CONSTRAINT-BASED GENERATION OF DATABASE STATES FOR TESTING DATABASE APPLICATIONS

by

Kai Pan

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2012

Approved by:

Dr. Xintao Wu

Dr. Bill Chu

Dr. Weichao Wang

Dr. Mohamed Shehab

Dr. Jiang (Linda) Xie

©2012
Kai Pan
ALL RIGHTS RESERVED

ABSTRACT

KAI PAN. Constraint-based generation of database states for testing database applications. (Under the direction of DR. XINTAO WU)

Testing is essential for quality assurance of database applications. To test the quality of database applications, it usually requires test inputs consisting of both program input values and corresponding database states. However, producing these tests could be very tedious and labor-intensive in a non-automated way. It is thus imperative to conduct automatic test generation helping reduce human efforts.

The research focuses on automatic test generation of both program input values and corresponding database states for testing database applications. We develop our approaches based on the Dynamic Symbolic Execution (DSE) technique to achieve various testing requirements. We formalize a problem for program-input-generation given an existing database state to achieve high program code coverage and propose an approach that conducts program-input-generation through auxiliary query construction based on the intermediate information accumulated during DSE's exploration. We develop a technique to generate database states to achieve advanced code coverage criteria such as Boundary Value Coverage and Logical Coverage. We develop an approach that constructs synthesized database interactions to guide the DSE's exploration to collect constraints for both program inputs and associated database states. 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 develop an approach that generates tests for mutation testing on database applications. We use a state-of-the-art white-box testing tool called Pex for .NET from Microsoft Research as the DSE engine. Empirical evaluation results show that our approaches are able to generate effective program input values and sufficient database states to achieve various testing requirements.

ACKNOWLEDGMENTS

I especially would like to thank my advisor, Dr. Xintao Wu, for supporting me with guidance throughout preparing this dissertation. Dr. Wu has provided me professional and valuable advice during my past Ph.D. study. The memorable experience working with Dr. Wu will also continue to benefit my future career. I am also grateful to have Dr. Bill Chu, Dr. Weichao Wang, Dr. Mohamed Shehab, and Dr. Jiang (Linda) Xie as my dissertation committee. I would like to thank their great help with my research work. I also want to thank our research collaborator Dr. Tao Xie who has provided insightful guidance in helping shape and revise my research ideas and writing.

I want to thank the people who provided tools, applications, and experimental subjects used in my research. The Pex team from Microsoft Research released the tool Pex, which was used as our major testing tool. Xusheng Xiao and Kunal Taneja introduced the experimental subjects used in our papers. Many anonymous reviewers kindly provided their valuable feedback to help improve my papers that this dissertation is based on.

I want to thank Graduate Assistant Support Plan in the University of North Carolina at Charlotte for the financial support. My research work is also supported in part by U.S. National Science Foundation under CCF-0915059.

Many colleagues and friends offered me great helps and encourage to my research work and daily life. I want to thank Xiaowei Ying, Leting Wu, Ling Guo, Jun Zhu,

Yue Wang, and Zhilin Luo. Last but not least, I am grateful to my family. They have given me continuous love and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Database applications are ubiquitous nowadays and have influenced many aspects of human life. It is critical to assure high quality of database applications before they are deployed. As testing database applications can be classified into various categories: functional testing, performance testing (load and stress, scalability), security testing, environment and compatibility testing, and usability testing, a fundamental task is to conduct test generation. Specially, for the purpose of testing database applications, the test generation consists of both program input values and associated database states. However, manually producing these tests could be very time-consuming and labor-intensive. In this research, we focus on developing automated test-generation approaches under various problem contexts, aiming to satisfy multiple testing requirements.

To reduce laborious human effort, researchers have developed techniques and tools to automate the software testing activities. Recently, the Dynamic Symbolic Execution (DSE) technique was proposed and tools for different programming languages were developed (e.g., C [49], Java [38], and C# [1]). DSE has also been applied on testing database applications [38, 53]. However, for testing database applications, under different problem contexts, there are drawbacks and limitations within existing

approaches. In addition, as code coverage is considered to be a good indicator to expose potential faults in program code, test generation approaches aiming to satisfy multiple testing requirements are still missing.

1.1 Background

Automated test data generation techniques could be divided into two main categories: static and dynamic. For the dynamic class, recently, *dynamic symbolic execution* (DSE) (or concolic testing, as a portmanteau of *concrete* and *symbolic*) was proposed [24, 49]. In the traditional symbolic execution [35, 16], a program is executed symbolically with symbolic inputs rather than concrete inputs. DSE extends the traditional symbolic execution by running a program with concrete inputs while collecting both concrete and symbolic information at runtime, making the analysis more precise [24].

DSE first starts with default or arbitrary inputs and executes the program concretely. Along the execution, DSE simultaneously performs symbolic execution to collect symbolic constraints on the inputs obtained from predicates in conditions. DSE flips a branch condition and conjuncts the negated branch condition with constraints from the prefix of the path before the branch condition. DSE then feeds the conjuncted conditions to a constraint solver to generate new inputs to explore not-yet-covered paths. The whole process terminates when all the feasible program paths have been explored or the number of explored paths has reached the predefined upper bound.

DSE has also been used in testing database applications [38, 53]. Emmi et al. [38] developed an approach for automatic test generation based on DSE. Their approach uses a constraint solver to solve collected symbolic constraints to generate both program input values and corresponding database records. The approach involves running the program simultaneously on concrete program inputs as well as on symbolic inputs and a symbolic database. In the first run, the approach uses random concrete program input values, collects path constraints over the symbolic program inputs along the execution path, and generates database records such that the program execution with the concrete SQL queries can cover the current path. To explore a new path, it flips a branch condition and generates new program input values and corresponding database records.

To address the issue that the associated databases are not in place during test generation, Taneja el al. [53] proposed the MODA framework that establishes a mock database to mimic the real database. Within the MODA framework, operations on the real database are performed over the mock database. The approach is implemented upon Pex [55] and extends Pex with two capabilities: first, the transformed code helps Pex handle database interactions; second, the approach inserts the generated records into the mock database to help prepare the initial database state.

