://sourceforge.net/projects/se549unixusage
information, such as zipcode and income. It has an existing database containing 13 tables, 57 attributes, and more than 1.2 million records. UnixUsage is an application to obtain statistics about how users interact with the Unix systems using different commands. It has a database containing 8 tables, 31 attributes, and more than 0.25 million records. Since our approach is able to conduct database state generation from the scratch, we do not need to make use of the existing database records. Both applications were written in Java. To test them with the Pex DSE engine, we convert the original Java source code into C# code using a tool called Java2CSharpTranslator\(^6\). The detailed evaluation subjects and results can be found on our project website\(^7\).

For these two applications, we focus on the methods whose SQL queries are constructed dynamically. The RiskIt application consists of 44 classes, of which 32 methods are found to have at least one SQL query. Within these 32 methods, 17 methods contain queries whose variables are data-dependent on program inputs. We choose these 17 methods to conduct our evaluation. The UnixUsage application consists of 26 classes, of which 76 methods are found to have at least one SQL query. Within these 76 methods, we choose 22 methods that contain queries whose variables are data-dependent on program inputs to conduct our evaluation.

### 4.2 Evaluation Setup

The two applications have predefined their own schemas for the associated databases in attached .sql files. However, we observe that the predefined database schema constraints are quite simple, such as primary key constraints and data type constraints. To better evaluate the effectiveness of our approach, we extend the existing database schema constraints by adding extra constraints. We choose certain attributes from the tables and augment their constraints. The added extra constraints are ensured, as much as possible, to be reasonable and consistent with real world settings. For example, for the RiskIt application, we add a length constraint to the attribute ZIP from the userrecord table to ensure that the length of ZIP must be 5. Similarly, we ensure that the value of the attribute EDUCATION from the education table must be chosen from the set \{high school, college, graduate\}. The details of the added extra constraints are listed in Table 4.

We next implement code transformation on the original program code under test. As aforementioned, our approach constructs synthesized database states based on the schemas (e.g., attribute names and data types) and incorporates the database schema constraints into normal program code by checking these constraints on the synthesized database states. We then apply Pex on the transformed code to conduct test generation.

Initially, for each method under test, the output of Pex’s execution on the transformed code is saved in a methodname.g.cs file consisting of a number of generated tests. To investigate RQ1, we intend to directly measure the code coverage on the original program code under test. We conduct the measurements in the following way. From those methodname.g.cs files, we first populate the generated records back into the real database. To do so, we instrument code at the end of each methodname.g.cs file. The instrumented code builds connections with the real database, constructs INSERT queries for each table, and runs the INSERT queries. Second, we construct new tests using the program inputs generated by Pex’s execution on the transformed code. Note that these program inputs have also been saved in the methodname.g.cs files. Third, we run the constructed new tests for the original program under test interacting with the real database to measure the code coverage. We record the statistics of the code coverage, including total program blocks, covered blocks, and coverage percentages.

We choose one method filterZipcode from the RiskIt application to illustrate the evaluation process. After the code transformation, we get a new method SynfilterZipcode shown in Figure 9. We next run Pex on the transformed code SynfilterZipcode to conduct test generation. The generated tests are then saved in a SynfilterZipcode.g.cs file. For example, one of the tests is shown in Figure 10 (Lines 01-21). Running this test covers the path where Line 09 = true, and Line 12 = true in Figure 9. For the test in Figure 10, the generated database record is shown in Lines 09-13 and the corresponding program inputs for method arguments zip and dbStateRiskIt are shown in Line 20. The last statement (Line 21) makes an assertion and completes the current test. After the assertion, we instrument auxiliary code to populate the generated records back to the real database. We build a connection with the real database and insert the records to corresponding tables (pseudocode in Lines 22-28). Then, we construct new tests for the original code under test using the program inputs contained in tests generated by Pex. For example, based on the input values in Line 20 of the test shown in Figure 10, we construct a new test shown in Figure 11. We run these new tests and then

