Generating Program Inputs for Database Application Testing

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

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

***Abstract*—Testing is essential for quality assurance of database applications. Achieving high code coverage of the database application is important in testing. In practice, there may exist a copy of live databases that can be used for database application testing. Using an existing database state is desirable since it tends to be representative of real-world objects' characteristics, helping detect faults that could cause failures in real-world settings. However, to cover a specific program code portion (e.g., block), appropriate program inputs also need to be generated for the given existing database state. To address this issue, in this paper, we propose a novel approach that generates program inputs for achieving high code coverage of a database application, given an existing database state. Our approach uses symbolic execution to track how program inputs are transformed before appearing in the executed SQL queries and how the constraints on query results affect the application's execution. One significant challenge in our problem context is the gap between program-input constraints derived from the program and from the given existing database state; satisfying both types of constraints is needed to cover a specific program code portion. Our approach includes novel query formulation to bridge this gap. Our approach is loosely integrated into Pex, a state-of-the-art white-box testing tool for .NET from Microsoft Research. Empirical evaluations on two real database applications show that our approach assists Pex to generate program inputs that achieve higher code coverage than the program inputs generated by Pex without our approach's assistance.**

I. Introduction

Database applications are ubiquitous, and it is critical to assure high quality of database applications. To assure high quality of database applications, testing is commonly used in practice. Testing database applications can be classified as functional testing, performance testing (load and stress, scalability), security testing, environment and compatibility testing, and usability testing. Among them, functional testing aims to verify the functionality of the code under test. An important task of functional testing is to generate test inputs to achieve full or at least high code coverage, such as block or branch coverage of the database application under test. For database applications, test inputs include both program inputs and database states.

A. Illustrative Example

The example code snippet shown in Figure 1 includes a portion of C# source code from a database application that calculates some statistic related to mortgages. The corresponding database contains two tables: `customer` and `mortgage`. Their schema-level descriptions and constraints are given in Table I. The `calcStat` method described in the example code snippet receives two program inputs: `type` that determines the years of mortgages and `zip` that indicates the zip codes of customers. A variable `fzip` is calculated from `zip` and in our example `fzip` is given as "`zip`+1". Then the database connection is set up (Lines 08-10). The database query is constructed (Line 11) and executed (Lines 12 and 13). The tuples from the returned result set are iterated (Lines 14-20). For each tuple, a variable `diff` is calculated from the values of the `income` field and the `balance` field. If `diff` is greater than `100000`, a counter variable `count` is increased (Line 19) and `totalBalance` is updated (Line 20). The method finally returns the calculation result.

```
01:public int calcStat(int type,int zip) {
02:  int years = 0, count = 0, totalBalance = 0;
03:  int fzip = zip + 1;
04:  if (type == 0)
05:     years = 15;
06:  else
07:     years = 30;
08:  SqlConnection sc = new SqlConnection();
09:  sc.ConnectionString = "..";
10:  sc.Open();
11:  string query = "SELECT C.SSN, C.income,"
    +" M.balance FROM customer C, mortgage M"
    +" WHERE M.year='" + years +"' AND"
    +" C.zipcode='"+ fzip + "' AND C.SSN = M.SSN";
12:  SqlCommand cmd = new SqlCommand(query, sc);
13:  SqlDataReader results = cmd.ExecuteReader();
14:  while (results.Read()){
15:     int income = int.Parse(results["income"]);
16:     int balance = int.Parse(results["balance"]);
17:     int diff = income - 1.5 * balance;
18:     if (diff > 100000){
19:        count++;
20:        totalBalance = totalBalance + balance;}}
21:  return totalBalance;}
```

Fig. 1. An example code snippet from a database application under test

Both program inputs (i.e., input parameters) and database states are crucial in testing this database application because (1) the program inputs determine the embedded SQL statement in Line 11; (2) the database states determine whether the true branch in Line 14 and/or the true branch in Line 18 can be covered, being crucial to functional testing, because covering

TABLE I
DATABASE SCHEMA

customer table			mortgage table		
Attribute	Type	Constraint	Attribute	Type	Constraint
SSN	Int	Primary Key	SSN	Int	Primary Key
zipcode	String	[1, 99999]			Foreign Key
name	Int		year	Int	
gender	String				
age	Int	(0, 100)	balance	Int	(1000, Max)
income	Int				

a branch is necessary to expose a potential fault within that branch; (3) the database states also determine how many times the loop body in Lines 14-20 is executed, being crucial to performance testing.

B. Problem Formalization

In practice, there may exist a copy of live databases that can be used for database application testing. Using an existing database state is desirable since it tends to be representative of real-world objects' characteristics, helping detect faults that could cause failures in real-world settings. However, it often happens that a given database with an existing database state (even with millions of records) returns no records (or returned records do not satisfy branch conditions in the subsequently executed program code) when the database receives and executes a query with arbitrarily chosen program input values. For example, method `calcStat` takes both `type` and `zip` as inputs. To cover a path where conditions at Lines 14 and 18 are both `true`, we need to assign appropriate values to variables `years` and `fzip` so that the execution of the SQL statement in Line 12 with the query string in Line 11 will return non-empty records, while at the same time attributes `income` and `balance` of the returned records also satisfy the condition in Line 18. Since the domain for program input `zip` is large, it is very likely that, if a tester enters an arbitrary `zip` value, execution of the query on the existing database will return no records, or those returned records do not satisfy the condition in Line 18. Hence, it is crucial to generate program input values such that test inputs with these values can help cover various code portions when executed on the existing database.