1.2 Challenges and Solutions

Existing database application testing approaches usually employ reaching a high program code coverage as a main objective. However, it is also imperative to enforce

advanced structural coverage criteria such as *Logical Coverage* (LC) and *Boundary Value Coverage* (BVC) for effective testing. In particular, BVC requires to execute programs using values from both the input range and boundary conditions and requires multiple test inputs at boundaries [36]. The reason is that errors tend to occur at extreme or boundary points. LC criteria involve instantiating clauses in a logical expression with concrete truth values. Researchers have focused on active clause coverage criteria to construct a test such that the value of a logical expression is directly dependent on the value of the clause that we want to test. Among these active clause coverage criteria, the *Correlated Active Clause Coverage* (CACC) [3] is equivalent to *masking Modified Condition/Decision Coverage* (MC/DC), of which the MC/DC has been chosen by US Federal Aviation Administration [14] as a recommended test-generation criterion among logical criteria.

In this research, we develop an approach on how to generate sufficient database states to satisfy advanced coverage criteria including BVC and CACC criteria. We investigate the close relationship between program inputs and executed queries. We also make sure the generated database states can lead the later program executions satisfy those advanced coverage criteria after query result set is returned.

Existing approaches often focus on generating database states from scratch. In practice, there may exist a copy of live databases that can be used for database application testing. Using an existing database state is desirable since it tends to be representative of real-world objects' characteristics, helping detect faults that could

cause failures in real-world settings. On the other hand, to cover a specific program code portion (e.g., block), appropriate program inputs also need to be generated for the given existing database state. However, it often happens that a given database with an existing database state (even with millions of records) returns no records (or returned records do not satisfy branch conditions in the subsequently executed program code) when the database receives and executes a query with arbitrarily chosen program input values. Hence, there is a significant challenge in our problem context: there exists a gap between program-input constraints derived from the program and those derived from the given existing database state; satisfying both types of constraints is needed to cover a specific program code portion. During DSE, these two types of constraints cannot be naturally collected, integrated, or solved for test generation.

In this research, we formalize a problem for program-input-generation given an existing database state to achieve high program code coverage and propose an approach that conducts program-input-generation through auxiliary query construction based on the intermediate information accumulated during DSE's exploration. We deal with the problem of extending DSE to handle database applications. We propose an algorithm that, given a path condition that cannot be covered(because it depends on a particular result of an SQL query executed on the path), constructs auxiliary queries to find database entries that are partially consistent with the program state, and then uses a constraint solver to construct new input values that are consistent

with the path conditions and the existing database state. We expect the proposed approach results in an improved program code coverage.

Another problem context is that, sometimes, the real database is not physically available. The MODA framework [53] uses a mock database, replacing the real database, that the tests can be executed with. As aforementioned, existing approaches [38, 53] often use constraints from concrete queries observed at runtime to conduct test generation. The generated records are then inserted back to the database(either real database or mock database). However, it often happens that the concrete queries may contain program inputs directly or after a chain of computations. Constraints obtained from the concrete queries derived from DSE's exploration may conflict with other constraints (e.g. query-result-manipulation constraints and database schema constraints). There exists a gap between the associated database state and the query result set, which may lead to conflicts and failure of high program code coverage.

In this research, we develop an approach called *SynDB* that treats symbolically both the associated database state and the embedded query by constructing synthesized database state and synthesized database interactions. We transform the original code under test into another form that the synthesized database interactions can operate on. After the code transformation, the synthesized database interactions integrate the query constraints into normal program code. The synthesized database interactions guide the DSE's exploration to collect constraints for both program inputs and associated database states. In this way, we correlate various kinds of constraints

within a database application.

Meanwhile, we extend our research to another testing aspect called mutation testing that is to assess and improve the quality of test inputs. Mutation testing is a fault-based software testing technique that is intensively studied for evaluating the adequacy of tests [19]. The original program under test is mutated into a set of new programs, called *mutants*, caused by syntactic changes following a set of rules. The mutants are *(strongly) killed* if running the mutants against given tests produces different results than the results of the original program. Killing more mutants reflects better adequacy and higher reliability of the tests under assessment. However, automatically producing tests that can kill mutants could be very time-consuming and even intractable [20], because a short program may contain a large number of mutants.

In this research, we develop an approach called *MutaGen* for killing mutants in database applications based on our previous *SynDB* framework [46]. To generate mutants that occur in the program code, we apply an existing code-mutation tool [70] on the code transformed with the SynDB framework. To generate SQL-query mutants, we apply an existing SQL-query-mutation tool [57] to generate SQL-query mutants at query-issuing points. We then derive query-mutant-killing constraints considering both the original query and its mutants. We finally incorporate the derived constraints into the transformed code. Specifically, solving these query-mutant-killing constraints helps produce a database state on which running the original query and its mutants

can cause different query results, thus killing the corresponding SQL-query mutants.

1.3 Tools and Subject Applications

Various DSE tools for different languages have been developed (e.g., C [49], Java [38], and C# [1]). Throughout this research, we use a state-of-the-art white-box testing tool called Pex [1] for .NET from Microsoft Research as the DSE engine. Pex uses DSE to explore feasible execution paths of the program under test. Pex also allows testers to predefine thresholds regarding various settings (e.g., running time, number of runs), so that the DSE procedure could be terminated if the predefined thresholds are exceeded at runtime.

Pex contains a built-in constraint solver called Z3 [1]. Z3 is a high-performance theorem prover being developed at Microsoft Research. The constraint solver Z3 supports linear real and integer arithmetic, fixed-size bit-vectors, extensional arrays, uninterpreted functions, and quantifiers. The powerful functionalities of Z3 support generating effective tests for Pex.

We mainly conduct the empirical evaluations on three open source database applications: `iTRUST`[2], `RiskIt`[3], and `UnixUsage`[4]. These applications contain comprehensive programs and have been previously widely used as evaluated applications (`iTRUST` [15], `RiskIt` and `UnixUsage` [27, 52]).

`iTRUST` is a class project created at North Carolina State University for teaching

[1] http://research.microsoft.com/en-us/um/redmond/projects/z3/
[2] http://agile.csc.ncsu.edu/iTrust
[3] https://riskitinsurance.svn.sourceforge.net
[4] http://sourceforge.net/projects/se549unixusage

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. Programs in `UnixUsage` have about 2.8K non-commented lines of code and have a database containing 8 tables, 31 attributes, and more than 0.25 million records. 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[5]. 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[6].