---

**Table 4: Added extra constraints on RiskIt and UnixUsage**

<table>
<thead>
<tr>
<th>Application</th>
<th>Table</th>
<th>Attribute</th>
<th>Original constraints</th>
<th>Added constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>RiskIt</td>
<td>education</td>
<td>EDUCATION</td>
<td>char(50) ∈ {high school, college, graduate}</td>
<td></td>
</tr>
<tr>
<td>RiskIt</td>
<td>job</td>
<td>SSN</td>
<td>int, NOT NULL, primary key</td>
<td>[000000001, 999999999]</td>
</tr>
<tr>
<td>RiskIt</td>
<td>userrecord</td>
<td>SSN</td>
<td>int, NOT NULL, primary key</td>
<td>[000000001, 999999999]</td>
</tr>
<tr>
<td>RiskIt</td>
<td>userrecord</td>
<td>ZIP</td>
<td>varchar(50)</td>
<td></td>
</tr>
<tr>
<td>RiskIt</td>
<td>userrecord</td>
<td>MARITAL</td>
<td>varchar(50) ∈ {single, married, divorced, widow}</td>
<td></td>
</tr>
<tr>
<td>UnixUsage</td>
<td>COURSE_INFO</td>
<td>COURSE_ID</td>
<td>int, NOT NULL, primary key</td>
<td>[100,999]</td>
</tr>
<tr>
<td>UnixUsage</td>
<td>DEPT_INFO</td>
<td>RACE</td>
<td>varchar(50) ∈ {white, black, asian, hispanic}</td>
<td></td>
</tr>
<tr>
<td>UnixUsage</td>
<td>TRANSCRIPT</td>
<td>USER_ID</td>
<td>varchar(50) NOT NULL</td>
<td>[000000001, 999999999]</td>
</tr>
</tbody>
</table>

---

\(^6\)http://sourceforge.net/projects/j2cstranslator/

\(^7\)http://www.sis.uncc.edu/$\sim$xwu/DBGen
01: [TestMethod]
02: [PexGeneratedBy(typeof(SynMethodTestRiskIt))]
03: public void SynfilterZipcode101()
04: List<Userrecord> list;
05: UserrecordTable userrecordTable;
06: int i;
07: Userrecord[] userrecords = new Userrecord[1];
08: Userrecord s0 = new Userrecord();
09: s0.CITIZENSHIP = (string)null;
10: s0.NAME = "";
11: s0.ZIP = "10001";
12: ...
13: s0.CITIZENSHIP = (string)null;
14: userrecords[0] = s0;
15: list = new List<Userrecord>
    (){([IEnumerable<Userrecord>]userrecords);}
16: userrecordTable = new UserrecordTable();
17: userrecordTable.UserrecordList = list;
18: dbStateRiskIt sl = new dbStateRiskIt();
19: s1.userrecordTable = userrecordTable;
20: i = this.SynfilterZipcode("10001", s1);
21: Assert.AreEqual<int>(1, i);
22: //Code instrumentation for records insertion (pseudocode)
23: SqlConnection conn = new SqlConnection();
24: for each table t in s1
25: if t.count > 0
26: for each record r in t
27: string query = INSERT INTO t VALUES (r)
28: conn.run(query)}

Figure 10: Tests generated by Pex on SynfilterZipcode


4.3 Results

We report the evaluation results in Tables 5 and 6, from the perspectives of code coverage and cost. The evaluation is conducted on a machine with hardware configuration Intel Pentium 4 CPU 3.0 GHz, 2.0 GB Memory and OS Windows XP SP2.

4.3.1 Code Coverage

In Tables 5 and 6, the first part (Columns 1-2) shows the index and method names. The second part (Columns 3-6) shows the code coverage result. Column 3 “total blocks”) shows the total number of blocks in each method. Columns 4-6 “covered blocks”) show the number of covered blocks using tests generated by our approach, the number of covered blocks using tests generated by existing approaches, and the percentage increase, respectively. Note that our approach does not deal with generating program inputs and database states to cause runtime database connection exceptions. Thus, the code blocks related to these exceptions (e.g., the catch statements) cannot be covered.

Within the RiskIt application, the 17 methods contain 943 code blocks in total. Tests generated by existing approaches cover 672 blocks while our approach can cover 871 blocks. Within the UnixUsage application, the 22 methods contain 336 code blocks in total. Tests generated by existing approaches cover 238 blocks while our approach can also cover the whole 336 blocks.

We observe that tests generated by existing approaches fail to cover certain blocks for some methods. The reason is that the generated records violate the database schema constraints. When populating such records back into the real database, the insertion operations are rejected by the database. Take the aforementioned example method SynfilterZipcode shown in Figure 9 to illustrate such cases. Our simulated results show that existing approaches are able to generate a record with a value “\"0\"” for the ZIP field. However, the value “\"0\"” does not satisfy the database schema constraint where ZIP.length = 5 as shown in Table 4. Thus, the real database refuses the insertion of this record. As a result, correspondingly, running the tests generated by existing approaches cannot retrieve effective records from the database and fails to cover certain blocks (e.g., the while loop for the query-result iteration).

