Guided Test Generation for Database Applications via Synthesized Database Interactions

KAI PAN and XINTAO WU, University of North Carolina at Charlotte
TAO XIE, University of Illinois at Urbana-Champaign

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 article, 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.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Automatic test generation, dynamic symbolic execution, synthesized database interactions, database application testing

ACM Reference Format:
Kai Pan, Xintao Wu, and Tao Xie. 2014. Guided test generation for database applications via synthesized database interactions. ACM Trans. Softw. Eng. Methodol. 23, 2, Article 12 (March 2014), 27 pages.
DOI: http://dx.doi.org/10.1145/2491529

1. INTRODUCTION

For quality assurance of software applications, testing is essential before the applications are deployed [Tassey 2002; Cusumano and Selby 1997]. Testing software applications can be classified into categories such as functional testing, performance testing, security testing, environment and compatibility testing, and usability testing. Among different types of testing, functional testing focuses on functional correctness. An important task of functional testing is to generate test inputs to achieve full or at least high code coverage. There, covering a branch is necessary to expose a potential fault within that branch. To cover specific branches, it is crucial to generate appropriate

K. Pan and X. Wu were supported in part by the U.S. National Science Foundation under CCF-0915059, and T. Xie under CCF-0915400.
Authors' addresses: K. Pan and X. Wu, Department of Software and Information Systems, University of North Carolina at Charlotte, Charlotte, NC 28223; email: {kpan, xwu}@uncc.edu; T. Xie, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 60801; email: taoxie@illinois.edu.

DOI: http://dx.doi.org/10.1145/2491529

tests, including appropriate program inputs (i.e., input arguments). 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 [Godefroid et al. 2005; Sen et al. 2005]. DSE extends the traditional symbolic execution [King 1976; Clarke 1976] by running a program with concrete inputs while collecting both concrete and symbolic information at runtime, making the analysis more precise [Godefroid et al. 2005]. 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.

Testing database applications requires generating test inputs of both appropriate program inputs and sufficient database states. However, producing these test inputs faces great challenges, because database states play crucial roles in database application testing, and constraints from the issued SQL queries and queries' returned result set impact which paths or branches to execute within program code. Recently, some approaches [Emmi et al. 2007; Taneja et al. 2010] adapt DSE to generate tests, including both program inputs and database states, for achieving high structural coverage of database applications. Emmi et al.[2007] proposed an approach that runs the program simultaneously on concrete program inputs as well as on symbolic inputs and a symbolic database. The symbolic database is a mapping from symbolic expressions to logical formulas over symbolic values. The symbolic path constraint is treated as a logical formula over symbolic values. Solving these logical formulas can help generate database records that satisfy the execution of a concrete query. 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. Then, to explore a new path, the approach flips a branch condition and generates new program inputs and corresponding database records. To solve the problem when the associated database is not available, the MODA framework [Taneja et al. 2010] transforms the program under test to interact with a mock database in place of the real database. The approach applies a DSE-based test generation tool called Pex [Tillmann and de Halleux 2008] for .NET to collect constraints of both program inputs and the associated database state. 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. Both approaches collect constraints from program code and treat the associated database (either real or mock) as an external component.

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 subpaths 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 subpaths 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.

To generate both effective program inputs and sufficient database states, we need to correlate program-execution constraints and environment constraints seamlessly when applying DSE on testing database applications. 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 [Emmi et al. 2007; Taneja et al. 2010] 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 [Emmi et al. 2007] or mock database [Taneja et al. 2010]) 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 (i.e., constraints from the conditions in the WHERE clause of the prematurely concretized query) may conflict with later constraints. First, constraints from the concrete query may conflict with database schema constraints. 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. Second, constraints from the concrete query may conflict with query-result-manipulation constraints. The violation of query-result-manipulation constraints could cause low code coverage of query-result manipulation code. While it is essential to collect sufficient constraints required by generating both program inputs and database states, naturally correlating the aforementioned four parts of constraints remains a significant problem. The root cause stems from the fact that although the problem could be solved by a thorough symbolic representation of the database state as well as the symbolic input variables with a sufficiently powerful constraint solver, it would be still challenging or even infeasible to bridge the gap caused by an external associated database.

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 article, we propose a DSE-based test generation approach called *SynDB* to address the preceding problems of two types of constraint conflicts. SynDB treats the associated database as an internal component rather than the black-box query-execution engine. Our approach is the first work that uses a fully symbolic database. 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

synthesized 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. This way, we correlate the 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 operations on the symbolic database state to collect constraints for both program inputs and the associate database state. By applying a constraint solver on the collected constraints, we thus attain effective program inputs and sufficient database states to achieve high code coverage. Note that our approach does not require the physical database to be in place. In practice, if needed, we can map the generated database records back to the real database for further use.

This article makes the following main contributions.

—We present 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.
—We introduce the first approach to provide a thorough symbolic representation of the database state and 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.
—We provide a prototype implemented for the proposed approach using a state-of-the-art tool called Pex [Microsoft 2007] for .NET from Microsoft Research as the DSE engine 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 the aforementioned two types of constraint conflicts of existing test generation approaches. We then 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 I. The method `calcStat` first sets up database connection (Lines 03–05). It then constructs 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.

