Database State Generation via Dynamic Symbolic Execution for Coverage Criteria

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

Automatically generating sufficient database states is imperative to reduce human efforts in testing database applications. Complementing the traditional block or branch coverage, we develop an approach that generates database states to achieve advanced code coverage including boundary value coverage(BVC) and logical coverage(LC) for source code under test. In our approach, we leverage dynamic symbolic execution to examine close relationships among host variables, embedded SQL query statements, and branch conditions in source code. We then derive constraints such that data satisfying those constraints can achieve the target coverage criteria. We implement our approach upon Pex, which is a state-of-the-art DSE-based test-generation tool for .NET. Empirical evaluations on two real database applications show that our approach assists Pex to generate test database states that can effectively achieve both BVC and LC, complementing the block or branch coverage.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution, testing tools*

General Terms

Algorithms, Design, Reliability, Verification

Keywords

dynamic symbolic execution, database state generation, coverage criteria

1. INTRODUCTION

Database application testing has attracted much attention recently in both academia and industry. Generating database states as well as test inputs to achieve high branch coverage of source code has been widely studied [5, 7, 12, 16]. An effective approach for automatic test-input generation based on *Dynamic Symbolic Execution* (DSE) was proposed for testing database applications [1, 8, 12, 15] and various tools for different languages have been developed (e.g., C [15], Java [12], and C# [1]). DSE uses symbolic constraints in conjunction with a constraint solver to generate database states. The approach involves running the program simultaneously on default or random program inputs as well as on symbolic inputs and a symbolic database. When DSE is applied on database applications, the symbolic execution generates path constraints over the symbolic program inputs along the execution path and then generates database constraints over the symbolic database by symbolically tracking the concrete SQL queries executed along the execution path [12].

In this paper, we leverage DSE as a supporting technique to generate database states to achieve advanced coverage criteria in addition to branch coverage. In database applications, close relationships exist among program inputs, host variables, branch conditions, embedded SQL queries, and database states. For example, program inputs and host variables often appear in the embedded SQL queries and branch conditions in source code after executing SQL queries are often logical expressions that involve comparisons with retrieved values from database states. It is imperative to enforce advanced structural coverage criteria such as *Logical Coverage* (LC) and *Boundary Value Coverage* (BVC) for effective testing. In particular, BVC requires to execute programs using values from both the input range and boundary conditions and requires multiple test inputs at boundaries [10]. The reason is that errors tend to occur at extreme or boundary points. LC criteria involve instantiating clauses in a logical expression with concrete truth values. Researchers have focused on active clause coverage criteria to construct a test such that the value of a logical expression is directly dependent on the value of the clause that we want to test. Among these active clause coverage criteria, the *Correlated Active Clause Coverage* (CACC) [2] is equivalent to *masking Modified Condition/Decision Coverage* (MC/DC), of which the MC/DC has been chosen by US Federal Aviation Administration [4] as a recommended test-generation criterion among logical criteria.

1.1 Illustrating Example

The example in Figure 1 shows a portion of C# source code from a database application that counts the number of mortgage customers according to their profiles. The corresponding database contains two tables: `customer` and `mortgage`. Their schema-level descriptions and constraints are given in Table 1. The `calcStat` method described in the example code receives two parameters: `type` that determines the years of mortgages, `inputAge` that determines the age from customer profiles. The database query is then constructed dynamically (Lines 07). If the input age is greater than 25, a variable `fAge` is computed with `fAge=inputAge+10` and the query string is updated (Lines 08-10). We use the expression `fAge=inputAge+10` to illustrate that host variables appearing in the executed queries may be derived from program inputs