1.4 Outline

The remainder of this dissertation is organized as follows: Chapter 2 surveys existing work related to our research. Chapter 3 presents a database state generation technique for advanced coverage criteria. Other than code coverage as a main goal, we aim to achieve advanced structural coverage criteria for generating database states. As these advanced structural coverage criteria focus on covering branch conditions with multiple values, we investigate how the constraints from the branch conditions

[5]http://sourceforge.net/projects/j2cstranslator/
[6]http://www.sis.uncc.edu/ xwu/DBGen

will have impact on generating associated database states. Chapter 4 presents an approach to generate effective program input values given an existing database state. While it is sometimes beneficial to use an existing database state to conduct testing activities, we solve the problem about how to generate corresponding appropriate program input values. We develop an approach that uses the intermediate information collected from DSE's exploration to construct auxiliary queries. After running the constructed auxiliary queries on the given database state, we attain effective program input values. Chapter 5 presents a DSE-based test-generation technique via synthesized database interactions. We observe that existing test database generation approaches often use constraints from concrete queries in the program code to conduct the generation task. However, such generation strategy may face a problem that there are conflicts between constraints from concrete queries and other constrains (e.g. query-result-manipulation constraints and database schema constrains). In this chapter, we present an approach that constructs synthesized database states and database interactions. We transform the constraints from both queries and database schema into normal program code. We bridge various constraints within a database application: query-construction constraints, query constraints, database schema constraints, and query-result-manipulation constraints. In this way, we guide DSE's exploration to track the constraints needed for generating both program input values and associated database states. Chapter 6 investigates test generation for mutation testing. Mutation testing aims to improve the quality of test inputs. We present an approach that

conducts test generation consisting of both program input values and database states so that these tests are able to reach a high mutation testing score. The approach is based on our *SynDB* framework [46]. We generate mutants for both program code and SQL queries. Query-mutant-killing constraints are derived and incorporated into the transformed code generated by *SynDB*. The transformed code is able to guide DSE to collect constraints and generate tests for killing both program-code mutants and SQL-query mutants. Chapter 7 concludes the dissertation.

CHAPTER 2: RELATED WORK

Database application testing has attracted much attention recently. This chapter reviews background information. We also discuss how our research relates with existing work about database application testing.

2.1 Testing Database Applications with DSE

Emmi et al. [38] developed an approach for automatic test generation for a database application. Their approach is based on DSE and uses symbolic constraints in conjunction with a constraint solver to generate both program inputs‘ and database states. We develop an approach that leverages DSE to generate database states to achieve advanced structural coverage criteria. On the other hand, this approach inserts records into the database during the DSE process, thus it necessarily requires the database to be in place. In our research, we focus on program-input generation given an existing database state, avoiding the high overhead of generating new database states during test generation. We also develop an approach that uses synthesized database states and interactions even if the physical database is not available. Li and Csallner [37] considered a similar scenario, i.e., how to exploit existing databases to maximize the coverage under DSE. However, their approach constructs a new query by analyzing the current query, the result tuples, the covered and not-covered

paths, and the satisfied and unsatisfied branch conditions. It can neither capture the close relationship between program inputs and results of SQL queries, nor generate program inputs to maximize code coverage.

Taneja et al. [53] developed the MODA framework that is applicable when the database is not available. The approach uses the mock database that the tests can be executed with. Also, tests generated by MODA can lower the false warnings than previous mocking techniques. Our proposed approach focuses on the problem, which is often neglected by existing approaches, that the embedded queries could contain program parameters. There may not be enough constraints before the embedded queries. Thus, when considering the constraints from the concrete queries, we may face the conflicts with either schema constraints or the late executed program path conditions. Meanwhile, our approach does not require the database to be in place, which can improve the executing efficiency. Some approaches [69, 56] 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 the generated tests are always valid.

2.2 Testing Database Applications with Other Techniques

The AGENDA project [11, 12, 22] addressed how to generate test inputs to satisfy basic database integrity constraints and does not consider parametric queries or constraints on query results during input generation. One problem with AGENDA is that it cannot guarantee that executing the test query on the generated database states can produce the desired query results. Bati et al. developed an approach [4] that uses random inputs to be the tests for testing database applications. However, the control-flow and data-flow information within an application could be ignored, leading that a great number of redundant tests will be generated, of which only a small number of tests are valid and effective to cover feasible program paths.

Willmor and Embury [63] developed an approach that builds a database state for each test case intensionally, in which the user provides a query that specifies the pre- and post-conditions for the test case. The approach is able to reduce human effort while predefining effective constraints for generating database states are still challenging. Binnig et al. [10, 5] extended symbolic execution and used symbolic query processing to generate some query-aware databases. However, the test database states generated by their QAGen prototype system [10] mainly aim to be used in database management systems (DBMS) testing. QAGen can generate a test database that guarantees the size of the intermediate join results to test the accuracy of the cardinality estimation components or a test database that guarantees the input and the output sizes for an aggregation operator in order to evaluate the performance of

the aggregation algorithm. However, QAGen considers only cardinality constraints on query results, which are not sufficient to support analyzing various kinds of constraints within a database application. Binnig et al. [9, 6] proposed the approach of *Reverse Query Processing* (RQP) that considers cases where the query-execution result is given. It can generate database states prior to the execution of a single query by considering the query itself as well as the results after the execution. 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. [51] conducted black-box testing of database management systems (DBMS). They developed 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, ADUSA uses the Alloy Analyzer that uses SAT to generate data. Veanes et al. [61] proposed 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.

2.3 Other Testing Criteria

Various coverage criteria [3, 36] have been proposed to generate test inputs for traditional (non-database) applications. BVC and LC are criteria that complement the widely used branch coverage. Most recently, Pandita et al. [48] developed a

general approach that instruments transformed branch conditions to source code and guides DSE to reach high BVC and LC for traditional applications. However, those criteria have not been directly supported in testing database applications.