C. Proposed Solution

To address this issue, in this paper, we propose a novel approach that generates program inputs for achieving high code coverage of a database application, given an existing database state. In our approach, we first examine close relationships among program inputs, program variables, branch conditions, embedded SQL queries, and database states. For example, program variables used in the executed queries may be derived from program inputs via complex chains of computations (we use `fzip=zip+1` in our illustrative example) and path conditions involve comparisons with record values in the query's result set (we use `if (diff>100000)`) in our illustrative example). We then automatically generate appropriate program inputs via executing a formulated auxiliary query on the given database state.

In particular, our approach uses dynamic symbolic execution (DSE) [7] to track how program inputs to the database application under test are transformed before appearing in the executed queries and how the constraints on query results affect the later program execution. We use DSE to collect various intermediate information.

Our approach addresses one significant challenge in our problem context: there exists a gap between program-input constraints derived from the program and those derived from the given existing database state; satisfying both types of constraints is needed to cover a specific program code portion. During DSE, these two types of constraints cannot be naturally collected, integrated, or solved for test generation. To address this challenge, our approach includes novel query formulation to bridge this gap. In particular, based on the intermediate information collected during DSE, our approach automatically constructs new auxiliary queries from the SQL queries embedded in code under test. The constructed auxiliary queries use those database attributes related with program inputs as the target selection and incorporate those path constraints related with query result sets into selection condition. After the new auxiliary queries are executed against the given database, we attain effective program input values for achieving code coverage.

This paper makes the following main contributions:

- The first problem formalization for program-input generation given an existing database state to achieve high code coverage.
- A novel program-input-generation approach based on symbolic execution and query formulation for bridging the gap between program-input constraints from the program and from the given existing database state.
- Evaluations on two real database applications to assess the effectiveness of our approach upon Pex [12], a state-of-the-art white-box testing tool for .NET from Microsoft Research. Empirical results show that our approach assists Pex to generate program inputs that achieve higher code coverage than the program inputs generated by Pex without our approach's assistance.

II. DYNAMIC SYMBOLIC EXECUTION IN DATABASE APPLICATION TESTING

Recently, *dynamic symbolic execution* [7] (DSE) was proposed for test generation. DSE first starts with default or arbitrary 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 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 feeds 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

TABLE II
A GIVEN DATABASE STATE

customer table						mortgage table		
SSN	zipcode	name	gender	age	income	SSN	year	balance
001	27695	Alice	female	35	50000	001	15	20000
002	28223	Bob	male	40	150000	002	15	30000

have been explored or the number of explored paths has reached the predefined upper bound.

DSE has also been used in testing database applications [6], [11]. Emmi et al. [6] developed an approach for automatic test generation based on DSE. Their approach uses a constraint solver to solve collected symbolic constraints to generate both program input values and corresponding database records. The approach involves running the program simultaneously on concrete program inputs as well as on symbolic inputs and a symbolic database. In the first run, the approach uses random concrete program input values, 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 can cover the current path. To explore a new path, it flips a branch condition and generates new program input values and corresponding database records. However, their approach cannot generate effective program inputs based on the content of an existing database state. The reason is that some program inputs (e.g., `zip` in our illustrative example) appear only in the embedded SQL queries and there is no path constraint over them.

Our approach differs from Emmi et al.'s approach [6] in that we leverage DSE as a supporting technique to generate effective program input values by executing constructed auxiliary queries against the existing database state. As a result, high code coverage of the application can be achieved without generating new database states. When DSE is applied on a database application, DSE often fails to cover specific branches due to an insufficient returned result set because returned record values from the database often involve in deciding later branches to take. We use Pex [12], a DSE tool for .NET, to illustrate how our approach assists DSE to determine program input values such that the executed query can return sufficient records to cover various code portions. During the program execution, DSE maintains the symbolic expressions for all variables. When the execution along one path terminates, DSE tools such as Pex have collected all the preceding path constraints to form the path condition. Pex also provides a set of APIs that help access intermediate information of its DSE process. For illustration purposes, we assume that we have an existing database state shown in Table II for our preceding example shown in Figure 1.

To run the program for the first time against the existing database state, Pex uses default values for program inputs `type` and `zip`. In this example, because `type` and `zip` are both integers. Pex simply chooses "`type`=0, `zip`=0" as default values. The condition in Line 04 is then satisfied and the query statement with the content in Line 11 is dynamically constructed. In Line 12 where the query is executed, we can dynamically get the concrete query string as

```
Q1: SELECT C.SSN, C.income, M.balance
    FROM customer C, mortgage M
    WHERE M.year=15 AND C.zipcode=1 AND C.SSN=M.SSN
```

Through static analysis, we can also get `Q1`'s corresponding abstract form as

```
Q1abs: SELECT C.SSN, C.income, M.balance
       FROM customer C, mortgage M
       WHERE M.year=: years AND C.zipcode=: fzip
             AND C.SSN=M.SSN
```

The execution of `Q1` on Table II yields zero record. Thus, the `while` loop body in Lines 14-20 is not entered and the exploration of the current path is finished. We use the Pex API method `PexSymbolicValue.GetPathConditionString()` after Line 14 to get the path condition along this path:

```
P1:(type == 0) && (results.Read() != true)
```

To explore a new path, Pex flips a part of the current path condition from "`type == 0`" to "`type != 0`" and generates new program inputs as "`type=1, zip=0`". The condition in Line 04 is then not satisfied and the SQL statement in Line 11 is dynamically determined as

```
Q2: SELECT C.SSN, C.income, M.balance
    FROM customer C, mortgage M
    WHERE M.year=30 AND C.zipcode=1 AND C.SSN=M.SSN
```

Note that here we have the same abstract form for Q2 as for Q1. However, the execution of `Q2` still returns zero record, and hence the execution cannot enter the `while` loop body either. The path condition for this path is

```
P2:(type == 1) && (results.Read() != true)
```

We can see that at this point no matter how Pex flips the current collected path condition, it fails to explore any new paths. Since Pex has no knowledge about the `zipcode` distribution in the database state, using the arbitrarily chosen program input values often incurs zero returned record when the query is executed against the existing database state. As a result, none of paths involving the `while` loop body could be explored.