DBTest '11 June 13, 2011 Athens, Greece

```
01: public int calcStat(int type, int inputAge) {
02:  int years = 0, count = 0;
03:  if (type == 0)
04:     years = 15;
05:  else
06:     years = 30;
07:  string query = "SELECT C.SSN,C.jobStatus,
     C.marriage,M.balance FROM customer C,mortgage M
     WHERE M.year='" + years + "' AND C.SSN = M.SSN";
08:  if (inputAge > 25){
09:     fAge = inputAge + 10;
10:     query = query + " AND C.age='" + fAge +"'";}
11:  SqlConnection sc = new SqlConnection();
12:  sc.ConnectionString = "..";
13:  sc.Open();
14:  SqlCommand cmd = new SqlCommand(query, sc);
15:  SqlDataReader results = cmd.ExecuteReader();
16:  while (results.Read()){
17:     int bal = int.Parse(results["balance"]);
18:     bool employed = bool.Parse(results["jobStatus"]);
19:     bool married = bool.Parse(results["marriage"]);
20:     if (bal >= 250000||employed && married){
21:        count++;}
22:     else {...}}
23:  return count;}
```

Figure 1: An example code snippet from a database application under test

or other host variables via complex chains of computations. Then the database connection is set up (Lines 11-13) and the constructed query is executed (Line 14). The tuples from the returned result set are iterated (Lines 16-22). For each tuple, if the value of the `balance` field is greater than or equal to `250000`, or if the customer is both employed and married, a counter variable `count` is increased by one (Line 21). Otherwise, the program does other computations. The method finally returns the calculation result.

Table 1: Database schema

customer table			mortgage table		
Attribute	Type	Constraint	Attribute	Type	Constraint
SSN	Int	Primary Key	SSN	Int	Primary Key Foreign Key
name	String				
age	Int	age > 0	year	Int	
income	Int				
jobStatus	bool		balance	Int	balance > 1000
marriage	bool				

1.2 Problem Formalization

To test the program `calcStat` in the preceding example or the entire database application, we need to generate sufficient database states as well as desirable values for program inputs. The input parameters often determine the embedded SQL statement in Line 14 and the database states determine whether the branches in Lines 16, 20, and 22 can be entered. Our approach uses Dynamic Symbolic Execution(DSE) [8, 15] to track how the inputs to the program under test are transformed before appearing in the executed queries and how the constraints on query results affect the later program execution. We use Pex [1], a state-of-the-art DSE tool for .NET to illustrate our idea.

During its execution, Pex maintains the symbolic expressions for all variables. DSE involves running the program simultaneously on default or random inputs and some initial database state as well as on symbolic inputs and a symbolic database. The symbolic execution generates path constraints over the symbolic program inputs along the execution path and then generates database constraints over the symbolic database by symbolically tracking the concrete SQL queries executed along the execution path.

When the execution along one path terminates, Pex has collected all the preceding path constraints to form the path condition. Pex also provides a set of APIs that help access the intermediate information of its DSE process[1]. For example, with `type = 0, inputAge = 30` as input values, for the path P where branch conditions in Lines 03, 08, 16, and 20 are *true*.

To satisfy the branch condition in Line 20, we need to generate a sufficient database state such that the execution of the embedded query returns sufficient records to satisfy the branch condition in Line 20. Approaches [12] have been proposed to generate both program inputs and suitable database states to cover a feasible path, including the executions depending on the executed query's returned result set.

Our work focuses on how to generate sufficient database states to satisfy advanced coverage criteria including BVC and CACC criteria. For example, the variable `inputAge` is indirectly involved in the embedded SQL through variable `fage` and is also involved in the branch condition of Line 8, `inputAge > 25`. The boundary values from where the range starts and ends are imperative for testing critical domains. The branch condition `(bal >= 250000 || employed && married) == true` involves a predicate with three clauses, and variables `bal`, `employed`, and `married` retrieve values from the attribute values in the returned query result set. We need to generate sufficient data records in the database such that the branch condition can be evaluated as true and false, respectively. Furthermore, to achieve CACC criteria, we need to generate data records such that the value of this logical expression is directly dependent on the value of a particular clause that we want to test.

2. APPROACH

2.1 Overview of Our Approach

We present our approach to generating database states such that the executed query can return sufficient records to satisfy coverage criteria. Algorithm 1 shows details about our approach. The algorithm is invoked when the DSE process encounters one branch condition that either contains host variables data-dependent on the attributes of the query's returned result set (directly or after chains of computations) or is related with accessing the query-returned result set. We treat the access to the query-returned result set as a query execution point.