To test the preceding code, DSE [Emmi et al. 2007] chooses random or default values for `inputYear` (e.g., `inputYear = 0` [1]). Here, the query-construction constraints are simply *true*. Constraints from the concrete query are C.SSN = M.SSN AND C.zipcode = 28223 AND M.year = 0. For generation of a database state, the constraint for the attribute `M.year` in the concrete query becomes `M.year = 0`. However, we observe from the schema in Table I that the randomly chosen value (e.g., `inputYear = 0`) violates

[1]The same problem occurs with inputYear == 1.

```
01:public int calcStat(int inputYear) {
02:  int zip = 28223, count = 0;
03:  SqlConnection sc = new SqlConnection();
04:  sc.ConnectionString = "..";
05:  sc.Open();
06:  string query = buildQuery(zip, inputYear);
07:  SqlCommand cmd = new SqlCommand(query, sc);
08:  SqlDataReader results = cmd.ExecuteReader();
09:  while (results.Read()){
10:     int income = results.GetInt(1);
11:     int balance = results.GetInt(2);
12:     int year = results.GetInt(3);
13:     int diff = (income - 1.5 * balance) * year;
14:     if (diff > 100000){
15:        count++;}}
16:  return count;}

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

Fig. 1. A code snippet from a database application in C#.

Table I. Database Schema

customer table			mortgage table		
Attribute	Type	Constraint	Attribute	Type	Constraint
SSN	Int	Primary Key	SSN	Int	Primary Key
name	String	Not null			Foreign Key
gender	String	$\in$ {F, M}	year	Int	$\in$ {10, 15, 30}
zipcode	Int	[00001, 99999]			
age	Int	(0, 100]	balance	Int	[2000, Max)
income	Int	[100000, Max)			

a database schema constraint: `M.year` can be chosen from only the set {`10, 15, 30`}. Thus, we have the first type of conflict: query constraints (i.e., constraints derived from the WHERE clause of the concrete query), thus conflict with the database schema constraints. As previously mentioned, the violation of database schema constraints would cause the generation of invalid database states. Thus, existing DSE-based test generation approaches may fail to generate sufficient database records to cause the execution to enter the query result manipulation (e.g., the `while` loop in Lines 09–15). Furthermore, even if the specific database schema constraint (i.e., `M.year` $\in$ {`10, 15, 30`}) does not exist and test execution is able to reach later part, the branch condition in Line 14 cannot be satisfied. The values for the attribute `M.year` (i.e., `M.year` = `0` or `M.year` = `1`) from the query in Line 06b are prematurely concretized. Then, such premature concretization causes conflict with later constraints (i.e., in Line 13, we have diff = (income − 1.5 * balance) * 0 or 1, which conflicts with the condition in Line 14) from subpaths for manipulating the query result. Thus, we have the second type of constraint conflict. From these two types of constraint conflicts, we observe that treating the database as an external component isolates the query constraints with database schema constraints and query-result-manipulation constraints.

```
01:public int calcStat(int inputYear, DatabaseState dbState) {
02:  int zip = 28223, count = 0;
03:  SynSqlConnection sc = new SynSqlConnection(dbState);
04:  sc.ConnectionString = "..";
05:  sc.Open();
06:  string query = buildQuery(zip, inputYear);
07:  SynSqlCommand cmd = new SynSqlCommand(query, sc);
08:  SynSqlDataReader results = cmd.ExecuteReader();
09:  while(results.Read()){
10:     int income = results.GetInt(1);
11:     int balance = results.GetInt(2);
12:     int year = results.GetInt(3);
13:     int diff = (income - 1.5 * balance) * year;
14:     if (diff > 100000){
15:        count++;}}
16:  return count;}

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

Fig. 2. Transformed code produced by SynDB for the code in Figure 1.

```
public class customerTable {
    public class customer {/*define attributes*/}
    public List<customer> customerRecords;
    public void checkConstraints() {/*check constraints for each attribute*/}}
public class mortgageTable {
    public class mortgage {/*define attributes*/}
    public List<mortgage> mortgageRecords;
    public void checkConstraints() {/*check constraints for each attribute;
                                      e.g., 'year' must be in {10,15,30}*/}}
public class DatabaseState {
    public customerTable customerT = new customerTable( );
    public mortgageTable mortgageT = new mortgageTable( );
    public void checkConstraints(){/*check constraints for each table*/}}
```

Fig. 3. Synthesized database state.

To address the preceding two types of constraint conflicts in testing database applications, our SynDB approach replaces the original database interactions by constructing synthesized database interactions. For example, we transform the example code in Figure 1 into another form shown in Figure 2. Note that in the transformed code, methods in the bold font indicate our new synthesized database interactions. We also add a new input `dbState` to the program with a synthesized data type `DatabaseState`. The type `DatabaseState` represents a synthesized database state whose structure is consistent with the original database schema. For example, for the schema in Table I, its synthesized database state is shown in Figure 3. The program input `dbState` is then passed through synthesized database interactions `SynSqlConnection`, `SynSqlCommand`, and `SynSqlDataReader`. Meanwhile, at the beginning of the synthesized database connections, we ensure that the associated database state is valid by calling a method predefined in `dbState` to check the database schema constraints for each table.

Table II. Generated Program Inputs and Database States to Cover Paths `Line09 = true`, `Line14 = false` and `Line09 = true`, `Line14 = true`

	dbState								
	dbState.Customer						dbState.Mortgage		
inputYear	SSN	name	gender	zipcode	age	income	SSN	year	balance
15	001	AAA	F	28,223	45	150,000	001	15	100,000
15	002	BBB	M	28,223	55	150,000	002	15	50,000

To synthesize the database operations for the synthesized database interactions, we incorporate the query constraints as program-execution constraints in normal program code. To do so, within the synthesized method `ExecuteReader`, 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). 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` $\in \{$`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 II (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). The transformed code can also guide DSE's exploration to collect later constraints (the query-result-manipulation constraint in Line 14) from subpaths 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 II).

Note that although the MODA approach [Taneja et al. 2010] conducts code-transformation on the program under test to interact with a mock database in place of the real database, the preceding two types of constraint conflicts still exist. To force DSE to treat the associated database records symbolically, the approach calls a Pex API method to initialize and assign values to corresponding table columns. However, the relations (i.e., data dependencies) among program inputs, symbolic variables assigned by the Pex API method, and program variables involved in the query are lost. Database records are generated based on constraints from the prematurely concretized SQL queries. Thus, the aforementioned two types of constraint conflicts are not avoided by this design decision. On the other hand, our SynDB approach correlates all four kinds of constraints within database applications into a seamless framework and thus is able to avoid constraint conflicts during test generation.