Since we add only a small number of extra database schema constraints, the results show that existing approaches achieve the same code coverage as our approach does for some methods. For example, for the No.2 method filterOccupation in the RiskIt application shown in Table 5, we did not add any other constraints to the associated tables. The result shows that, for the total 41 blocks, both existing approaches and our approach can cover 37 blocks while the remaining not-covered blocks are related to handling runtime exceptions. Note that the number of added extra constraints in our evaluation is limited. In practice, applications could contain more complex constraints. In that case, we expect that our approach can achieve much better coverage than existing approaches.

Another observation that we would like to point out is on complex constraints involving multiple attributes and multiple tables. For example, for the No.12 method filterEstimatedIncome, the program input String getIncome appears in a branch condition involving a mathematical formula comparing with a complex calculation using the query’s returned result. The complex calculation derives a value from multiple attributes (workweeks, weekwage, capitalGains, capitalLosses, and stockDividends) across multiple tables (data tables job and investment). Recall that our approach is able to capture complex constraints defined at the schema level. For this method, if an extra complex constraint is defined for these attributes at the schema level, we expect that our approach can achieve much better coverage than existing ap-
Table 5: Evaluation results on RiskIt

<table>
<thead>
<tr>
<th>No.</th>
<th>method</th>
<th>total covered (blocks)</th>
<th>covered (blocks)</th>
<th>analysis time of SynDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>getAllZipcode</td>
<td>39</td>
<td>37</td>
<td>2m40s</td>
</tr>
<tr>
<td>2</td>
<td>filterOccupation</td>
<td>41</td>
<td>37</td>
<td>1m18s</td>
</tr>
<tr>
<td>3</td>
<td>filterZipcode</td>
<td>42</td>
<td>38</td>
<td>1m20s</td>
</tr>
<tr>
<td>4</td>
<td>filterEducation</td>
<td>41</td>
<td>37</td>
<td>1m28s</td>
</tr>
<tr>
<td>5</td>
<td>filterMaritalStatus</td>
<td>41</td>
<td>37</td>
<td>1m31s</td>
</tr>
<tr>
<td>6</td>
<td>findTopIndustryCode</td>
<td>19</td>
<td>14</td>
<td>0s</td>
</tr>
<tr>
<td>7</td>
<td>findTopOccupationCode</td>
<td>19</td>
<td>14</td>
<td>0s</td>
</tr>
<tr>
<td>8</td>
<td>updateStability</td>
<td>79</td>
<td>75</td>
<td>3m22s</td>
</tr>
<tr>
<td>9</td>
<td>userinformation</td>
<td>61</td>
<td>57</td>
<td>3m45s</td>
</tr>
<tr>
<td>10</td>
<td>updateTable</td>
<td>60</td>
<td>56</td>
<td>3m19s</td>
</tr>
<tr>
<td>11</td>
<td>updateAgeTable</td>
<td>52</td>
<td>48</td>
<td>3m28s</td>
</tr>
<tr>
<td>12</td>
<td>filterEstimatedIncome</td>
<td>58</td>
<td>54</td>
<td>2m57s</td>
</tr>
<tr>
<td>13</td>
<td>calculateUnemploymentRate</td>
<td>49</td>
<td>45</td>
<td>2m45s</td>
</tr>
<tr>
<td>14</td>
<td>calculateScore</td>
<td>93</td>
<td>87</td>
<td>2m11s</td>
</tr>
<tr>
<td>15</td>
<td>getValues</td>
<td>107</td>
<td>99</td>
<td>3m25s</td>
</tr>
<tr>
<td>16</td>
<td>getOneZipcode</td>
<td>34</td>
<td>23</td>
<td>1m35s</td>
</tr>
<tr>
<td>17</td>
<td>browseUserProperties</td>
<td>108</td>
<td>104</td>
<td>4m21s</td>
</tr>
<tr>
<td>all methods (total)</td>
<td>943</td>
<td>871</td>
<td>4m23s</td>
<td></td>
</tr>
</tbody>
</table>

4.3.2 Analysis Cost

We observe that the major factor that impacts the analysis time for test generation is the complexity of the query embedded in a method. If a query joins multiple tables, the exploration of checking database schema constraints for each table is linearly increased. Meanwhile, if a table contains a large number of attributes, high cost is also incurred. Complexity of the qualification in a query also influences the analysis time as evaluating the conditions has been transformed into normal program code in our approach.