In testing database applications, previous test-generation approaches (e.g., Emmi et al. [6]) then invoke constraint solvers to generate new records and instantiate a new test database state, rather than using the given existing database state, required in our focused problem.

In contrast, by looking into the existing database state as shown in Table II, we can see that if we use an input like "`type=0, zip=27694`", the execution of the query in Line 11 will yield one record {`C.SSN = 001, C.income = 50000, M.balance = 20000`}, which further makes Line 14 condition `true` and Line 18 condition `false`. Therefore, using the existing database state, we are still able to explore this new path:

```
P3:(type == 0) && (results.Read() == true)
   &&(diff <= 100000)
```

Furthermore, if we use "type=0, zip=28222", the execution of the query in Line 11 will yield another record {C.SSN = 002, C.income = 150000, M.balance = 30000}, which will make both Line 14 condition and Line 18 condition true. Therefore, we can explore this new path:

```
P4:(type == 0) && (results.Read() == true)
   &&(diff > 100000)
```

In Section III, we present our approach that can assist Pex to determine appropriate program input values such that high code coverage can be achieved using the existing database state.

III. Approach

Our approach assists Pex to determine appropriate program inputs so that high code coverage can be achieved in database application testing. As illustrated in our example, not-covered branches or paths are usually caused by the empty returned result set (e.g., for path P1) or insufficient returned records that cannot satisfy later executed conditions (e.g., for path P3).

The major idea of our approach is to construct an auxiliary query based on the intermediate information (i.e., the executed query's concrete string and its abstract form, symbolic expressions of program variables, and path conditions) collected by DSE. There are two major challenges here. First, program input values are often combined into the executed query after a chain of computations. In our illustrative example, we simply set fzip = zip+1 in Line 3 to represent this scenario. We can see that fzip is contained in the WHERE clause of the executed query and zip is one program input. Second, record values in the returned query result set are often directly or indirectly (via a chain of computations) involved in the path condition. In our illustrative example, the program variable diff in the branch condition diff>100000 (Line 18) is calculated from the retrieved values of attributes income and balance. To satisfy the condition (e.g., diff>100000 in Line 18), we need to make sure that the program input values determined by our auxiliary query are appropriate so that the query's return records are sufficient for satisfying later executed branch conditions.

A. Auxiliary Query Construction

Algorithm 1 illustrates how to construct an auxiliary query. The algorithm accepts as inputs a simple SQL query in its both concrete and abstract forms, program input values, and the current path condition.

Formally, suppose that a program takes a set of parameters I = $\{I_1, I_2, ..., I_k\}$ as program inputs. During path exploration, DSE flips a branch condition pc_s (e.g., one executed after the query execution) from the false branch to the true branch to cover a target path. Such flipping derives a new constraint or path condition for the target path as PC = $pc_1 \wedge pc_2 \wedge ... \wedge pc_s$. DSE feeds this constraint to the constraint solver to generate a new test input, whose later execution, however, does not cover the true branch of pc_s as planned, likely due to database interactions along the path. In the path exploration, DSE also keeps records of all program variables and their concrete and symbolic expressions in the program along this path when DSE reaches pc_s. From the records, we determine program variables V = $\{V_1, V_2, ..., V_t\}$ that are data-dependent on program inputs I. DSE also collects the concrete string of an executed query along the current path. In our approach, we assume the SQL query takes the form:

```
SELECT  C1, C2, ..., Ch
FROM    from-list
WHERE   A1 AND A2 ... AND An
```

In the SELECT clause, there is a list of h strings where each may correspond to a column name or with arithmetic or string expressions over column names and constants following the SQL syntax. In the FROM clause, there is a from-list that consists of a list of tables. We assume that the WHERE clause contains n predicates, A = $\{A_1, A_2, ..., A_n\}$, connected by $n-1$ "AND"s. Each predicate A_i 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. Note that here we assume that the WHERE clause contains only conjunctions using the logical connective "AND". We discuss how to process complex SQL queries in Section III-C. Some predicate expressions in the WHERE clause of Q may involve comparisons with program variables. From the corresponding abstract query Q_{abs}, we check whether each predicate A_i contains any program variables from V.

We take the path P3 (Line 04 true, Line 14 true, and Line 18 false) in our preceding example shown in Figure 1 to illustrate the idea. The program input set is I = {type, zip} and the path condition PC is

```
P3:(type == 0) && (results.Read() == true)
   &&(diff <= 100000)
```

The program variable set V is {type,zip,fzip}. When flipping the condition diff<=100000, Pex fails to generate satisfiable test inputs for the flipped condition diff > 100000. The abstract form is shown as

```
Qabs:  SELECT C.SSN, C.income, M.balance
       FROM  customer C, mortgage M
       WHERE M.year=: years AND
             C.zipcode=: fzip AND C.SSN=M.SSN
```

We can see that the predicate set A in the WHERE clause is formed as {M.year=:years, C.zipcode=:fzip, C.SSN=M.SSN}. Predicates M.year=:years and C.zipcode=:fzip contain program variables years and fzip, respectively. Furthermore, the program variable fzip is contained in V. In other words, the predicate C.zipcode=:fzip involves comparisons with program inputs.