3. APPROACH

Our approach relates the schema constraints, 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

Table III. A Summary of Synthesized Database Interactions

Original class	SqlConnection	SqlCommand	SqlDataReader
New class	SynSqlConnection	SynSqlCommand	SynSqlDataReader
New field	DatabaseState dbStateConn	DatabaseState dbStateComm	DataTable resultSet
Main modified methods and functionalities	SynSqlConnection (DatabaseState dbStatePara) ->pass the symbolic database state	SynSqlCommand(string q, SynSqlConnection SSConn) ->pass the symbolic database state	bool Read() -> iterate through the field `DataTable resultSet`
	DatabaseState getDB() ->return the field `dbStateConn`	SynSqlDataReader ExecuteReader() ->simulate the query execution on the symbolic database state	int GetInt32(), double GetDouble().... ->read column values from `DataTable resultSet`

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 an SQL server database [Microsoft 2012b]. 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. 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 an `SqlConnection` object). Second, it constructs a query to be executed and combines the query into the connection (e.g., construct an `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 an `SqlDataReader` object by calling the API method `ExecuteReader()`[2]). Fourth, the returned query result is manipulated for further execution. Table III 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 schema in Table I, 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. We construct

[2]If the query is to modify the database state (such as INSERT, UPDATE, and DELETE), methods `ExecuteNonQuery()` or `ExecuteScalar()` are applied.

a checkConstraints() method to implement the checking for database integrity constraints: for Check Constraints, we check whether the value in a certain column satisfies an arbitrary expression defined in the schema; for Not-Null Constraints, we specify that a column must not assume the null value; for Unique Constraints, we ensure that the data contained in a column is unique with respect to all the other rows in the table; for Primary Keys, we indicate that the column is unique and not-null for rows in the table; for Foreign Keys, we check the consistency of a certain column that is referred to another column in another table. During the implementation, we call a Pex API method PexAssume() when enforcing the database schema checking. PexAssume() is called to filter out undesirable test inputs and force the test inputs to satisfy all the constraints indicated within the provided conditions. Such design decision stems from the fact that the database schema constraints are unlike those conditions in if-else statements for which we must explore both true and false branches. In fact, the negation of schema constraints is actually what we try to avoid. Thus, invoking PexAssume() could mandatorily enforce database schema constraints with better efficiency. For example, to enforce the primary key constraint, we include the expression of the primary key's behavior into a PexAssume() statement.

Note that we are also able to capture complex constraints at the schema level, such as constraints across multiple tables and multiple attributes. To implement triggers and stored procedures, we manually map their original definitions into normal program methods that take parameters as corresponding database columns. Within the method body, we manually apply the execution on associated database tables and return result set as required. To implement cascading updates/deletes, we conduct the implementation on the indicated column by monitoring the types of actions on that column. Once we observe that there comes an update or a delete, we automatically invoke corresponding updating/deleting actions on the referred tables/columns.

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 III, for the interactions SynSqlConnection and SynSqlCommand, 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 SynSqlConnection, 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 SynSqlCommand whose methods ExecuteReader()

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

Fig. 4. Synthesized `SqlConnection`.

```
public class SynSqlCommand{
   public string query;
   public SynSqlConnection synSC;
   public DatabaseState dbStateComm; //new field
   public SynSqlCommand(string q, SynSqlConnection SSConn){
          query = q;
          synSC = SSConn;
          dbStateComm = SSConn.getDB(); //pass the synthesized database}
   public SynSqlDataReader ExecuteReader(){ //execute the select operation;
          SynSqlDataReader synReader = new SynSqlDataReader();
          DatabaseState synDB = this.getDB();
          synReader = SelectExe(synDB,this.getQuery()); //details in Algorithm 1
          return synReader;}
   public SynSqlDataReader ExecuteNonQuery(){ //execute the modify operation;
          DatabaseState synDB = this.getDB();
          ModifyExe(synDB,this.getQuery());} //details in Algorithm 2
   public DatabaseState getDB() {return dbStateComm;} //new method
   public string getQuery() {return query;} //new method}
```

Fig. 5. Synthesized `SqlCommand`.

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).

We construct a synthesized data type to represent the query result, whose structures are built dynamically based on the query to be executed. For example, we construct `SynSqlDataReader` to represent the query result and use a field with the type `DataTable` to represent the returned records. We choose the type `DataTable` [Microsoft 2012a] because its characteristics are very similar to a query's real returned result set. The entire data structure is expressed in a table format. For a `DataTable` object, its columns can be built dynamically by indicating the column names and their data types.

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*

```
SELECT  select-list
FROM    from-list
WHERE   qualification
```

Fig. 6. A simple query.

ALGORITHM 1: *SelectExe*: Evaluate a SELECT statement on a symbolic database state

Input: DatabaseState *dbStateComm*, a SELECT query Q
Output: SynSqlDataReader R

1: Construct a `SynSqlDataReader` object R;
2: **for** each table T_i in Q's from-list **do**
3: **for** each attribute A in T_i **do**
4: Find A's corresponding field F in schema;
5: Construct a new DataColumn C;
6: C.ColumnName = F.name;
7: C.DataType = F.type;
8: R.resultSet.Columns.Add(C);
9: **end for**
10: **end for**
11: **for** each table T_i in Q's from-list **do**
12: Find T_i's corresponding table T_i' in *dbStateComm*;
13: **end for**
14: $R.resultSet.Rows = T_1' \times T_2' ... \times T_n'$;
15: Construct a string $S = Q$'s WHERE clause;
16: Replace the database attributes in S with their corresponding column names in R.resultSet;
17: Replace the SQL logical connectives in S with corresponding program logical operators;
18: **for** each row r in $R.resultSet.Rows$ **do**
19: **if** r does not satisfy S **then**
20: $R.resultSet.Rows.Remove(r)$;
21: **end if**
22: **end for**
23: **for** each column c in $R.resultSet.Columns$ **do**
24: **if** c does not appear in Q's SELECT clause **then**
25: $R.resultSet.Columns.Remove(c)$;
26: **end if**
27: **end for**
28: **return** R;