Pex has provided a package of APIs that help users fetch intermediate results inside its DSE process[2]. We mainly use the methods from the class `PexSymbolicValue` (shortened as `PexS` thereafter). We insert a call to API method `PexS.ToString()` at each execution point to get the target query string. We call `PexS.GetRelevantInputNames<Type>()` to detect the data dependency between this execution point and program input parameters. To retrieve the path condition at the execution point, we call the method `PexS.GetPathConditionString()`.

Throughout this section, we use path P (where branch conditions in Lines 03, 08, 16, and 20 are *true*) and program inputs (`type = 0, inputAge = 30`) to illustrate our algorithm. When the DSE process encounters one branch condition, we call the Pex API method `PexS.GetPathConditionString()` to get the path

[1]http://research.microsoft.com/en-us/projects/pex/

[2]http://research.microsoft.com/en-us/projects/pex/

Algorithm 1 Database State Generation in Achieving CACC and BVC

Input: database schema S, schema level constraint C_S
path condition $PC = pc_1 \wedge pc_2 \wedge ... \wedge pc_s$
Output: database state D

1: Find query execution point T, get concrete query Q and its symbolic expression Q_{sym};
2: Decompose Q and Q_{sym} with a SQL parser;
3: Create a constraint set C_Q = {(A_{11} AND ... AND A_{1n}) OR ... OR (A_{m1} AND ... AND A_{mn})} from Q_{sym}'s WHERE clause;
4: Create a variable set $V_Q = \varnothing$;
5: **for** each $A_{ij} \in C_Q$ **do**
6: **if** A_{ij} contains any host variable v **then**
7: **if** v is related with branch conditions before T **then**
8: Add v to V_Q;
9: Replace v with its symbolic expression expressed by variables in the related branch conditions;
10: **else**
11: Replace v with its corresponding concrete value in the concrete query Q;
12: **end if**
13: **end if**
14: **end for**
15: **for** each branch condition $pc \in PC$ before T **do**
16: **if** pc contains variables related with V_Q **then**
17: I_{pc} = *EnforceCriteria*(pc, $true$); [Algorithm 2]
18: $C_Q = C_Q \times I_{pc}$;
19: **end if**
20: **end for**
21: Create a variable set V_R that contains variables directly dependent on attributes $C_1, C_2, ..., C_h$;
22: Create a constraint set C_R = {1=1};
23: **for** each branch condition $pc \in PC$ after T **do**
24: **if** pc contains variables in or dependent on V_R **then**
25: I_{pc} = *EnforceCriteria*(pc, $true$); [Algorithm 2]
26: Replace the variables expressed by V_R in I_{pc} with their corresponding database attributes;
27: $C_R = C_R \times I_{pc}$;
28: **end if**
29: **end for**
30: Create a constraint set $C = C_Q \times C_R$;
31: Call a constraint solver to instantiate C and C_S, get database state D;
32: **return** Database state D;

condition along this path. In our example, we get $PC = pc_1 \wedge pc_2 \wedge pc_3 \wedge pc_4$, where pc_1 = `(type == 0)`, pc_2 = `(inputAge > 25)`, pc_3 = `(results.Read() == true)`, and pc_4 = `((bal >= 250000 || employed && married) == true)`. Some branch conditions are related with the database state via the dynamically constructed embedded query or the query's returned result set. In our algorithm, we call the Pex API method `PexS.ToString(...)` to get the concrete executed query string Q and the symbolic query string Q_{sym}. We decompose Q_{sym} using the SQL parser[3] and get its clauses. We assume the embedded SQL query takes the canonical DPNF form [4]:

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

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. In the WHERE clause, there is a disjunction of conjunctions. Each condition (e.g., A11) 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. In practice, the queries could be very complex such as sub-queries can appear in the WHERE clause. There are extensive studies [9] on how to map complex queries such as nested queries to their canonical forms.