Algorithm 1 shows our procedure to construct the auxiliary query $\tilde{Q}$ based on the executed query (Q's concrete string and

Algorithm 1 Auxiliary Query Construction

Input: a canonical query Q, Q's abstract form Q_{abs}, program input set $I = \{I_1, I_2, ..., I_k\}$, path condition $PC = pc_1 \wedge pc_2 \wedge ... \wedge pc_s$
Output: an auxiliary query $\tilde{Q}$

1: Find variables $V = \{V_1, V_2, ..., V_t\}$ data-dependent on I;
2: Decompose Q_{abs} with a SQL parser for each clause;
3: Construct a predicate set $A = \{A_1, A_2, ..., A_n\}$ from Q's WHERE clause;
4: Construct an empty predicate set $\tilde{A}$, an empty attribute set C_V, and an empty query $\tilde{Q}$;
5: **for** each predicate $A_i \in A$ **do**
6: **if** A_i does not contain program variables **then**
7: Leave A_i unmodified and check the next predicate;
8: **else**
9: **if** A_i does not contain program variables from V **then**
10: Substitute A_i's program variables with their corresponding concrete values in Q;
11: **else**
12: Substitute the variables from V with the expression expressed by I;
13: Substitute the variables not from V with their corresponding concrete values in Q;
14: Copy A_i to $\tilde{A}$;
15: Add A_i's associated database attributes to C_V;
16: **end if**
17: **end if**
18: **end for**
19: Append C_V to $\tilde{Q}$'s SELECT clause;
20: Copy Q's FROM clause to $\tilde{Q}$'s FROM clause;
21: Append $A - \tilde{A}$ to $\tilde{Q}$'s WHERE clause;
22: Find variables $U = \{U_1, U_2, ..., U_u\}$ coming directly from Q's result set;
23: Find U's corresponding database attributes $C_U = \{C_{U1}, C_{U2}, ..., C_{Uw}\}$;
24: **for** each branch condition $pc_i \in PC$ after Q's execution **do**
25: **if** pc_i contains variables data-dependent on U **then**
26: Substitute the variables in pc_i with the expression expressed by the variables from U;
27: Substitute the variables from U in pc_i with U's corresponding database attributes in C_U;
28: Add the branch condition in pc_i to $\tilde{PC}$;
29: **end if**
30: **end for**
31: Flip the last branch condition in $\tilde{PC}$;
32: Append all the branch conditions in $\tilde{PC}$ to $\tilde{Q}$'s WHERE clause;
33: **return** $\tilde{Q}$;

its abstract form Q_{abs}) and the intermediate information collected by DSE. Lines 5-21 present how to construct the clauses (SELECT, FROM, and WHERE) of the auxiliary query $\tilde{Q}$. We decompose Q_{abs} using a SQL parser[1] and get its n predicates $A = \{A_1, A_2, ..., A_n\}$ from the WHERE clause. We construct an empty predicate set $\tilde{A}$. For each predicate $A_i \in A$, we check whether A_i contains program variables. If not, we leave A_i unchanged and check the next predicate. If yes, we then check whether any contained program variable comes from the set V. If no program variables in the predicate are from V, we substitute them with their corresponding concrete values in Q. In our example, the predicate `M.year=:years` belongs to this category. We retrieve the concrete value of `years` from Q and the predicate expression is changed as `M.year=15`. If some program variables contained in the predicate come from V, we substitute them with their symbolic expressions (expressed by the program inputs in I), substitute all the other program variables that are not from V with their corresponding concrete values in Q and copy the predicate A_i to $\tilde{A}$. The predicate `C.zipcode=:fzip` in our example belongs to this category. We replace `fzip` with `zip+1` and the new predicate becomes `C.zipcode=:zip+1`. We also add A_i's associated database attributes into a temporary attribute set C_V. Those attributes will be included in the SELECT clause of the auxiliary query $\tilde{Q}$. For the predicate `C.zipcode=:fzip`, the attribute `C.zipcode` is added to $\tilde{A}$ and is also added in the SELECT clause of the auxiliary query $\tilde{Q}$.

After processing all the predicates in A, we get an attribute set $C_V = \{C_{V1}, C_{V2}, ..., C_{Vj}\}$ and a predicate set $\tilde{A} = \{\tilde{A}_1, \tilde{A}_2, ..., \tilde{A}_l\}$. Note that here all the predicates in $\tilde{A}$ are still connected by the logical connective "AND". The attributes from C_V form the attribute list of the $\tilde{Q}$'s SELECT clause. All the predicates in A - $\tilde{A}$ connected by "AND" form the predicates in the $\tilde{Q}$'s WHERE clause. Note that the `from-list` of the $\tilde{Q}$'s FROM clause is the same as that of Q. In our example, $\tilde{A}$ is `C.zipcode=:zip+1`, $A - \tilde{A}$ is `M.year=15 AND C.SSN=M.SSN`, and the attribute set C_V is `C.zipcode`. The constructed auxiliary query $\tilde{Q}$ has the form:

```
SELECT  C.zipcode
FROM    customer C, mortgage M
WHERE   M.year=15 AND C.SSN=M.SSN
```

When executing the preceding auxiliary query against the existing database state, we get two `zipcode` values, `27695` and `28223`. The corresponding program input `zip` can take either `27694` or `28222` because of the constraint {`C.zipcode=:zip+1`} in our example. A test input with the program input either "`type=0, zip=27694`" or "`type=0, zip=28222`" can guarantee that the program execution enters the `while` loop body in Lines 14-20. However, there is no guarantee that the returned record values satisfy later executed branch conditions. For example, if we choose "`type=0, zip=27694`" as the program input, the execution can enter the `while` loop body but still fails to satisfy the branch condition (i.e., `diff>100000`) in Line 18. Hence it is imperative to incorporate constraints from later branch conditions into the constructed auxiliary query.