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. We construct the full cross-product of relation tables followed by selection and then projection, which is based on the conceptual evaluation of the SQL queries. 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 `DataTable resultSet` by adding new columns corresponding to the attributes of the tables appearing in the FROM clause. The new columns have the same names and data types as their corresponding attributes. 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 previously mentioned, 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. For SQL operators that correspond directly to programming language operators, we map them to corresponding operators (e.g., $>$, $=$, and $<$). For those SQL operators that do not correspond directly to programming language operators (e.g., LIKE, IS NULL), we map them to corresponding programming language API methods provided by the basic data types (e.g., map LIKE to string.contains()). Currently we do not handle 3-valued logic for SQL queries. 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

ALGORITHM 2: *ModifyExe*: 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
```

the WHERE clause. We then check whether this deletion would violate the schema constraints; otherwise, we remove these rows.

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 subqueries with aggregation functions, union, distinct, and group-by views, etc. The syntax of SQL queries is defined in the ISO standardization.[3] The fundamental structure of an 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 [Kim 1982; Dayal 1987; Ahmed et al. 2006] on query transformation in databases has been explored to unnest complex queries into equivalent single-level canonical

[3]American National Standard Database Language SQL. ISO/IEC 9075:2008. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=45498.

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

Fig. 7. A canonical query in DPNF.

queries. Researchers showed that almost all types of subqueries 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 `(A11 OR... OR A1n) AND ... AND (Am1 OR... OR Amn)`). Note that DPNF and CPNF can be transformed mutually using DeMorgan's rules [Goodstein 2007]. 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 that 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 involved 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 can 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

```
 ...
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;}
```

Fig. 8. An example where cardinality constraints come from the query result manipulation.

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 a small number of records can satisfy the condition in Line 09b. Hence, this problem is reduced to a traditional issue [Xie et al. 2009]: to explore a program that contains one or more branches with relational conditions (here, we have `(count > 10)`) where the operands are scalar values (i.e., integers or floating-point numbers) computed based on control-flow decisions connected to program inputs through data flow (here, we have `if (balance > 50000) count++;`).

In the literature, Xie et al. proposed an approach *Fitnex* [2009] 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 [Xie et al. 2009] 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 [Emmi et al. 2007; Taneja et al. 2010] in testing database applications?

RQ2. What is the running cost of our approach compared with an existing approach [Taneja et al. 2010] on generating tests?

4.1. Subject Applications

We conduct an empirical evaluation on three open-source database applications: iTRUST[4], RiskIt[5], and UnixUsage[6]. These applications contain comprehensive programs and have been previously widely used as evaluated applications (iTRUST [Clark et al. 2011], RiskIt and UnixUsage [Grechanik et al. 2010; Taneja et al. 2011; Pan et al. 2011b]). iTRUST is a class project created at North Carolina State University for teaching software engineering. It consists of functionalities that cater to patients and the medical staff. The accompanied database contains 30 tables and more than 130 attributes. 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. 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. The three applications were originally 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.[7] Java2CSharpTranslator is an Eclipse plug-in based on the fact that Java and C# have a lot of syntax/concept in common. The detailed evaluation subjects and results can be found on our project website.[8]

We report the code coverage using block coverage provided by Pex. Code coverage could be calculated for code blocks, lines of code, and partial lines if they are executed by a test run. Among these code entities under coverage measurement, a code block is a code path with a single entry point, a single exit point, and a set of instructions that are all run in sequence. We choose block coverage (e.g., the percentage of code blocks being covered) to be the metric as we find for these subject applications, the number of covered blocks could reflect the covered code portions because most of the methods under test do not contain complex logics that involve many blocks. For the analysis time of our approach, we measure the execution time by running Pex on the transformed code. In our evaluation, we set Pex's TimeOut value as 120 seconds. Pex provides a set of commands to specify the upper bound of the execution cost of which the TimeOut command is used to indicate the time to stop if no more code could be covered.

For these applications, we focus on the methods whose SQL queries are constructed dynamically. For the iTRUST application, from the subpackage called iTRUST.DAO, we choose 14 methods that contain queries whose variables are data-dependent on program inputs. The iTRUST.DAO package mainly deals with database interactions and data accessing. 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]http://agile.csc.ncsu.edu/iTrust.
[5]https://riskitinsurance.svn.sourceforge.net.
[6]http://sourceforge.net/projects/se549unixusage.
[7]http://sourceforge.net/projects/j2cstranslator/.
[8]http://www.sis.uncc.edu/~xwu/DBGen.

Table IV. Schema Constraints on iTRUST

Application	Table:Attribute	Constraints
iTRUST	Users:Role	enum('patient','admin','hcp','uap','er','tester','pha', 'lt') NOT NULL
	Personnel:role	enum('patient','admin','hcp','uap','er','tester','pha', 'lt') NOT NULL
	Personnel:state	enum('AK','AL', . . . ,'WV', 'WY') NOT NULL
	Patients:state	enum('AK','AL', . . . ,'WV', 'WY') NOT NULL
	icdcodes:Chronic	enum('no','yes') NOT NULL
	icdcodes:Code	decimal(5,2) NOT NULL

Table V. Added Extra Schema Constraints on RiskIt and UnixUsage("PK" stands for "Primary Key")

Application	Table:Attribute	Original constraints	Added constraints
RiskIt	education:EDUCATION	char(50)	$\in$ {high school, college, graduate}
	job:SSN	int, NOT NULL, PK	[000000001, 999999999]
	userrecord:SSN	int, NOT NULL, PK	[000000001, 999999999]
	userrecord:ZIP	char(5)	ZIP.length = 5
	userrecord:MARITAL	char(50)	$\in$ {single, married, divorced, widow}
UnixUsage	COURSE_INFO:COURSE_ID	int, NOT NULL, PK	[100,999]
	DEPT_INFO:RACE	varchar(50)	$\in$ {white, black, asian, hispanic}
	TRANSCRIPT:USER_ID	varchar(50) NOT NULL	[000000001, 999999999]

4.2. Evaluation Setup

The three applications have predefined their own schemas for the associated databases in attached .sql files. For the iTRUST application, the predefined database schema constraints are more comprehensive than the other two. We list the contained constraints related with the methods under test in Table IV. However, for RiskIt and UnixUsage, we observe that the predefined database schema constraints are over-simplified and contain only the basic primary key constraints and data type constraints. To better reflect real-world database schema constraints in real practice, 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 for RiskIt and UnixUsage are listed in Table V.

We next implement code transformation on the original program code under test. As previously mentioned, 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 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. We construct new tests using the program inputs generated by Pex's execution on the transformed code. Note that these program