In our example, we have the concrete query Q

```
SELECT C.SSN, C.jobStatus, C.marriage, M.balance
FROM customer C, mortgage M
WHERE M.year=15 AND C.SSN=M.SSN And C.age=40
```

and its corresponding symbolic string Q_{sym}

```
SELECT C.SSN, C.jobStatus, C.marriage, M.balance
FROM customer C, mortgage M
WHERE M.year=:years AND C.SSN=M.SSN AND C.age=:fAge
```

We observe that conditions in the WHERE clause often contain host variables from the program under test and some of those host variables may appear directly in branch conditions or are dependent on host variables in branch conditions. For example, the condition `C.age=:fAge` in the WHERE clause of `Qsym` contains host variable `fAge`. The `fAge`'s symbolic expression is calculated as `fage=inputAge+10` and host variable `inputAge` involves in the branch condition pc_2 = `(inputAge > 25)`. Hence enforcing BVC on this branch condition incurs constraints on the generated data. Lines 5-20 in Algorithm 1 give details about how to derive the constraints C_Q by examining the conditions in the WHERE clause and the branch conditions before the query's execution. We discuss details in Section 2.3.

We also observe that the attribute strings (`C1, ..., Ch`) in the SELECT clause may indirectly involve in branch conditions after the query execution. For example, the branch condition pc_4 = `((bal >= 250000 || employed && married) == true)` contains three host variables (`bal`, `employed`, and `married`) that retrieve values from three database attributes (`M.balance, C.jobStatus, C.marriage`) in the returned result set. To enforce CACC and BVC on the branch condition pc_4, we incur new constraints on the generated data. Lines 21-29 in Algorithm 1 give details about how to derive the constraints C_R by examining the attribute strings in the SELECT clause and the branch conditions after the query's execution. Finally, we combine the derived constraints (C_Q and C_R) with the database constraints (C_S) specified at the schema level and call a constraint solver to generate database states.

2.2 Instantiating a Predicate to Satisfy BVC and CACC

A predicate is an expression that evaluates to a boolean value. A predicate may consist of a list of clauses that are joined with logical operators (e.g., NOT, AND, OR). Each clause contains a boolean variable, a non-boolean variable that is compared with a constant or another variable via relational operators, or even a call to a function that returns a boolean value. The predicate `bal>=250000||employed&&married` in branch condition pc_4 contains three clauses: a relational expression `bal>=250000`, a boolean variable `employed`, and another boolean variable `married`.

Test coverage is evaluated in terms of test criteria, as specified by test requirements. Test requirements are specific elements that must be satisfied or covered for software artifacts. Predicate coverage requires that for each p there are instantiations that evaluate p to be `true` and instantiations that evaluate p to be `false`. Clause coverage ensures that for each clause $c \in p$ there are instantiations that

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

[4] In general, there are two types of canonical queries: DPNF with the WHERE clause consisting of a disjunction of conjunctions, and CPNF with the WHERE clause consisting of a conjunction of disjunctions.

evaluate c to be `true` and instantiations that evaluate c to be `false`. Predicate coverage is equivalent to the branch coverage criterion for testing source code while clause coverage is equivalent to the condition coverage. However, neither predicate coverage nor clause coverage subsumes the other. To test both individual clauses and predicates, combinatorial coverage (also called Multiple Condition Coverage) is used to evaluate clauses to each possible combination of truth values. We can see, for a predicate p with n independent clauses, there are 2^n possible combinations; thus, combinatorial coverage is often infeasible in practice.

Active clause criteria such as CACC have been widely adopted to construct a test such that the value of the predicate is directly dependent on the value of the clause that we want to test. CACC is defined in previous work [2]: For each p and each major clause $c_i \in p$, choose minor clauses c_j, $j \neq i$ so that c_i determines p. There are two requirements for each c_i: c_i evaluates to true and c_i evaluates to false. The values chosen for the minor clauses c_j must cause p to be true for one value of the major clause c_i and false for the other, that is, it is required that $p(c_i$ = true) $\neq p(c_i$ = false).