Program variables in branch condition $pc_i \in PC$ after executing the query may be data-dependent on returned record values. In our example, the value of program variable `diff` in branch condition "`diff > 100000`" is derived from the values of the two variables `income`, `balance` that correspond to the values of attributes `C.income`, `M.balance` of returned records. Lines 22-32 in Algorithm 1 show how to incorporate later branch conditions in constructing the WHERE clause of the auxiliary query.

Formally, we get the set of program variables $U = \{U_1, U_2, ..., U_w\}$ that directly retrieve the values from the query's

[1] http://zql.sourceforge.net/

Algorithm 2 Program Input Generation

Input: an auxiliary query $\tilde{Q}$, program inputs I
intermediate results C_V and $\tilde{A}$ from Algorithm 1
Output: program input values R for I

1: Execute $\tilde{Q}$ against the given database, get resulting values R_V for the attributes in C_V;
2: Substitute the attributes C_V for predicates in $\tilde{A}$ with the values in R_V, resulting in new predicates in $\tilde{A}$;
3: Feed the new predicates in $\tilde{A}$ to a constraint solver and get final values R for I;
4: **return** Output final program input values R;

returned result set, and treat them as symbolic inputs. For each program variable U_i, we also keep its corresponding database attribute C_{Ui}. Note that here C_{Ui} must come from the columns in the SELECT clause. We save them in the set C_U = {C_{U1}, C_{U2}, ..., C_{Uw}}. For each branch condition $pc_i \in PC$, we check whether any program variables in pc_i are data-dependent on variables in U. If yes, we substitute such variables in pc_i with their symbolic expressions with respect to the symbolic input variables from U and replace each U_i in pc_i with its corresponding database attribute C_{Ui}. The modified pc_i is then appended to the $\tilde{Q}$'s WHERE clause. In our example, the modified branch condition `C.income-1.5*M.balance>100000` is appended to the WHERE clause, and the new auxiliary query is

```
SELECT  C.zipcode
FROM    customer C, mortgage M
WHERE   M.year=15 AND C.SSN=M.SSN AND
        C.income - 1.5 * M.balance > 100000
```

When executing the preceding auxiliary query against the existing database state, we get the `zipcode` value as "28223". Having the constraint `C.zipcode=:zip+1`, input "`type=0, zip=28222`" can guarantee that the program execution enters the true branch in Line 18.

Program Input Generation. Note that executing the auxiliary query $\tilde{Q}$ against the database returns a set of values R_V for attributes in C_V. Each attribute in C_V can be traced back to some program variable in V = {V_1, V_2, ..., V_t}. Recall that V contains program variables that are data-dependent on program inputs I. Our final goal is to derive the values for program inputs I. Recall in Algorithm 1, we already collected in the predicate set $\tilde{A}$ the symbolic expressions of C_V with respect to program inputs I. After substituting the attributes C_V with their corresponding concrete values in R_V resulted from executing $\tilde{Q}$ against the given database, we have new predicates in $\tilde{A}$ for program inputs I. We then feed these new predicates in $\tilde{A}$ to a constraint solver to derive the values for program inputs I. We give our pseudo procedure in Algorithm 2.

In our illustrative example, after executing our auxiliary query on Table II, we get a returned value "28223" for the attribute `C.zipcode`. In $\tilde{A}$, we have `C.zipcode=:zip+1`. After substituting `C.zipcode` in `C.zipcode=:zip+1` with the value "28223", we have `28223=:zip+1`. The value "28222" for the program input `zip` can then be derived by invoking a constraint solver.

In our prototype, we use the constraint solver Z3[2] integrated in Pex. Z3 is a high-performance theorem prover being developed at Microsoft Research. The constraint solver Z3 supports linear real and integer arithmetic, fixed-size bit-vectors, extensional arrays, uninterpreted functions, and quantifiers. In practice, the result R could be a set of values. For example, the execution of the auxiliary query returns a set of satisfying zip code values. If multiple program input values are needed, we can repeat the same constraint solving process to produce each returned value in R.

B. Dealing with Aggregate Calculation

Up to now, we have investigated how to generate program inputs through auxiliary query construction. Our algorithm exploits the relationships among program inputs, program variables, executed queries, and path conditions in source code. Database applications often deal with more than one returned record. In many database applications, multiple records are iterated from the query's returned result set. Program variables that retrieve values from the returned result set further take part in aggregate calculations. The aggregate values then are used in the path condition. In this section, we discuss how to capture the desirable aggregate constraints on the result set returned for one or more specific queries issued from a database application. These constraints play a key role in testing database applications but previous work [3], [5] on generating database states has often not taken them into account.

Consider the following code after the query's returned result set has been iterated in our preceding example shown in Figure 1:

```
...
14:  while (results.Read()){
15:     int income = int.Parse(results["income"]);
16:     int balance = int.Parse(results["balance"]);
17:     int diff = income - 1.5 * balance;
18:     if (diff > 100000){
19:        count++;
20:        totalBalance = totalBalance + balance;}}
20a: if (totalBalance > 500000)
20b:  do other calculation...
21:  return ...;}
```

Here, the program variable `totalBalance` is data-dependent on the variable `balance` and thus is associated with the database attribute `M.balance`. The variable `totalBalance` is involved in a branch condition `totalBalance > 500000` in Line 20a. Note that the variable `totalBalance` is aggregated from all returned record values. For simple aggregate calculations (e.g., sum, count, average, minimum, and maximum), we are able to incorporate the constraints from the branch condition in our auxiliary query formulation. Our idea is to extend the auxiliary query with the GROUP BY and HAVING clauses. For example,