```
01:public int SynfilterZipcode(String zip, dbStateRiskIt dbstateRiskIt){
02:  int count = 0;
03:  SynRiskItSqlDataReader result = null;
04:  String cmd_zipSearch = "SELECT * from userrecord where zip = '" + zip + "'";
05:  SynRiskItSqlConnection conn = new SynRiskItSqlConnection(dbstateRiskIt);
06:  conn.ConnectionString = "Data Source=(local);Initial Catalog=riskit;Integrated Security=SSPI";
06:  conn.Open();
07:  SynRiskItSqlCommand cmd = new SynRiskItSqlCommand(cmd_zipSearch, conn);
08:  result = cmd.ExecuteReader();
09:  Console.WriteLine(``List of customers for zipcode : '' + zip);
10:  Console.WriteLine(``%20s |%20s |'', "NAME", "SSN");
11:  while (result.Read()){
12:     ++count;
13:     Console.WriteLine(``%s |%s |'', result.GetValue(1).ToString(), result.GetValue(0).ToString());}
14:  if (count == 0)
15:     Console.WriteLine(``There are no customers enrolled in this zipcode'');
16:  else
17:     Console.WriteLine(``No. of customers in zipcode : '' + zip + `` is '' + count);
18:  result.Close();
19:  return count;}
```

Fig. 9. SynfilterZipcode: Transformed code of method filterZipcode.

inputs have also been saved in the methodname.g.cs files. Then, 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. The method accepts an input zip to form an query, searches corresponding records from the userrecord table, and conducts some calculation from the returned records. After the code transformation, we get a new method SynfilterZipcode, as shown in Figure 9. We next run Pex on the transformed code SynfilterZipcode to conduct test generation. The generated tests are then automatically saved by Pex 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 11 = true, and Line 16 = 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 measure the code coverage for the original code under test.

To compare our approach with an existing test-generation approach for database application testing [Emmi et al. 2007], we make use of our SynDB framework. Basically, based on our transformed code, we simulate the existing approach by generating database records using constraints from the concrete queries obtained at each query-issuing point by DSE's exploration. We insert these generated records back to the real database so that DSE can enter the query-result iteration. Note that at this point, if there exists a conflict with the database schema, the generated records to be inserted will be rejected by the database, corresponding to the first type of constraint conflict as previously mentioned. Then, when DSE is able to enter the query-result iteration, to explore the query-result-manipulation code, we get additional constraints from the query-result-manipulation code. Using these constraints together with the constraints obtained previously, we generate new records and insert them back to the

```
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.SSN = 100000000;
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 s1 = new dbStateRiskIt();
19:  s1.userrecordTable = userrecordTable;
20:  i = this.SynfilterZipcode("10001", s1);
21:  Assert.AreEqual<int>(1, i); //Code instrumentation for records insertion (pseudocode)
22:  SqlConnection conn = new SqlConnection();
23:  conn.ConnectionString = "RiskIt";
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.execute(query)}
```

Fig. 10. Tests generated by Pex on `SynfilterZipcode`.