BVC requires multiple test inputs at boundaries [10] because errors tend to occur at extreme or boundary points. For different data types, various boundary values are considered. We use the integer data type for illustration. Suppose that the conditional statement takes the form $[AopB]$ where A is a variable's expression, B is a constant, and *op* is a comparison operator ($==$, $!=$, $>$, $>=$, $<$, $<=$). Depending on the comparison operator, we seek to choose values for A coming from the minimum boundary, immediately above minimum, between minimum and maximum (nominal), immediately below maximum, or the maximum boundary. We list the choices in Table 2. For other data types, we omit details.

Table 2: BVC enforcement for integer

Condition	BVC requirements
A == B	A == B
A != B	A == B + 1, A == maximum A == B − 1, A == minimum
A > B	A == B + 1, A > B + 1, A == maximum
A >= B	A == B , A > B, A == maximum
A < B	A == B − 1, A < B − 1, A == minimum
A <= B	A == B , A < B, A == minimum

Algorithm 2 shows how to generate instantiations satisfying both CACC and BVC for a given predicate. The algorithm accepts a predicate and a boolean evaluation as input and generates a list of instantiations as output. Lines 2-6 in Algorithm 2 enforce CACC by iterating each clause to be the major one and generating assignments for all other minor clauses. Recall that a major clause should determine the predicate for a given instantiation of other clauses. Consider the predicate pc_4 = `bal>=250000||employed && married` with the target evaluation result as `true`. When we choose `bal>=250000` as the major clause, Line 3 generates `(true||employed && married)` $\neq$ `(false||employed && married)` $\wedge$ `(p=true)`. By calling a constraint solver, we can generate instantiations shown in rows 1-3 in Table 3. Similarly we can get instantiations shown in rows 4-5 when choosing `employed` (`married`) as the major clauses. Note that the assignments in rows 4-5 are the same. To enforce CACC for this predicate, we only need one instantiation from rows 1-3 and another instantiation from either row 4 or 5. For example, by choosing rows 1 and 4, we have `{(bal>=250000)=true, employed=true, married=false}`, `{(bal>=250000)=false, employed=true, married=true}`.

Algorithm 2 *EnforceCriteria*: CACC and BVC enforcement

Input: Predicate $p = \{c_1$ op c_2 ... op $c_m\}$, target evaluation E for p
Output: Instantiations I for p

1: Instantiation set $I = \varnothing$;
2: **for** each clause $c_i \in p$ **do**
3: $C_i = (p(c_i = true) \neq p(c_i = false)) \wedge p = E$;
4: Send C_i to a constraint solver and get instantiations I_i;
5: $I = I \cup I_i$;
6: **end for**
7: **for** each instantiation in I **do**
8: **for** each clause c **do**
9: **if** BVC should be satisfied **then**
10: Enforce BVC;
11: Add new instantiations in I;
12: **end if**
13: **end for**
14: **end for**
15: **return** I;

Table 3: Truth table for predicate `(bal>=250000 || employed && married)=true` to satisfy CACC. We choose one instantiation from No.1-3 for `bal>=250000` and an instantiation No.4 or No. 5 for `employed` and `married`

major clause	No.	bal>=250000	employed	married
bal>=250000	1	T	T	F
	2	T	F	T
	3	T	F	F
employed	4	F	T	T
married	5	F	T	T

After further enforcing BVC on the clause `bal>=250000` (Lines 7-14 of Algorithm 2), we have the following six instantiations:

```
{bal=250000,   employed=true, married=false},
{bal>250000,   employed=true, married=false},
{bal=maximum,  employed=true, married=false},
{bal=250000-1, employed=true, married=true},
{bal<250000-1, employed=true, married=true},
{bal=minimum,  employed=true, married=true}.
```

2.3 Deriving Constraints of Database States

Algorithm 1 is invoked when DSE encounters a branch condition. In the illustrative example, the last branch condition pc_4 invokes our algorithm. Using various APIs provided by Pex, we retrieve all branch conditions along the execution of the current path, the query execution point T, the concrete executed query Q and its corresponding symbolic query Q_{sym}, and symbolic expressions of host variables appeared in branch conditions and Q_{sym}.