Other than achieving high program code coverage that benefits in exposing program faults, testing database applications also has other target requirements. Focusing on an isolated SQL statement, Tuya et al. [60] proposed a coverage criterion, SQLFpc (short for SQL Full Predicate Coverage), based on the Modified Condition/Decision Coverage. Their approach mutates a given SQL query statement into a set of queries that satisfy MC/DC with the aim of detecting faults in the SQL query statement. Riva et al. [18] developed a tool to generate data to enforce SQLFpc for a SQL statement. Our research work leaves the embedded SQL statement unchanged. Instead, we generate database states with the aim of detecting faults in source code. Cabal and Tuya [8] considered the SQL statement coverage as the major target and proposed an approach that can improve the test data. Kapfhammer and Soffa [33] defined a set of testing criteria for database applications by incorporating the dependency of database information. Gould et al. [26] proposed an approach that conducts static type-checking for SQL queries generated by a Java program. Such verification of dynamically generated SQL strings could be incorporated into database applications for fault localization. Halford and Orso [30] presented 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. Tang et al. [54] defined the test

data adequacy criteria for database applications. However, these criteria could not be applied straightforwardly for generating tests without effectively analyzing various constraints within database applications.

2.4 Other Testing Aspects

There are also studies on other testing aspects regarding functional testing on database applications, such as mutation testing and performance testing.

Some approaches focus on the purpose of performance testing [7, 66, 64, 65]. Wu et al. developed the PPGen prototype system [66, 64, 67, 65] that can generate mock databases by reproducing the statistical distribution of realistic database states, so that the privacy issues could be addressed when the real database is not available. However, PPGen assumes constraints are explicit and focuses on SQL workload's performance testing. Our proposed approach has captured the relationship between the database state and the manipulation of the query result set. Thus, we can estimate the performance of a database application by specifying various distribution properties. Zhang et al. [71] addressed the problem that load testing often ignores considering particular input values of test inputs. The proposed approach applies a mixed symbolic execution to generate test suites that can cause different response times and memory consumptions.

Some other techniques focus on mutation testing for database applications. Tuya et al. [57, 59] proposed a set of mutation operators for SQL queries and a tool called SQLMutation that implements these mutation operators to generate SQL-query mu-

tants. To assess the adequacy of tests for Java database applications, Zhou et al. [72] developed a tool called JDAMA based on the mutation operators for SQL queries [59]. Zhou and Frankl [73] proposed the inferential checking technique for determining whether SQL mutants of updating statements are killed, without actually executing the mutants. The approach mainly considers the INSERT, DELETE, and UPDATE statements, complementing previous techniques that usually only focused on SELECT statements. Shahriar et al. [50] investigated the mutation of SQL updating statements under the problem context of mutation testing for evaluating vulnerability to SQL injection attacks. However, the approach does not consider how to generate tests that can cause the invoke of vulnerabilities.

There is also a large body of research work focusing on other testing scenarios for database application testing, e.g., generating tests for parallel testing [28, 31], database schema validation [23], SQL query generation [58, 41], and regression testing [62, 29]. These studies are all complementary to this research.

CHAPTER 3: GENERATE DB STATE FOR COVERAGE CRITERIA

In database applications, close relationships exist among program inputs, host variables, branch conditions, embedded SQL queries, and database states. For example, program inputs and host variables often appear in the embedded SQL queries and branch conditions in source code after executing SQL queries are often logical expressions that involve comparisons with retrieved values from database states. It is imperative to enforce advanced structural coverage criteria such as *Logical Coverage* (LC) and *Boundary Value Coverage* (BVC) for effective testing. In particular, BVC requires to execute programs using values from both the input range and boundary conditions and requires multiple test inputs at boundaries [36]. The reason is that errors tend to occur at extreme or boundary points. LC criteria involve instantiating clauses in a logical expression with concrete truth values. Researchers have focused on active clause coverage criteria to construct a test such that the value of a logical expression is directly dependent on the value of the clause that we want to test. Among these active clause coverage criteria, the *Correlated Active Clause Coverage* (CACC) [3] is equivalent to *masking Modified Condition/Decision Coverage* (MC/DC), of which the MC/DC has been chosen by US Federal Aviation Administration [14] as a recommended test-generation criterion among logical criteria.

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

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

3.1 Illustrative Example

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

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

Table 1: Database schema

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

3.2 Problem Formalization and Proposed Solution

To test the program `calcStat` in the preceding example or the entire database application, we need to generate sufficient database states as well as desirable values for program inputs. The input parameters often determine the embedded SQL statement in Line 14 and the database states determine whether the branches in Lines 16, 20, and 22 can be entered. Our approach uses Dynamic Symbolic Execution(DSE) [24, 49] to track how the inputs to the program under test are transformed before appearing in the executed queries and how the constraints on query results affect the

later program execution. We use Pex [1], a state-of-the-art DSE tool for .NET to illustrate our idea.

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

When the execution along one path terminates, Pex has collected all the preceding path constraints to form the path condition. Pex also provides a set of APIs that help access the intermediate information of its DSE process[7]. For example, with `type = 0, inputAge = 30` as input values, for the path P where branch conditions in Lines 03, 08, 16, and 20 are *true*.

To satisfy the branch condition in Line 20, we need to generate a sufficient database state such that the execution of the embedded query returns sufficient records to satisfy the branch condition in Line 20. Approaches [38] have been proposed to generate both program inputs and suitable database states to cover a feasible path, including the executions depending on the executed query's returned result set.

Our work focuses on how to generate sufficient database states to satisfy advanced coverage criteria including BVC and CACC criteria. For example, the variable `inputAge` is indirectly involved in the embedded SQL through variable `fage` and is

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

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

3.3 Approach

3.3.1 Overview of Our Approach

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

Pex provides a package of APIs that help users fetch intermediate results inside its DSE process[8]. We mainly use the methods from the class `PexSymbolicValue`

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

(shortened as `PexS` thereafter). We insert a call to API method `PexS.ToString()` at each execution point to get the target query string. We call the API method `PexS.GetRelevantInputNames<Type>()` to detect the data dependency between this execution point and program input parameters. To retrieve the path condition at the execution point, we call the method `PexS.GetPathConditionString()`.