```
01:[TestMethod]
02:[PexGeneratedBy(typeof(FilterZipcode))]
03:public void filterZipcode101(){
04:  int i;
05:  i = this.filterZipcode("10001");
06:  Assert.AreEqual<int>(1, i);}
```

Fig. 11. Constructed tests for `filterZipcode`.

real database again. Note that at this point, if there exists a conflict between the query-result-manipulation constraints and query constraints, the generation will fail, corresponding to the first type of constraint conflict as previously mentioned. To measure the code coverage achieved by the existing approach [Emmi et al. 2007], we create tests using corresponding program inputs generated by Pex and run these tests with the real database. Such simulated results are expected to be equivalent to the results produced with the original implementation of the existing approach.

To compare our approach with the MODA framework [Taneja et al. 2010], we modify the MODA package's settings to match our subject applications, such as constructing mock databases (e.g., creating tables and columns) based on the given database schema. We run the package on each method under test and record the code coverage and running time.

4.3. Results

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

4.3.1. Code Coverage. In Tables VI, VII, and VIII, 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

Table VI. Evaluation Results on iTrust

No.	method	total (blocks)	covered(blocks)			time(seconds)	
			others	SynDB	increase	SynDB	MODA
1	addUser	21	16	19	14.28%	17.1	33.3
2	updateCode	19	15	17	10.53%	14.3	38.2
3	addICDCode	23	17	21	17.39%	21.0	40.1
4	getICDCode	21	16	19	14.28%	18.9	35.3
5	addEmptyPersonnel	23	17	21	17.39%	17.3	39.6
6	editPersonnel	20	15	18	15.00%	16.2	28.9
7	getRole	27	19	25	22.22%	15.4	31.1
8	editPatient	25	17	23	24.00%	16.8	27.7
9	getDiagnosisCounts	49	41	47	12.24%	28.9	time out
10	getWeeklyCounts	22	17	20	13.64%	15.7	43.8
11	findEarliestIncident	19	15	17	10.53%	11.4	33.5
12	add(DiagnosesDAO)	17	13	15	11.76%	10.3	28.5
13	edit(DiagnosesDAO)	19	15	17	10.53%	11.8	30.1
14	getAllOfficeVisitsForDiagnosis	38	29	36	18.42%	22.0	time out
all methods (total)		343	262	315	15.45%	237.1	650.1

Table VII. Evaluation Results on RiskIt

No.	method	total (blocks)	covered(blocks)			time(seconds)	
			others	SynDB	increase	SynDB	MODA
1	getAllZipcode	39	17	37	51.28%	27.9	42.1
2	filterOccupation	41	37	37	0%	17.6	36.2
3	filterZipcode	42	28	38	23.81%	14.2	54.1
4	filterEducation	41	27	37	24.39%	13.1	35.9
5	filterMaritalStatus	41	27	37	24.39%	8.9	33.7
6	findTopIndustryCode	19	14	14	0%	11.8	28.5
7	findTopOccupationCode	19	14	14	0%	12.1	27.7
8	updatestability	79	67	75	10.13%	68.8	time out
9	userinformation	61	51	57	9.84%	74.3	time out
10	updatetable	60	50	56	10.00%	96.2	time out
11	updatewagetable	52	48	48	0%	101.6	time out
12	filterEstimatedIncome	58	44	54	17.24%	19.4	32.2
13	calculateUnemploymentRate	49	45	45	0%	42.7	74.4
14	calculateScore	93	16	87	76.35%	66.5	time out
15	getValues	107	68	99	28.97%	82.2	time out
16	getOneZipcode	34	23	32	26.47%	21.3	38.6
17	browseUserProperties	108	96	104	7.41%	time out	time out
all methods (total)		943	672	871	21.10%	798.6	1243.4

generated by our approach, the number of covered blocks using tests generated by existing approaches, and the percentage increase, respectively. We find that for existing approaches [Emmi et al. 2007; Taneja et al. 2010], they achieve the same code coverage for all the methods. The reason being that they use the same design decision when conducting the generation for database records, interacting with either a real database [Emmi et al. 2007] or a mock database [Taneja et al. 2010]. 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. The fourth part (Columns 7–8) shows the

Table VIII. Evaluation Results on UnixUsage

No.	method	total (blocks)	covered(blocks) others	covered(blocks) SynDB	covered(blocks) increase	time(seconds) SynDB	time(seconds) MODA
1	courseNameExists	7	7	7	0%	5.0	14.1
2	getCourseIDByName	10	10	10	0%	7.1	14.8
3	computeFileToNetworkRatio ForCourseAndSessions	25	8	25	68.00%	13.7	23.3
4	outputUserName	14	14	14	0%	9.8	26.6
5	computeBeforeAfterRatioByDept	24	24	24	0%	58.3	92.0
6	getDepartmentIDByName	11	11	11	0%	13.6	28.0
7	computeFileToNetworkRatioForDept	21	21	21	0%	38.3	66.2
8	raceExists	11	7	11	36.36%	12.9	31.1
9	userIdExists(version1)	11	7	11	36.36%	13.3	30.3
10	transcriptExist	11	7	11	36.36%	13.9	30.8
11	getTranscript	6	5	6	16.67%	7.7	16.1
12	commandExists(version1)	10	10	10	0%	7.9	28.6
13	getCommandsByCategory	10	10	10	0%	7.3	31.2
14	retrieveUsageHistoriesById	21	7	21	66.67%	14.8	39.0
15	userIdExists(version2)	11	7	11	36.36%	10.2	22.4
16	commandExists(version2)	11	11	11	0%	10.1	21.7
17	retrieveMaxLineNo	10	7	10	30.00%	11.8	25.9
18	retrieveMaxSequenceNo	10	7	10	30.00%	12.3	24.7
19	getSharedCommandCategory	11	7	11	36.36%	11.1	27.1
20	getUserInfoBy	47	15	47	68.09%	52.2	time out
21	doesUserIdExist	10	9	10	10.00%	6.6	16.8
22	getPrinterUsage	34	27	34	20.59%	21.9	44.4
all methods (total)		336	238	336	29.17%	359.8	775.1

running time cost by MODA and our approach. In our evaluation, we set the `TimeOut` as 120 seconds for Pex.

Within the `iTRUST` application, the 14 methods contain 343 code blocks in total. Tests generated by existing approaches cover 262 blocks while our approach can cover 315 blocks (15.45% average increase). 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 (21.10% average increase). 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 (29.17% average increase).

We observe that tests generated by existing approaches fail to cover certain blocks for some methods. The reason being 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 previously mentioned 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 V. 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).

For `iTRUST`, the accompanied schema constraints are more comprehensive than the constraints for the other two applications. The results show that our approach can