Conditional expressions in the WHERE clause contain constraints of the generated database state. Formally, we call a SQL parser to decompose Q_{sym} and set C_Q = {(A_{11} AND ... AND A_{1n}) OR ... OR (A_{m1} AND ... AND A_{mn})} from the Q_{sym}. We create an empty variable set V_Q. For each $A_{ij} \in C_Q$, we check whether A_{ij} contains host variables. For each contained host variable, if it is related with any branch condition before the query execution point T, we add it to V_Q and replace it with its symbolic string expressed by variables in the related branch conditions. If not, we replace the variable with its corresponding concrete value contained in the concrete query Q. In our example, we get the constraints C_Q = `{M.year=:years AND C.SSN= M.SSN AND C.age=:fAge}`. For `M.year=:years`, we replace `years` with the value `15`. For `C.age=:fAge`, we replace `fAge` with `inputAge + 10`. We leave `C.SSN= M.SSN` unchanged since it does not contain any host variable. The variable set V_Q = `{fAge}` and C_Q = `{M.year=15 AND C.SSN= M.SSN AND C.age=:inputAge + 10}`.

Next, for each branch condition pc along this path, we check whether pc contains host variables related with the set V_Q. If yes, we enforce it with both CACC and BVC by calling Algorithm 2.

Algorithm 2 returns instantiations I_{pc} that make pc =`true`. The returned instantiations incur further constraints on the generated database state. We refine C_Q as $C_Q = C_Q \times I_{pc}$.

In our example, the branch condition pc_2 (`inputAge> 25`) is related with host variable `fAge`. Algorithm 2 returns new instantiations as {`inputAge=25+1, inputAge>25+1, inputAge=maximum`}. The constraint set C_Q is updated with three constraints:

```
{M.year=15 AND C.SSN= M.SSN AND
C.age=:inputAge+10 AND inputAge=25+1},
{M.year=15 AND C.SSN= M.SSN AND
C.age=:inputAge+10 AND inputAge>25+1},
{M.year=15 AND C.SSN= M.SSN AND
C.age=:inputAge+10 AND inputAge=maximum}.
```

Note that branch conditions after the SQL's execution point T may contain host variables that are dependent on database attributes in the SELECT clause. Enforcing CACC and BVC on those branch conditions further incur constraints on the generated database state. We first identify host variables V_R that are directly dependent on attributes of the returned query result set. For example, we get that the variable `bal` is directly dependent on the attribute `M.balance` in Line 17 in our illustrative example. Then, for each branch condition that contains host variables in or dependent on V_R, we treat pc as a predicate and call Algorithm 2 to enforce both CACC and BVC requirements. In our example, pc_4 = `((bal >= 250000 || employed && married) == true)` and V_R = {`bal, married, employed`}. The returned six instantiations are shown in the last paragraph of Section 2.2. Note that the returned instantiations contain host variables related with variables V_R. We need to replace them with their corresponding database attributes. In our example, host variables `bal, employed, married` are replaced with database attributes `M.balance`, `M.marriage`, and `M.jobStatus`, respectively.

Finally, the set $C = C_Q \times C_R$ contains constraints on the generated database state to enforce CACC and BVC on the source code under test. We also collect basic constraints C_S at the database schema level (e.g., not-NULL, uniqueness, referential integrity constraints, domain constraints, and semantic constraints). For example, attribute `balance` in table `mortgage` must be greater than 0. We then send C together with the schema level constraints C_S to a constraint solver to conduct the data instantiation on the symbolic database. In our prototype system, we use the constraint solver Z3 [5], which is integrated into Pex. Z3 is a high-performance theorem prover being developed at Microsoft Research. Z3 supports linear real and integer arithmetic, fixed-size bit-vectors, extensional arrays, uninterpreted functions, and quantifiers.

3. EVALUATION

Our approach is to provide an assistance to the DSE-based test-generation tools (e.g., Pex [1, 17] for .NET) to improve code coverage with respect to CACC and BVC in database application testing.

We conduct evaluations on two open source database applications: `RiskIt`[6] and `UnixUsage`[7]. `RiskIt` is an insurance quote application that makes estimation based on users' personal information, such as zipcode and income. Programs in `RiskIt` have about 4.3K non-commented lines of code and have a 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. Programs in `UnixUsage` have about 2.8K non-commented lines of code and have a database containing 8 tables, 31 attributes, and more than 0.25 million records. Both applications were originally written in Java. To test them in the Pex environment, we convert their Java source code into C# using a tool called Java2CSharpTranslator[8].