We report the analysis cost of our approach in Tables 5 and 6. Column 7 shows the analysis time for each method. For example, the analysis time for methods in RiskIt varies from less than 1 minute to about 5 minutes, while the analysis time for methods in UnixUsage varies from less than 1 minute to about 3 minutes.

Another observation that we would like to point out is related to Pex’s path exploration. As aforementioned, we evaluate the qualification in a query by transforming it into normal program code. For example, the qualification in a query is expressed by a boolean combination of conditions connected by program logical connectives. A generated record that satisfies the whole qualification should satisfy all the conditions. However, when Pex explores a branch, it neglects to explore any subsequent boolean condition but starts a new run, if it finds that the first condition does not hold. Thus, to make all the conditions true, Pex takes more runs, whose number is linear to the number of conditions. In practice, to improve the efficiency, we may force Pex to consider all the conditions together at one time. This step could be implemented by marking that branch as an entirety if we find conditions in that branch come from the qualification of the same query.

5. RELATED WORK

Testing database applications is attracting increasing attention recently. With the focus on functional testing, test generation is a fundamental technique. Emmi et al. [12] develop an approach for extending DSE to consider query constraints. The approach generates tests consisting of both program inputs and database states for Java applications. However, it is required that the associated database should be in place. Taneja et al. [16] develop the MODA framework that is applicable when the database is not available by using a mock database. Our approach focuses on the problem faced by these two approaches: they generate database records based on query constraints from the concrete query and these query constraints may conflict with other constraints. Our approach correlates various constraints within a database application. Some other approaches also leverage DSE as a major supporting technique for testing database applications. A recent approach [13] uses DSE to generate database states to achieve advanced structural coverage criteria. Li and Csalog [11] propose an approach to exploit existing databases to maximize the coverage under DSE.

The AGENDA approach [9] deals with how to generate tests that satisfy some basic database integrity constraints. The approach does not handle parametric queries or constraints on query results, which are however very common in practice. The QAGen [6] approach extends symbolic execution using symbolic query processing to generate query-aware databases. However, QAGen mainly deals with isolated queries and considers only the cardinality constraints. Our approach focuses on the context of application programs. Binnig et al. [5] propose the approach of Reverse Query Processing (RQP) that considers cases where the query-execution result is given. Although RQP can be applied in application programs, it still lacks the ability to deal with complex program logic where the constraints derived from concrete queries are infeasible. Another approach [14] focuses on program-input generation given an existing database state. Using the intermediate information gathered by the DSE process, the approach constructs auxiliary queries and executes them on the existing database state to attain effective program inputs. In contrast, our approach mainly generates database states from the scratch.

Other than achieving high code coverage that aims to expose program faults, some techniques [8, 19] focus on mutation testing for database applications. Our approach can be extended to satisfy other testing requirements as long as we have correlated various constraints within a database application. For performance testing, the PPGen approach [17] generates mock databases by reproducing the statistical distribution of realistic database states. However, PPGen assumes that constraints are explicit and focuses on SQL workload’s performance testing. Our approach can generate database records and can be extended to estimate the performance of a database application by specifying various distribution properties.

6. CONCLUSION AND FUTURE WORK

In this paper, we propose a DSE-based approach called SynDB for testing database applications. The approach synthesizes new
database interactions to replace the original ones. Through this way, we bridge the gap between program-execution constraints and environment constraints. Existing test-generation techniques treat the database as an external component and may face problems when considering constraints within a database application in an insufficient way. Our approach considers both query constraints and database schema constraints, and transform them to normal program code. We use a state-of-the-art DSE engine called Pex to generate effective tests consisting of both program inputs and database states. Empirical evaluations show that our approach achieves higher program code coverage than existing approaches.

In future work, we plan to extend our approach to various phases of functional testing. We plan to investigate the problem of locating logical faults in database applications using our approach. For example, there could be inherent constraint conflicts within an application caused by careless developers. We plan to apply our approach on more complex application contexts such as multiple queries. We also plan to investigate how to apply our approach on generating a large number of database records.

### 7. REFERENCES