Throughout this section, we use path P (where branch conditions in Lines 03, 08, 16, and 20 are *true*) and program inputs (`type = 0, inputAge = 30`) to illustrate our algorithm. When the DSE process encounters one branch condition, we call the Pex API method `PexS.GetPathConditionString()` to get the path condition along this path. In our example, we get $PC = pc_1 \wedge pc_2 \wedge pc_3 \wedge pc_4$, where pc_1 = `(type == 0)`, pc_2 = `(inputAge > 25)`, pc_3 = `(results.Read() == true)`, and pc_4 = `((bal >= 250000 || employed && married) == true)`. Some branch conditions are related with the database state via the dynamically constructed embedded query or the query's returned result set. In our algorithm, we call the Pex API method `PexS.ToString(...)` to get the concrete executed query string Q and the symbolic query string Q_{sym}. We decompose Q_{sym} using the SQL parser[9] and get its clauses. We assume the embedded SQL query takes the canonical DPNF form [10]:

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

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

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

In the SELECT clause, there is a list of h strings where each may correspond to a column name or with arithmetic or string expressions over column names and constants following the SQL syntax. In the FROM clause, there is a from-list that consists of a list of tables. In the WHERE clause, there is a disjunction of conjunctions. Each condition (e.g., A11) is of the form *expression op expression*, where *op* is a comparison operator (=, <>, >, >=, <, <=) or a membership operator (IN, NOT IN) and *expression* is a column name, a constant or an (arithmetic or string) expression. In practice, the queries could be very complex such as sub-queries can appear in the WHERE clause. There are extensive studies [34] on how to map complex queries such as nested queries to their canonical forms.

In our example, we have the concrete query Q

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

and its corresponding symbolic string Q_{sym}

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

We observe that conditions in the WHERE clause often contain host variables from the program under test and some of those host variables may appear directly in branch conditions or are dependent on host variables in branch conditions. For example, the condition `C.age=:fAge` in the WHERE clause of `Qsym` contains host variable `fAge`. The `fAge`'s symbolic expression is calculated as `fage=inputAge+10` and host variable

`inputAge` involves in the branch condition pc_2 = `(inputAge > 25)`. Hence enforcing BVC on this branch condition incurs constraints on the generated data. Lines 5-20 in Algorithm 1 give details about how to derive the constraints C_Q by examining the conditions in the WHERE clause and the branch conditions before the query's execution. We discuss details in Section 3.3.3.

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

3.3.2 Instantiating a Predicate to Satisfy BVC and CACC

A predicate is an expression that evaluates to a boolean value. A predicate may consist of a list of clauses that are joined with logical operators (e.g., NOT, AND, OR). Each clause contains a boolean variable, a non-boolean variable that is compared with a constant or another variable via relational operators, or even a call to a function that

returns a boolean value. The predicate `bal>=250000||employed&&married` in branch condition pc_4 contains three clauses: a relational expression `bal>=250000`, a boolean variable `employed`, and another boolean variable `married`.

Test coverage is evaluated in terms of test criteria, as specified by test requirements. Test requirements are specific elements that must be satisfied or covered for software artifacts. Predicate coverage requires that for each p there are instantiations that evaluate p to be `true` and instantiations that evaluate p to be `false`. Clause coverage ensures that for each clause $c \in p$ there are instantiations that evaluate c to be `true` and instantiations that evaluate c to be `false`. Predicate coverage is equivalent to the branch coverage criterion for testing source code while clause coverage is equivalent to the condition coverage. However, neither predicate coverage nor clause coverage subsumes the other. To test both individual clauses and predicates, combinatorial coverage (also called Multiple Condition Coverage) is used to evaluate clauses to each possible combination of truth values. We can see, for a predicate p with n independent clauses, there are 2^n possible combinations; thus, combinatorial coverage is often infeasible in practice.

Active clause criteria such as CACC have been widely adopted to construct a test such that the value of the predicate is directly dependent on the value of the clause that we want to test. CACC is defined in previous work [3]: For each p and each major clause $c_i \in p$, choose minor clauses c_j, $j \neq i$ so that c_i determines p. There are two requirements for each c_i: c_i evaluates to true and c_i evaluates to false. The

values chosen for the minor clauses c_j must cause p to be true for one value of the major clause c_i and false for the other, that is, it is required that $p(c_i = \text{true}) \neq p(c_i = \text{false})$.

BVC requires multiple test inputs at boundaries [36] because errors tend to occur at extreme or boundary points. For different data types, various boundary values are considered. We use the integer data type for illustration. Suppose that the conditional statement takes the form $[AopB]$ where A is a variable's expression, B is a constant, and op is a comparison operator ($==, !=, >, >=, <, <=$). Depending on the comparison operator, we seek to choose values for A coming from the minimum boundary, immediately above minimum, between minimum and maximum (nominal), immediately below maximum, or the maximum boundary. We list the choices in Table 2. For other data types, we omit details.

Table 2: BVC enforcement for integer

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

Algorithm 2 shows how to generate instantiations satisfying both CACC and BVC for a given predicate. The algorithm accepts a predicate and a boolean evaluation as input and generates a list of instantiations as output. Lines 2-6 in Algorithm 2

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

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

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

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

bal=250000, employed=true, married=false,

bal>250000, employed=true, married=false,

bal=maximum, employed=true, married=false,

bal=250000-1, employed=true, married=true,

bal<250000-1, employed=true, married=true,

bal=minimum, employed=true, married=true.

3.3.3 Deriving Constraints of Database States

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

Conditional expressions in the WHERE clause contain constraints of the generated database state. Formally, we call a SQL parser to decompose Q_{sym} and set C_Q = {(A_{11} AND ... AND A_{1n}) OR ... OR (A_{m1} AND ... AND A_{mn})} from the Q_{sym}. We create an empty variable set V_Q. For each $A_{ij} \in C_Q$, we check whether A_{ij} contains host variables. For each contained host variable, if it is related with any branch condition before the query execution point T, we add it to V_Q and replace it with its symbolic string expressed by variables in the related branch conditions. If not, we replace the variable with its corresponding concrete value contained in the concrete

query Q. In our example, we get the constraints C_Q = {`M.year=:years AND C.SSN= M.SSN AND C.age=:fAge`}. For `M.year=:years`, we replace `years` with the value `15`. For `C.age=:fAge`, we replace `fAge` with `inputAge + 10`. We leave `C.SSN= M.SSN` unchanged since it does not contain any host variable. The variable set V_Q = {`fAge`} and C_Q = {`M.year=15 AND C.SSN= M.SSN AND C.age=:inputAge + 10`}.