To set up the evaluation, we choose methods that have boundary values and/or logical expressions in branch conditions from the applications. Since there are no tools to measure CACC and BVC directly, we apply a tool [14] that transforms the problem of achieving CACC and BVC to the problem of achieving block coverage by introducing new blocks through code instrumentation. We use `PexGoal.Reached()` to identify whether each introduced block is covered.

For example, the statement `if (inputAge > 25) {...}` in Line 08 in our illustrative example becomes

```
08a:  if (inputAge == 25+1)
          {PexGoal.Reached()...}
08b:  else if (inputAge > 25+1)
          {PexGoal.Reached()...}
 08:  if (inputAge > 25){...}
```

after the code instrumentation.

Tables 4 show the results of our evaluation. We use n to denote the number of `PexGoal.Reached()` statements introduced in each method. The coverage of all n introduced blocks indicates the full achievement of CACC and BVC. Given a database state, the current Pex cannot generate sufficient program inputs to achieve higher code coverage especially when program inputs are directly or indirectly involved in embedded SQL statements. In our experiment, we also apply our previous approach [13] to assist Pex generate sufficient input values for program parameters.

We first run Pex (in addition to our program input generation tool [13]) without applying Algorithm 1 to generate new records. We use n_1 to denote the number of covered `PexGoal.Reached()` statements in this experiment. We then apply Algorithm 1 to generate new database records and run Pex in addition to our program input generation tool [13]. We use n_2 to denote the number of covered `PexGoal.Reached()` statements during this step. The value of (n_2 - n_1)/n captures the increase gained by Pex assisted by our new approach in achieving CACC and BVC. We see from Tables 4 that the n_2 values are equal to n for all methods, indicating that our new approach assists Pex to reach the full CACC and BVC coverage (a 21.21% increase on average for `RiskIt` and a 46.43% increase for `UnixUsage`). The detailed evaluation subjects and results can be found on our project website[9].

4. RELATED WORK

Various coverage criteria [2, 10] have been proposed to generate test inputs for traditional (non-database) applications. BVC and LC are criteria that complement the widely used branch coverage. Most recently, Pandita et al. [14] developed a general approach that instruments transformed branch conditions to source code and guides DSE to reach high BVC and LC for traditional applications. However, those criteria have not been supported in testing database applications.

Database application testing has attracted much attention recently [5, 7, 12, 16, 19]. Most approaches aim to achieve high *branch coverage* of source code under test by either generating database states from scratch [5,12,16] or using existing database states to generate sufficient program inputs [11, 13]. Query-aware test database

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

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

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

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

[9]http://www.sis.uncc.edu/∼xwu/DBGen

Table 4: Evaluation results

	No.	method	class	parameter type	parameter name	total n	covered(blocks) n_1	covered(blocks) n_2	increase $(n_2\text{-}n_1)/n$
RiskIt	1	filterZipcode	UserManager	String	zip	23	20	23	13.04%
	2	filterEducation	UserManager	String	edu	4	2	4	50.00%
	3	filterOccupation	UserManager	String	occupation	4	2	4	50.00%
	4	filterMaritalStatus	UserManager	String	status	4	2	4	50.00%
	5	filterEstimatedIncome	UserManager	String	getIncome	7	5	7	28.57%
	6	browseUserProperties	UserManager	ArrayList<>	prop	100	81	100	19.00%
	7	getValues	EstimateIncome	int	ssn	52	44	52	15.38%
	8	calculateUnemploymentRate	UserManager	String	stateName	4	0	4	100.00%
	all methods (total)					198	156	198	21.21%
Unix Usage	1	computeFileToNetworkRatio ForCourseAndSessions	CourseInfoManager	int, int, int	courseId, startSession, endSession	4	0	4	100.00%
	2	computeBeforeAfterRatioByDept	DeptInfoManager	int, String	deptid, date	8	0	8	100.00%
	3	computeFileToNetworkRatioForDept	DeptInfoManager	int	deptId	11	0	11	100.00%
	4	courseIdExists	TranscriptManager	int	courseId	4	0	4	100.00%
	5	doesUserIdExist	UserInfoManager	String	userId	11	10	11	9.09%
	6	isDepartmentIdValid	UserInfoManager	int	departmentId	10	9	10	10.00%
	7	isOfficeIdValid	UserInfoManager	int	officeId	10	9	10	10.00%
	8	isRaceIdValid	UserInfoManager	int	raceId	10	9	10	10.00%
	9	getGPAForAllUsers	UserInfoManager	N/A	N/A	12	8	12	25.00%
	10	officeIdExists	OfficeInfoManager	int	officeId	4	0	4	100.00%
	all methods (total)					84	45	84	46.43 %