[2]http://research.microsoft.com/en-us/um/redmond/projects/z3/

we learn that the variable `totalBalance` is a summation of all the values from the attribute `M.balance`. The variable `totalBalance` can be transformed into an aggregation function `sum(M.balance)`. We include `C.zipcode` in the GROUP By clause and `sum(M.balance)` in the HAVING clause of the extended auxiliary query:

```
SELECT C.zipcode,sum(M.balance)
FROM  customer C, mortgage M
WHERE M.year=15 AND C.SSN=M.SSN
      AND C.income - 1.5 * M.balance > 100000
GROUP BY C.zipcode
HAVING sum(M.balance) > 500000
```

Cardinality Constraints. In many database applications, we often require the number of returned records to meet some conditions (e.g., for performance testing). For example, after execution reaches Line 20, we may have another piece of code appended to Line 20 as

```
20c:  if (count >= 3)
20d:    computeSomething();
```

Here we can use a special DSE technique [8] for dealing with input-dependent loops. With this technique, we can learn that the subpath with the conditions in Lines 14 and 18 being `true` has to be invoked at least three times in order to cover the branch condition `count >= 3` in Line 20c. Hence we need to have at least three records iterated into Line 18 so that true branches of Lines 14, 18, and 20c can be covered. In our auxiliary query, we can simply add `COUNT(*) >= 3` in the HAVING clause to capture this cardinality constraint.

```
SELECT   C.zipcode
FROM     customer C, mortgage M
WHERE    M.year=15 AND C.SSN=M.SSN
         AND C.income - 1.5 * M.balance > 100000
GROUP BY C.zipcode
HAVING   COUNT(*) >= 3
```

Program logic could be far more complex than the appended code in Lines 20a-d of our example. We emphasize here that our approach up to now works for only aggregate calculations that are supported by the SQL built-in aggregate functions. When the logic iterating the result set becomes more complex than SQL's support, we cannot directly determine the appropriate values for program inputs. For example, some zipcode values returned by our auxiliary query could not be used to cover the true branch of Lines 20a-b because the returned records with the input zipcode values may fail to satisfy the complex aggregate condition in Line 20a. However, our approach can still provide a super set of valid program input values. Naively, we could iterate all the candidate program input values to see whether some of them can cover a specific branch or path.

C. Dealing with Complex Queries

SQL queries embedded in application program code could be very complex. For example, they may involve nested subqueries with aggregation functions, union, distinct, and group-by views, etc. 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 op Q] where Q is also a query block, the query is a *nested query*. A large body of work exists on query transformation in databases. Various decorrelation techniques (e.g., [4], [9]) have been explored to unnest complex queries into equivalent single level canonical queries and recent work [1] showed that almost all types of subqueries can be unnested.

Generally, there are two types of canonical queries: DPNF with the WHERE clause consisting of a disjunction of conjunctions as shown below

```
SELECT  C1, C2, ..., Ch
FROM    from-list
WHERE   (A11 AND ... AND A1n) OR ...
        OR (Am1 AND ... AND Amn)
```

and CPNF with the WHERE clause consisting of a conjunction of disjunctions (such as `(A11 OR... OR A1n) AND ... AND (Am1 OR... OR Amn)`). Note that DPNF and CPNF can be transformed mutually using DeMorgan's rules[3].

Next we present our algorithm on how to formulate auxiliary queries and determine program input values given a general DPNF query. Our previous Algorithm 1 deals with only a special case of DPNF where the query's WHERE clause contains only one `A11 AND ... AND A1n`. We show the algorithm details in Algorithm 3. Our idea is to decompose the DPNF query Q_{dpnf} into m simple queries Q_i $(i = 1, \cdots, m)$. The WHERE clause of each Q_i contains only one disjunction in the canonical form, `Ai1 AND ... AND Ain`. We apply Algorithm 1 to generate its corresponding auxiliary query $\tilde{Q}_i$ and apply Algorithm 2 to generate program input values R_i. The union of R_is then contains all appropriate program input values.

Algorithm 3 Program Input Generation for DPNF Query

Input: a DPNF query Q_{dpnf}, program inputs I
Output: program input value set R_{dpnf} for I

1: **for** each disjunction D_i in Q_{dpnf}'s WHERE clause **do**
2: Build an empty query Q_i;
3: Append Q_{dpnf}'s SELECT clause to Q_i's SELECT clause;
4: Append Q_{dpnf}'s FROM clause to Q_i's FROM clause;
5: Append D_i to Q_i's WHERE clause;
6: Apply Algorithm 1 on Q_i and get its auxiliary query $\tilde{Q}_i$;
7: Apply Algorithm 2 on $\tilde{Q}_i$ and get output R_i;
8: $R_{dpnf} = R_{dpnf} \cup R_i$;
9: **end for**
10: **return** Output final program input values R_{dpnf};

IV. Evaluation

Our approach can provide assistance to DSE-based test-generation tools (e.g., Pex [12] for .NET) to improve code coverage in database application testing. In our evaluation, we seek to evaluate the benefit and cost of our approach from the following two perspectives:

[3]http://en.wikipedia.org/wiki/DeMorgan'slaws

TABLE III
EVALUATION RESULTS ON RISKIT

No.	method	total (blocks)	covered(blocks)			runs		time(seconds)	
			Pex	Pex+ours	increase	Pex	ours	Pex	ours
1	getAllZipcode	39	17	37	51.28%	12	3	time out	21.4
2	filterOccupation	41	27	37	24.39%	18	4	time out	34.3
3	filterZipcode	42	28	38	23.81%	76	4	42.3	25.7
4	filterEducation	41	27	37	24.39%	76	4	time out	27.6
5	filterMaritalStatus	41	27	37	24.39%	18	4	48.5	29.4
6	findTopIndustryCode	19	13	14	5.26%	32	4	time out	29.2
7	findTopOccupationCode	19	13	14	5.26%	81	5	time out	23.3
8	updatestability	79	61	75	17.72%	95	6	time out	23.4
9	userinformation	61	40	57	27.87%	37	3	62.4	20.8
10	updatetable	60	42	56	23.33%	42	3	67.8	20.9
11	updatewagetable	52	42	48	11.54%	75	8	time out	27.8
12	filterEstimatedIncome	58	44	54	17.24%	105	8	time out	23.6
13	calculateUnemploymentRate	49	45	45	0.00%	89	7	time out	23.7
14	calculateScore	93	16	87	76.35%	92	10	time out	23.3
15	getValues	107	38	99	57.01%	182	43	time out	42.7
16	getOneZipcode	34	23	32	26.47%	22	6	time out	39.1
17	browseUserProperties	108	85	104	17.60%	83	9	time out	81.1
all methods (total)		943	588	871	25.52%	1135	131	1781.0	517.3