Next, for each branch condition pc along this path, we check whether pc contains host variables related with the set V_Q. If yes, we enforce it with both CACC and BVC by calling Algorithm 2. Algorithm 2 returns instantiations I_{pc} that make pc =`true`. The returned instantiations incur further constraints on the generated database state. We refine C_Q as $C_Q = C_Q \times I_{pc}$.

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

M.year=15 AND C.SSN= M.SSN AND C.age=:inputAge+10 AND inputAge=25+1,

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

M.year=15 AND C.SSN= M.SSN AND C.age=:inputAge+10

AND inputAge=maximum.

Note that branch conditions after the SQL's execution point T may contain host variables that are dependent on database attributes in the SELECT clause. Enforcing CACC and BVC on those branch conditions further incur constraints on the gener-

ated database state. We first identify host variables V_R that are directly dependent on attributes of the returned query result set. For example, we get that the variable `bal` is directly dependent on the attribute `M.balance` in Line 17 in our illustrative example. Then, for each branch condition that contains host variables in or dependent on V_R, we treat *pc* as a predicate and call Algorithm 2 to enforce both CACC and BVC requirements. In our example, pc_4 = `((bal >= 250000 || employed && married) == true)` and V_R = {`bal, married, employed`}. The returned six instantiations are shown in the last paragraph of Section 3.3.2. Note that the returned instantiations contain host variables related with variables V_R. We need to replace them with their corresponding database attributes. In our example, host variables `bal, employed, married` are replaced with database attributes `M.balance, M.marriage, M.jobStatus`, respectively.

Finally, the set $C = C_Q \times C_R$ contains constraints on the generated database state to enforce CACC and BVC on the source code under test. We also collect basic constraints C_S at the database schema level (e.g., not-NULL, uniqueness, referential integrity constraints, domain constraints, and semantic constraints). For example, attribute `balance` in table `mortgage` must be greater than 0. We then send C together with the schema level constraints C_S to a constraint solver to conduct the data instantiation on the symbolic database. In our prototype system, we use the constraint solver Z3 [11], which is integrated into Pex. Z3 is a high-performance theorem prover being developed at Microsoft Research. The constraint solver Z3 supports linear real and integer arithmetic, fixed-size bit-vectors, extensional arrays, uninterpreted

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

functions, and quantifiers.

3.4 Evaluation

Our approach is to provide an assistance to the DSE-based test-generation tools (e.g., Pex [1, 55] for .NET) to improve code coverage with respect to CACC and BVC in database application testing.

We conduct evaluations on two open source database applications `RiskIt` and `UnixUsage`. The introductions to these two applications are previously presented in Chapter 1.

To set up the evaluation, we choose methods that have boundary values and/or logical expressions in branch conditions from the applications. Since there are no tools to measure CACC and BVC directly, we apply a tool [48] that transforms the problem of achieving CACC and BVC to the problem of achieving block coverage by introducing new blocks through code instrumentation. We use `PexGoal.Reached()` to identify whether each introduced block is covered.

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

08a: if (inputAge == 25+1)

{PexGoal.Reached()...}

08b: else if (inputAge > 25+1)

{PexGoal.Reached()...}

08: if (inputAge > 25){...}

after the code instrumentation.

Tables 4 shows the results of our evaluation. We use n to denote the number of `PexGoal.Reached()` statements introduced in each method. The coverage of all n introduced blocks indicates the full achievement of CACC and BVC. Given a database state, the current Pex cannot generate sufficient program inputs to achieve higher code coverage especially when program inputs are directly or indirectly involved in embedded SQL statements. In our experiment, we also apply our previous approach [44] to assist Pex generate sufficient input values for program parameters.

We first run Pex (in addition to our program input generation tool [44]) without applying Algorithm 1 to generate new records. We use n_1 to denote the number of covered `PexGoal.Reached()` statements in this experiment. We then apply Algorithm 1 to generate new database records and run Pex in addition to our program input generation tool [44]. We use n_2 to denote the number of covered `PexGoal.Reached()` statements during this step. The value of (n_2 - n_1)/n captures the increase gained by Pex assisted by our new approach in achieving CACC and BVC. We see from Tables 4 that the n_2 values are equal to n for all methods, indicating that our new approach assists Pex to reach the full CACC and BVC coverage (a 21.21% increase on average for `RiskIt` and a 46.43% increase for `UnixUsage`). The detailed evaluation subjects and results can be found on our project website[12].

[12]http://www.sis.uncc.edu/∼xwu/DBGen

Algorithm 1 Database State Generation in Achieving CACC and BVC

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

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

Algorithm 2 *EnforceCriteria*: CACC and BVC enforcement

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

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

Table 4: Evaluation results for Coverage Criteria

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

3.5 Conclusions

In this research, we present a general approach to generating test database states that can achieve program *Boundary Value Coverage*(BVC) and *Logical Coverage* such as *Correlated Active Clause Coverage*(CACC). We implemented our approach in Pex, a DSE tool for .NET. Our evaluation demonstrates the feasibility of our approach. By generating database states to meet the test requirements such as BVC and CACC, we expect that testers can detect more faults that occur in boundaries or involve complex logical expressions. Part of this research was published in DBTest 2011 [43].

In future work, we plan to investigate how to optimize the constraint collection and data instantiation. In practice, the number of generated database states for complex programs can grow exponentially as the number of logical conditions and boundary values grow. We plan to study complex SQL queries (e.g., GROUPBY queries with aggregations) and extend our technique to deal with a bag of multiple queries in database applications.