generation approaches [3], which consider both queries and schema level constraints, were also investigated with the aim of DBMS performance testing. Generating database states for achieving advanced coverage criteria such as CACC and BVC has not been explored.

Focusing on an isolated SQL statement, Tuya et al. [18] proposed a coverage criterion, SQLFpc (short for SQL Full Predicate Coverage), based on the Modified Condition/Decision Coverage. Their approach mutates a given SQL query statement into a set of queries that satisfy MC/DC with the aim of detecting faults in the SQL query statement. Riva et al. [6] developed a tool to generate data to enforce SQLFpc for a SQL statement. Our work leaves the embedded SQL statement unchanged. Instead, we generate database states with the aim of detecting faults in source code.

5. CONCLUSION AND FUTURE WORK

In this paper, we developed a general approach to generating test database states that can achieve program BVC and CACC. We implemented our approach in Pex, a DSE tool for .NET. Our evaluation demonstrated the feasibility of our approach. In our future work, we plan to investigate how to optimize the constraint collection and data instantiation. We plan to study complex SQL queries (e.g., GROUPBY queries with aggregations) and extend our technique to deal with multiple queries in database applications.

Acknowledgment

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

6. REFERENCES

[1] Microsoft Research Foundation of Software Engineering Group, Pex:Dynamic Analysis and Test Generation for .NET. 2007.

[2] P. Ammann, A. J. Offutt, and H. Huang. Coverage criteria for logical expressions. In *ISSRE*, pages 99–107, 2003.

[3] C.Binnig, D.Kossmann, and E.Lo. Reverse query processing. In *ICDE*, 2007.

[4] J. Chilenski and S.P.Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, 1994.

[5] D.Chays and J.Shahid. Query-based test generation for database applications. In *DBTest*, 2008.

[6] C. de la Riva, M. J. S. Cabal, and J. Tuya. Constraint-based test database generation for sql queries. In *AST*, 2010.

[7] Y. Deng and D. Chays. Testing database transactions with agenda. In *ICSE*, 2005.

[8] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

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

[10] N. Kosmatov, B. Legeard, F. Peureux, and M. Utting. Boundary coverage criteria for test generation from formal models. In *ISSRE*, pages 139–150, 2004.

[11] C. Li and C. Csallner. Dynamic symbolic database application testing. In *DBTest 2010*.

[12] M.Emmi, R.Majumdar, and K.Sen. Dynamic test input generation for database applications. In *ISSTA*, 2007.

[13] K. Pan, X. Wu, and T. Xie. Generating program inputs for database application testing. Technical Report, UNC Charlotte, 2011.

[14] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux. Guided test generation for coverage criteria. In *ICSM*, 2010.

[15] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, 2005.

[16] K. Taneja, Y. Zhang, and T. Xie. MODA: Automated Test Generation for Database Applications via Mock Objects. In *ASE 2010*.

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

[18] J. Tuya, M. J. S. Cabal, and C. de la Riva. Full predicate coverage for testing sql database queries. volume 20, pages 237–288, 2010.

[19] X. Wu, C. Sanghvi, Y. Wang, and Y. Zheng. Privacy Aware Data Generation for Testing Database Applications. In *IDEAS*, pages 317–326, 2005.