RQ1: What is the percentage increase in code coverage by the program inputs generated by Pex with our approach's assistance compared to the program inputs generated without our approach's assistance in testing database applications?
RQ2: What is the cost of our approach's assistance?

In our evaluation, we first run Pex without our approach's assistance to generate test inputs. We record their statistics of code coverage, including total program blocks, covered blocks, and coverage percentages. In our evaluation, we also record the number of runs and execution time. A run represents one time that one path is explored by Pex using a set of program input values. Because of the large or infinite number of paths in the code under test, Pex uses exploration bounds to make sure that Pex terminates after a reasonable amount of time. For example, the bound TimeOut denotes the number of seconds after which the exploration stops. In our evaluation, we use the default value TimeOut=120s and use "time out" to indicate timeout cases.

Pex often fails to generate test inputs to satisfy or cover branch conditions that are data-dependent on the query's execution or its returned result set. We then perform our algorithms to construct auxiliary queries based on the intermediate information collected from Pex's previous exploration. We then execute the auxiliary queries against the existing database and generate new test inputs. We then run the test inputs previously generated by Pex and the new test inputs generated by our approach, and then record new statistics.

We conduct an empirical evaluation on two open source database applications: `RiskIt`[4] and `UnixUsage`[5]. `RiskIt` is an insurance quote application that makes estimation based on users' personal 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. Both applications were written in Java. To test them in the Pex environment, we convert the 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].

A. Code coverage

We show the evaluation results in Table III and Table IV. For each table, 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 by Pex without our approach's assistance, the number of covered blocks by Pex together with our approach's assistance, and the percentage increase, respectively.

Within the `RiskIt` application, 17 methods are found to contain program inputs related with database attributes. These 17 methods contain 943 code blocks in total. Test inputs generated by Pex without our approach's assistance cover 588 blocks while Pex with our approach's assistance covers 871 blocks. In fact, Pex with our approach's assistance can cover all branches except those branches related to exception handling. For example, the method No. 1 contains 39 blocks in total. Pex without our approach's assistance covers 17 blocks while Pex with our approach's assistance covers 37 blocks. The two not-covered blocks belong to the `catch` statements, which mainly deal with exceptions at runtime.

[4]https://riskitinsurance.svn.sourceforge.net
[5]http://sourceforge.net/projects/se549unixusage
[6]http://sourceforge.net/projects/j2cstranslator/
[7]http://www.sis.uncc.edu/∼xwu/DBGen

TABLE IV
EVALUATION RESULTS ON UNIXUSAGE

No.	method	total (blocks)	covered(blocks)			runs		time(seconds)	
			Pex	Pex+ours	increase	Pex	ours	Pex	ours
1	courseNameExists	7	6	7	14.29%	17	3	32.2	20.0
2	getCourseIDByName	10	6	10	40.00%	14	3	29.9	20.0
3	computeFileToNetworkRatio ForCourseAndSessions	25	8	25	68.00%	35	7	time out	24.9
4	outputUserName	14	9	14	35.71%	18	4	38.3	20.5
5	deptNameExists	13	9	13	30.77%	18	3	43.3	20.0
6	computeBeforeAfterRatioByDept	24	8	24	66.67%	109	8	77.5	22.0
7	getDepartmentIDByName	11	7	11	36.36%	92	3	time out	20.0
8	computeFileToNetworkRatioForDept	21	20	21	4.76%	33	6	time out	21.5
9	officeNameExists	11	7	11	36.36%	18	3	41.4	20.0
10	getOfficeIdByName	9	5	9	44.44%	18	3	51.3	20.0
11	raceExists	11	7	11	36.36%	18	3	32.1	20.0
12	userIdExists(version1)	11	7	11	36.36%	18	3	40.3	20.0
13	transcriptExist	11	7	11	36.36%	18	3	39.9	20.0
14	getTranscript	6	5	6	16.67%	14	2	33.7	20.0
15	commandExists(version1)	10	6	10	40.00%	14	2	36.0	20.0
16	categoryExists	11	7	11	36.36%	18	3	33.3	20.0
17	getCategoryByCommand	8	5	8	37.50%	17	2	34.1	20.0
18	getCommandsByCategory	10	6	10	40.00%	17	2	38.4	20.0
19	getUnixCommand	6	5	6	16.67%	17	2	40.3	20.0
20	retrieveUsageHistoriesById	21	7	21	66.67%	86	3	58.4	27.2
21	userIdExists(version2)	11	7	11	36.36%	19	3	32.7	20.0
22	commandExists(version2)	11	7	11	36.36%	21	3	36.5	20.0
23	retrieveMaxLineNo	10	7	10	30.00%	53	3	time out	22.3
24	retrieveMaxSequenceNo	10	7	10	30.00%	35	3	time out	20.1
25	getSharedCommandCategory	11	7	11	36.36%	118	3	time out	20.4
26	getUserInfoBy	47	15	47	68.09%	153	4	time out	20.0
27	doesUserIdExist	10	9	10	10.00%	74	2	41.3	20.0
28	getPrinterUsage	34	27	34	20.59%	115	4	67.2	20.6
all methods (total)		394	258	394	34.52%	1197	93	1718.1	579.5

The `UnixUsage` application contains 28 methods whose program inputs are related with database attributes, with 394 code blocks in total. Pex without our approach's assistance covers 258 blocks while Pex with our approach's assistance covers all 394 blocks. The `UnixUsage` application constructs a connection with the database in a separate class that none of these 28 methods belong to. Thus, failing to generate inputs that can cause runtime database connection exceptions has not been reflected when testing these 28 methods.