```
01:public String getRole(long mid, String role) throws iTrustException, DBException {
02:  Connection conn = null;
03:  PreparedStatement ps = null;
04:  try {
05:    conn = factory.getConnection();
06:    ps = conn.prepareStatement("SELECT role FROM Users WHERE MID=? AND Role=?");
07:    ps.setLong(1, mid);
08:    ps.setString(2, role);
09:    ResultSet rs;
10:    rs = ps.executeQuery();
11:    if (rs.next())
12:      return rs.getString("role");
13:    else
14:       throw new iTrustException("User does not exist with the designated role");}
15:  catch (SQLException e){
16:    e.printStackTrace();
17:    throw new DBException(e);}
18:  finally{
19:    DBUtil.closeConnection(conn, ps);}}
```

Fig. 12. Method `getRole` from `iTRUST`.

generate tests to cover more blocks for all the 14 methods under test. For example, for the No.7 method `getRole` shown in Table VI, the original code is shown in Figure 12. One of its parameter `String role` is combined into a SQL query that selects records from the `Users` table. The attribute `Role` related with the variable `role` has a constraint shown in Table V, where the value can be chosen from only a predefined string set. Existing approaches fail to generate effective test inputs to cover Line 12, while our approach has captured such a constraint and is able to generate effective tests to cover more code blocks than existing approaches. Note that for this method, the not-covered code (Line 12) by existing approaches is directly related with query result manipulation, where such a case is different from the method shown in Figure 9. For the method shown in Figure 9, the not-covered code (Line 17) is related with the variable manipulated within the query result manipulation (Line 12). The experimental results show that our approach can handle both cases, because our approach has correlated the constraints from the query result manipulation and later program execution.

For `RiskIt` and `UnixUsage`, we add only a small number of extra database schema constraints, where these constraints have not affected all the methods. 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 VII, 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 code coverage than existing approaches.

Comparing with existing approaches, the increased code coverage by our approach is from addressing the aforementioned two types of constraint conflicts caused by prematurely concretized SQL queries. In our evaluation, we observe that most of the increased code coverage comes from the methods under test that have the issue of the first type of constraint conflict. Only two methods (methods 6 and 8) from `iTrust`, one method (method 14) from `RiskIt`, and no methods from `UnixUsage` have the issue of the second type of constraint conflict. For the two methods (methods 6 and 8) from `iTrust`, all of the increased code coverage comes from addressing the second type of constraint conflict. For the method (method 14) from `RiskIt`, within the total increased code coverage, 15 code blocks are covered due to addressing the second type of constraint

conflict. Although there are a limited number of methods under test that have the issue of the second type of constraint conflict since the logic of most methods under test is simple, we believe such issues exist widely when the program code logic becomes more complex.

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` in Table VII, 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 `stockDividents`) 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 approaches.

4.3.2. Cost. We observe that the major factor that impacts the execution 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. In the implementation of our approach, we call one of Pex's API methods `PexAssume()` to reduce the cost of exploring constraints. `PexAssume()` is for filtering out undesirable test inputs. By calling `PexAssume()`, it is beneficial to guarantee that database schema constraints are always enforced without unnecessary negations. Although we can apply the Pex API method to optimize the constraint-checking process, the cost still increases when more tables are involved. For example, for `RiskIt`, methods 1 and 16 have similar code logics except that the query in method 1 joins three tables, while method 16 deals with only one table called `userrecord`. We observe that the execution on method 1 (27.9 seconds) costs much more time than method 16 (21.3 seconds).

Meanwhile, if a table contains a large number of attributes, high cost is also incurred, because more values have to be generated for table columns. For example, for method 3 in `RiskIt`, the original query selects all columns from the `userrecord` table. To evaluate the performance when the size of table columns changes, we double the number of its columns by adding more attribute names. We observe that the execution costs more time than the original, increasing from 14.2 seconds to 17.5 seconds.

Another observation from the running cost that we would like to point out is related to Pex's path exploration. Complexity of the qualification in a query also affects the analysis time, as evaluating the conditions has been transformed into normal program code in our approach. 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 force Pex to consider all the conditions together in one time, still calling Pex's API method `PexAssume()`.

Since the existing approach [Emmi et al. 2007] is not publicly available and we simulate its functionalities by using our `SynDB` framework, we ignore reporting the running time for this approach. We report the analysis cost of our approach compared with MODA in Tables VI, VII, and VIII. Columns 7 and 8 show the running time for each method. For example, the running time of our approach for method `addUser` in `iTRUST` is 17.1 seconds, while MODA uses 33.3 seconds to conduct test generation. On

average, for all three applications, the results show that our approach uses much less running time than MODA.

5. RELATED WORK

Testing database applications has been attracting increasing attention recently. With the focus on functional testing, test generation is a fundamental technique. Emmi et al. [2007] 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. [2010] 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 [Pan et al. 2011a] uses DSE to generate database states to achieve advanced structural coverage criteria. Li and Csallner [2010] propose an approach to exploit existing databases to maximize the coverage under DSE.

Some approaches [Zhang et al. 2012; Tillmann and Schulte 2006] conduct test generation through modeling the environment and writing stub functions. Using stub functions can isolate the unit under test from the environment. However, for database applications, a significant problem of using stub functions is that program-execution constraints and environment constraints are also isolated. Our approach uses a fully symbolic database and passes it through synthesized database interactions. Hence all constraints within a database application are not isolated from each other. Our approach can guarantee that the generated tests are always valid.

Willmor and Embury [2006] propose an approach that builds a database state for each test case intensionally, where the pre-conditions and post-conditions to be satisfied for the test case are provided in a query by the user. Chays et al. propose a set of tools [Chays et al. 2004; Deng and Chays 2005] for testing database applications by gathering information from the schema and application and by populating the database with useful values, as well as preparing test cases to check application's resulting database state and output. Chays et al. [2008] develop an approach that generates inputs to satisfy certain properties specified by the tester. The approach generates test queries based on the SQL statements from the application. The AGENDA approach [Chays et al. 2004, 2008; Deng and Chays 2005] 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. Another problem is that it is not guaranteed that the execution of the test query on the generated database states can produce the desired query results. The QAGen [Binnig et al. 2007b] 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. [2007a] 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. Khalek et al. [2008] conduct black-box testing of database management systems (DBMS). They develop an ADUSA prototype to generate database states and expected results of executing given queries on generated database states, given a database schema and an SQL query as input. Unlike using constraint solver, they use the Alloy Analyzer that uses SAT to generate data. Veanes et al. [2009] propose a test generation approach for given