CHAPTER 4: GENERATE INPUTS USING EXISTING DB STATES

Testing is essential for quality assurance of database applications. Achieving high code coverage of the database application is important in testing. In practice, there may exist a copy of live databases that can be used for database application testing. Using an existing database state is desirable since it tends to be representative of real-world objects' characteristics, helping detect faults that could cause failures in real-world settings. However, to cover a specific program code portion (e.g., block), appropriate program inputs also need to be generated for the given existing database state. To address this issue, we develop a novel approach that generates program inputs for achieving high code coverage of a database application, given an existing database state. Our approach uses symbolic execution to track how program inputs are transformed before appearing in the executed SQL queries and how the constraints on query results affect the application's execution. One significant challenge in our problem context is the gap between program-input constraints derived from the program and from the given existing database state; satisfying both types of constraints is needed to cover a specific program code portion. Our approach includes novel query formulation to bridge this gap. We incorporate the data instantiation component in our framework to deal with the case that no effective program input values can be

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

Figure 2: An example code snippet from a database application under test

attained. We determine how to generate new records and populate them in the new database state such that the code along the path can be covered. We also extend our program-input-generation approach to test database applications including multiple queries. Our approach is loosely integrated into Pex, a state-of-the-art white-box testing tool for .NET from Microsoft Research. Empirical evaluations on two real database applications show that our approach assists Pex to generate program inputs that achieve higher code coverage than the program inputs generated by Pex without our approach's assistance.

4.1 Illustrative Example

The example code snippet shown in Figure 2 includes a portion of C# source code from a database application that calculates some statistic related to mortgages. The corresponding database contains two tables: `customer` and `mortgage`. Their schema-level descriptions and constraints are given in Table 5. The `calcStat` method described in the example code snippet receives two program inputs: `type` that determines the years of mortgages and `zip` that indicates the zip codes of customers. A variable `fzip` is calculated from `zip` and in our example `fzip` is given as "`zip`+1". Then the database connection is set up (Lines 08-10). The database query is constructed (Line 11) and executed (Lines 12 and 13). The tuples from the returned result set are iterated (Lines 14-20). For each tuple, a variable `diff` is calculated from the values of the `income` field and the `balance` field. If `diff` is greater than `100000`, a counter variable `count` is increased (Line 19) and `totalBalance` is updated (Line 20). The method finally returns the calculation result.

Both program inputs (i.e., input parameters) and database states are crucial in testing this database application because (1) the program inputs determine the embedded SQL statement in Line 11; (2) the database states determine whether the true branch in Line 14 and/or the true branch in Line 18 can be covered, being crucial to functional testing, because covering a branch is necessary to expose a potential fault within that branch; (3) the database states also determine how many times the loop body in Lines 14-20 is executed, being crucial to performance testing.

Table 5: Database schema

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

4.2 Problem Formalization and Proposed Solution

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

the query on the existing database will return no records, or those returned records do not satisfy the condition in Line 18. Hence, it is crucial to generate program input values such that test inputs with these values can help cover various code portions when executed on the existing database.

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

In particular, our approach uses dynamic symbolic execution (DSE)[49] to track how program inputs to the database application under test are transformed before appearing in the executed queries and how the constraints on query results affect the later program execution. We use DSE to collect various intermediate information.

Our approach addresses one significant challenge in our problem context: there exists a gap between program-input constraints derived from the program and those derived from the given existing database state; satisfying both types of constraints

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

As aforementioned, the DSE technique has also been used in testing database applications [38, 53]. Emmi et al. [38] developed an approach for automatic test generation based on DSE. Their approach uses a constraint solver to solve collected symbolic constraints to generate both program input values and corresponding database records. The approach involves running the program simultaneously on concrete program inputs as well as on symbolic inputs and a symbolic database. In the first run, the approach uses random concrete program input values, collects path constraints over the symbolic program inputs along the execution path, and generates database records such that the program execution with the concrete SQL queries can cover the current path. To explore a new path, it flips a branch condition and generates new program input values and corresponding database records. However, their approach cannot

generate effective program inputs based on the content of an existing database state. The reason is that some program inputs (e.g., `zip` in our illustrative example) appear only in the embedded SQL queries and there is no path constraint over them.

Our approach differs from Emmi et al.'s approach [38] in that we leverage DSE as a supporting technique to generate effective program input values by executing constructed auxiliary queries against the existing database state. As a result, high code coverage of the application can be achieved without generating new database states. When DSE is applied on a database application, DSE often fails to cover specific branches due to an insufficient returned result set because returned record values from the database often involve in deciding later branches to take. We use Pex [1], a DSE tool for .NET, to illustrate how our approach assists DSE to determine program input values such that the executed query can return sufficient records to cover various code portions. During the program execution, DSE maintains the symbolic expressions for all variables. When the execution along one path terminates, DSE tools such as Pex have collected all the preceding path constraints to form the path condition. Pex also provides a set of APIs that help access intermediate information of its DSE process. For illustration purposes, we assume that we have an existing database state shown in Table 6 for our preceding example shown in Figure 2.

To run the program for the first time against the existing database state, Pex uses default values for program inputs `type` and `zip`. In this example, because `type` and `zip` are both integers. Pex simply chooses "`type`=0, `zip`=0" as default values. The

Table 6: A Given Database State

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

condition in Line 04 is then satisfied and the query statement with the content in Line 11 is dynamically constructed. In Line 12 where the query is executed, we can dynamically get the concrete query string as

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

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

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

The execution of `Q1` on Table 6 yields zero record. Thus, the `while` loop body in Lines 14-20 is not entered and the exploration of the current path is finished. We use the Pex API method `PexSymbolicValue.GetPathConditionString()` after Line 14 to get the path condition along this path:

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

To explore a new path, Pex flips a part of the current path condition from "`type == 0`" to "`type != 0`" and generates new program inputs as "`type=1, zip=0`". The con-

dition in Line 04 is then not satisfied and the SQL statement in Line 11 is dynamically determined as

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

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

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

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

In testing database applications, previous test-generation approaches (e.g., Emmi et al. [38]) then invoke constraint solvers to generate new records and instantiate a new test database state, rather than using the given existing database state, required in our focused problem.