B. Cost

In Tables III and IV, the third part (Columns 7-10) shows the cost. Columns 7 and 9 "Pex" show the number of runs and the execution time used by Pex without our approach's assistance. We notice that, for both applications, Pex often terminates with "time out". The reason is that Pex often fails to enter the loops of iterating the returned result records. Columns 8 and 10 "ours" show the additional number of runs by Pex with assistance of our approach and the extra execution time (i.e., the time of constructing auxiliary queries, deriving program input values by executing auxiliary queries against the existing database, and running new test inputs) incurred by our approach.

We observe that, for both applications, Pex with assistance of our approach achieves much higher code coverage with relatively low additional cost of a few runs and a small amount of extra execution time. In our evaluation, we set the TimeOut as 120 seconds. For those "time out" methods, Pex could not achieve new code coverage even given larger TimeOut values. Our approach could effectively help cover new branches not covered by Pex with relatively low cost.

Note that in our current evaluation, we loosely integrate Pex and our approach: we perform our algorithms only after Pex finishes its previous exploration (i.e., after applying Pex without our approach's assistance) since our algorithms rely on the intermediate information collected during Pex's exploration. We expect that after our approach is tightly integrated into Pex, our approach can effectively reduce the overall cost of Pex integrated with our approach (which is currently the sum of the time in Columns 9 and 10). In such tight integration, our algorithms can be triggered automatically when Pex fails to generate test inputs to satisfy branch conditions that are data-dependent on a query's execution or its returned result set.

V. RELATED WORK

Database application testing has attracted much attention recently. The AGENDA project [5] addressed how to generate

test inputs to satisfy basic database integrity constraints and does not consider parametric queries or constraints on query results during input generation. One problem with AGENDA is that it cannot guarantee that executing the test query on the generated database states can produce the desired query results. Willmor and Embury [14] developed an approach that builds a database state for each test case intensionally, in which the user provides a query that specifies the pre- and post-conditions for the test case. Binnig et al. [2] also extended symbolic execution and used symbolic query processing to generate some query-aware databases. However, the test database states generated by their QAGen prototype system [2] mainly aim to be used in database management systems (DBMS) testing.

Emmi et al. [6] developed an approach for automatic test generation for a database application. Their approach is based on DSE and uses symbolic constraints in conjunction with a constraint solver to generate both program inputs and database states. We developed an approach [13] that leverages DSE to generate database states to achieve advanced structural coverage criteria. In this paper, we focus on program-input generation given an existing database state, avoiding the high overhead of generating new database states during test generation. Li and Csallner [10] considered a similar scenario, i.e., how to exploit existing databases to maximize the coverage under DSE. However, their approach constructs a new query by analyzing the current query, the result tuples, the covered and not-covered paths, and the satisfied and unsatisfied branch conditions. It can neither capture the close relationship between program inputs and results of SQL queries, nor generate program inputs to maximize code coverage.

VI. Conclusions

In this paper, we have presented an approach that takes database applications and a given database as input, and generates appropriate program input values to achieve high code coverage. In our approach, we employ dynamic symbolic execution to analyze the code under test and formulate auxiliary queries based on extracted constraints to generate program input values. Empirical evaluations on two open source database applications showed that our approach can assist Pex, a state-of-the-art DSE tool, to generate program inputs that achieve higher code coverage than the program inputs generated by Pex without our approach's assistance. In our future work, we plan to extend our technique to construct auxiliary queries directly from embedded complex queries (e.g., nested queries), rather than from their transformed norm forms.

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-0915400 for Tao Xie.

References

[1] R. Ahmed, A. Lee, A. Witkowski, D. Das, H. Su, M. Zait, and T. Cruanes. Cost-based query transformation in Oracle. In *VLDB*, pages 1026–1036, 2006.

[2] C. Binnig, D. Kossmann, E. Lo, and M.T. Ozsu. QAGen: generating query-aware test databases. In *SIGMOD*, pages 341–352, 2007.

[3] D. Chays and J. Shahid. Query-based test generation for database applications. In *DBTest*, pages 01–06, 2008.

[4] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB*, pages 197–208, 1987.

[5] Y. Deng, P. Frankl, and D. Chays. Testing database transactions with agenda. In *ICSE*, pages 78–87, 2005.

[6] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA*, pages 151–162, 2007.

[7] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[8] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *ISSTA*, pages 23–33, 2011.

[9] W. Kim. On optimizing an SQL-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, 1982.

[10] C. Li and C. Csallner. Dynamic symbolic database application testing. In *DBTest*, pages 01–06, 2010.

[11] K. Taneja, Y. Zhang, and T. Xie. MODA: Automated test generation for database applications via mock objects. In *ASE*, pages 289–292, 2010.

[12] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *TAP*, pages 134–153, 2008.

[13] K. Pan, X. Wu, and T. Xie. Database state generation via dynamic symbolic execution for coverage criteria. In *DBTest*, pages 01–06, 2011.

[14] D. Willmor and S. M. Embury. An intensional approach to the specification of test cases for database applications. In *ICSE*, pages 102–111, 2006.