SQL queries. The approach isolates each individual query to generate tests but does not consider the interactions among queries. Moreover, the approach requires explicit criteria from developers to specify what tests should be generated. Another approach [Pan et al. 2011b] 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, testing database applications also has other target requirements. Halford and Orso [2006] present a set of testing criteria named *command form coverage*. It is claimed that all command forms should be covered when issued to the associated database. Some other techniques [Zhou and Frankl 2009, 2011; de la Riva et al. 2010; Tuya et al. 2010; Gupta et al. 2010] focus on the mutation testing of database applications. Zhou and Frankl [2009] propose the JDAMA approach that conducts mutation testing for database applications. The approach evaluates the performance of mutant killing for given database states but cannot generate effective tests to kill mutants. Another approach [Cabal and Tuya 2004] addresses the problem of measuring the coverage of SQL queries and presents a tool that automates it. Authors propose two different coverage measures for the coverage of SQL queries, specifically for the case of the SELECT query. However, it is still imperative to integrate SQL coverage criteria with other criteria for program languages. Based on a given SQL query, de la Riva et al. [2010] propose an approach that generates database records to satisfy the constraints obtained from the query's mutants, following predefined transformation rules. However, the approach does not consider program constraints and cannot deal with database applications directly. Our approach can be extended to satisfy other testing requirements, such as mutation testing, as long as we have correlated various constraints within a database application. For performance testing, the PPGen approach [Wu et al. 2005, 2007] 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 article, we propose a DSE-based approach called SynDB for testing database applications. The approach synthesizes new database interactions to replace the original ones. 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.

REFERENCES

Rafi Ahmed, Allison W. Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zaït, and Thierry Cruanes. 2006. Cost-based query transformation in Oracle. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 1026–1036.

Carsten Binnig, Donald Kossmann, and Eric Lo. 2007a. Reverse query processing. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 506–515.

Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. 2007b. QAGen: Generating query-aware test databases. In *Proceedings of the ACM SIGMOD Conference*. 341–352.

María José Suárez Cabal and Javier Tuya. 2004. Using an SQL coverage measurement for testing database applications. In *Proceedings of the ACM SIGSOFT Sympsium on the Foundations of Software Engineering (SIGSOFT-FSE)*. 253–262.

David Chays, Yuetang Deng, Phyllis G. Frankl, Saikat Dan, Filippos I. Vokolos, and Elaine J. Weyuker. 2004. An AGENDA for testing relational database applications. *Softw. Test. Verif. Reliab.* 14, 1, 17–44.

David Chays, John Shahid, and Phyllis G. Frankl. 2008. Query-based test generation for database applications. In *Proceedings of the International Workshop on Testing Database Systems (DBTest)*. 6.

Sarah R. Clark, Jake Cobb, Gregory M. Kapfhammer, James A. Jones, and Mary Jean Harrold. 2011. Localizing SQL faults in database applications. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 213–222.

L. A. Clarke. 1976. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.* 2, 3, 215–222.

Michael A. Cusumano and Richard W. Selby. 1997. How Microsoft builds software. *Commun. ACM* 40, 6 (1997), 53–61.

U. Dayal. 1987. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 197–208.

Claudio de la Riva, María José Suárez Cabal, and Javier Tuya. 2010. Constraint-based test database generation for SQL queries. In *Proceedings of the International Workshop on Automation of Software Test (AST)*. 67–74.

Yuetang Deng and David Chays. 2005. Testing database transactions with AGENDA. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 78–87.

Michael Emmi, Rupak Majumdar, and Koushik Sen. 2007. Dynamic test input generation for database applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 151–162.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the Programming Language Design and Implementation (PLDI)*. 213–223.

R. L. Goodstein. 2007. *Boolean Algebra*. Dover Publications.

Mark Grechanik, Christoph Csallner, Chen Fu, and Qing Xie. 2010. Is data privacy always good for software testing? In *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*. 368–377.

Bhanu Pratap Gupta, Devang Vira, and S. Sudarshan. 2010. X-data: Generating test data for killing SQL mutants. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 876–879.

William G. J. Halfond and Alessandro Orso. 2006. Command-form coverage for testing database applications. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 69–80.

Shadi Abdul Khalek, Bassem Elkarablieh, Yai O. Laleye, and Sarfraz Khurshid. 2008. Query-aware test generation using a relational constraint solver. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 238–247.

Won Kim. 1982. On Optimizing an SQL-like nested query. *ACM Trans. Datab. Syst.* 7, 3 (1982), 443–469.

J. C. King. 1976. Symbolic execution and program testing. *Commun. ACM*, 19, 7, 385–394.

Chengkai Li and Christoph Csallner. 2010. Dynamic symbolic database application testing. In *Proceedings of the International Workshop on Testing Database Systems (DBTest)*. 1–6.

Microsoft. 2007. Pex: Dynamic analysis and test generation for .NET. Microsoft Research Foundation of Software Engineering Group.

Microsoft. 2012a. DataTable. Microsoft MSDN. http://msdn.microsoft.com/en-us/library/system.data.datatable.aspx. (Last accessed May 2012).

Microsoft. 2012b. .NET framework data provider for SQL server. Microsoft MSDN. (May 2012). http://msdn.microsoft.com/en-us/library/system.data.sqlclient.aspx. (Last accessed May 2012).

Kai Pan, Xintao Wu, and Tao Xie. 2011a. Database state generation via dynamic symbolic execution for coverage criteria. In *Proceedings of the International Workshop on Testing Database Systems (DBTest)*. 1–6.

Kai Pan, Xintao Wu, and Tao Xie. 2011b. Generating program inputs for database application testing. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 73–82.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 263–272.

Kunal Taneja, Mark Grechanik, Rayid Ghani, and Tao Xie. 2011. Testing software in age of data privacy: A balancing act. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 201–211.

Kunal Taneja, Yi Zhang, and Tao Xie. 2010. MODA: Automated test generation for database applications via mock objects. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 289–292.

G. Tassey. 2002. The economic impacts of inadequate infrastructure for software testing. Tech. Repport. NIST Planning. 02-3, National Institute of Standards and Technology.

Nikolai Tillmann and Jonathan de Halleux. 2008. Pex-White Box test generation for .NET. In *Proceedings of the International Conference on Tests and Proofs (TAP)*. 134–153.

Nikolai Tillmann and Wolfram Schulte. 2006. Mock-object generation with behavior. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 365–368.

Javier Tuya, María José Suárez Cabal, and Claudio de la Riva. 2010. Full predicate coverage for testing SQL database queries. *Softw. Test. Verif. Reliab.* 20, 3, 237–288.

Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. 2009. Symbolic query exploration. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM)*. 49–68.

David Willmor and Suzanne M. Embury. 2006. An intensional approach to the specification of test cases for database applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 102–111.

Xintao Wu, Chintan Sanghvi, Yongge Wang, and Yuliang Zheng. 2005. Privacy aware data generation for testing database applications. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*. 317–326.

Xintao Wu, Yongge Wang, Songtao Guo, and Yuliang Zheng. 2007. Privacy preserving database generation for database application testing. *Fundam. Inform.* 78, 4, 595–612.

Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 359–368.

Linghao Zhang, Xiaoxing Ma, Jian Lu, Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. 2012. Environment modeling for automated testing of cloud applications. *IEEE Softw.* Special Issue on Software Engineering for Cloud Computing 29, 2, 30–35.

Chixiang Zhou and Phyllis G. Frankl. 2009. Mutation testing for Java database applications. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 396–405.

Chixiang Zhou and Phyllis G. Frankl. 2011. Inferential checking for mutants modifying database states. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 259–268.

Received May 2012; revised January, May 2013; accepted June 2013