In contrast, by looking into the existing database state as shown in Table 6, we can see that if we use an input like "`type=0, zip=27694`", the execution of the query in Line 11 will yield one record {`C.SSN = 001, C.income = 50000, M.balance = 20000`},

which further makes Line 14 condition `true` and Line 18 condition `false`. Therefore, using the existing database state, we are still able to explore this new path:

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

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

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

4.3 Approach

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

The major idea of our approach is to construct an auxiliary query based on the intermediate information (i.e., the executed query's concrete string and its abstract form, symbolic expressions of program variables, and path conditions) collected by DSE. There are two major challenges here. First, program input values are often combined into the executed query after a chain of computations. In our illustrative example, we simply set `fzip = zip+1` in Line 3 to represent this scenario. We can see that `fzip` is contained in the WHERE clause of the executed query and `zip` is one pro-

gram input. Second, record values in the returned query result set are often directly or indirectly (via a chain of computations) involved in the path condition. In our illustrative example, the program variable `diff` in the branch condition `diff>100000` (Line 18) is calculated from the retrieved values of attributes `income` and `balance`. To satisfy the condition (e.g., `diff>100000` in Line 18), we need to make sure that the program input values determined by our auxiliary query are appropriate so that the query's return records are sufficient for satisfying later executed branch conditions.

4.3.1 Auxiliary Query Construction

Algorithm 3 illustrates how to construct an auxiliary query. The algorithm accepts as inputs a simple SQL query in its both concrete and abstract forms, program input values, and the current path condition.

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

are data-dependent on program inputs I. DSE also collects the concrete string of an executed query along the current path. In our approach, we assume the SQL query takes the form:

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

In the SELECT clause, there is a list of h strings where each may correspond to a column name or with arithmetic or string expressions over column names and constants following the SQL syntax. In the FROM clause, there is a from-list that consists of a list of tables. We assume that the WHERE clause contains n predicates, $A = \{A_1, A_2, ..., A_n\}$, connected by $n-1$ "AND"s. Each predicate A_i is of the form *expression op expression*, where *op* is a comparison operator (=, <>, >, >=, <, <=) or a membership operator (IN, NOT IN) and *expression* is a column name, a constant or an (arithmetic or string) expression. Note that here we assume that the WHERE clause contains only conjunctions using the logical connective "AND". We discuss how to process complex SQL queries in Section 4.3.3. Some predicate expressions in the WHERE clause of Q may involve comparisons with program variables. From the corresponding abstract query Q_{abs}, we check whether each predicate A_i contains any program variables from V.

We take the path P3 (Line 04 `true`, Line 14 `true`, and Line 18 `false`) in our preceding example shown in Figure 2 to illustrate the idea. The program input set is $I = \{$`type, zip`$\}$ and the path condition PC is

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

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

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

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

Algorithm 3 shows our procedure to construct the auxiliary query $\tilde{Q}$ based on the executed query (Q's concrete string and its abstract form Q_{abs}) and the intermediate information collected by DSE. Lines 5-21 present how to construct the clauses (SELECT, FROM, and WHERE) of the auxiliary query $\tilde{Q}$. We decompose Q_{abs} using a SQL parser[13] and get its n predicates $A = \{A_1, A_2, ..., A_n\}$ from the WHERE clause. We construct an empty predicate set $\tilde{A}$. For each predicate $A_i \in A$, we check whether A_i contains program variables. If not, we leave A_i unchanged and check the next predicate. If yes, we then check whether any contained program variable comes from the set V. If no program variables in the predicate are from V, we

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

substitute them with their corresponding concrete values in Q. In our example, the predicate `M.year=:years` belongs to this category. We retrieve the concrete value of `years` from Q and the predicate expression is changed as `M.year=15`. If some program variables contained in the predicate come from V, we substitute them with their symbolic expressions (expressed by the program inputs in I), substitute all the other program variables that are not from V with their corresponding concrete values in Q and copy the predicate A_i to $\tilde{A}$. The predicate `C.zipcode=:fzip` in our example belongs to this category. We replace `fzip` with `zip+1` and the new predicate becomes `C.zipcode=:zip+1`. We also add A_i's associated database attributes into a temporary attribute set C_V. Those attributes will be included in the SELECT clause of the auxiliary query $\tilde{Q}$. For the predicate `C.zipcode=:fzip`, the attribute `C.zipcode` is added to $\tilde{A}$ and is also added in the SELECT clause of the auxiliary query $\tilde{Q}$.

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

```
SELECT  C.zipcode
FROM    customer C, mortgage M
```

```
WHERE   M.year=15 AND C.SSN=M.SSN
```

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

Program variables in branch condition $pc_i \in PC$ after executing the query may be data-dependent on returned record values. In our example, the value of program variable `diff` in branch condition "`diff > 100000`" is derived from the values of the two variables `income`, `balance` that correspond to the values of attributes `C.income`, `M.balance` of returned records. Lines 22-32 in Algorithm 3 show how to incorporate later branch conditions in constructing the WHERE clause of the auxiliary query.

Formally, we get the set of program variables $U = \{U_1, U_2, ..., U_w\}$ that directly retrieve the values from the query's returned result set, and treat them as symbolic inputs. For each program variable U_i, we also keep its corresponding database at-

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

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

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

Program Input Generation: Note that executing the auxiliary query $\tilde{Q}$ against the database returns a set of values R_V for attributes in C_V. Each attribute in C_V can be traced back to some program variable in $V = \{V_1, V_2, ..., V_t\}$. Recall that V contains program variables that are data-dependent on program inputs I. Our final goal is to derive the values for program inputs I. Recall in Algorithm 3, we already collected in the predicate set $\tilde{A}$ the symbolic expressions of C_V with respect to program inputs

I. After substituting the attributes C_V with their corresponding concrete values in R_V resulted from executing $\tilde{Q}$ against the given database, we have new predicates in $\tilde{A}$ for program inputs I. We then feed these new predicates in $\tilde{A}$ to a constraint solver to derive the values for program inputs I. We give our pseudo procedure in Algorithm 4.

In our illustrative example, after executing our auxiliary query on Table 6, we get a returned value "28223" for the attribute `C.zipcode`. In $\tilde{A}$, we have `C.zipcode=:zip+1`. After substituting `C.zipcode` in `C.zipcode=:zip+1` with the value "28223", we have `28223=:zip+1`. The value "28222" for the program input `zip` can then be derived by invoking a constraint solver.

In our prototype, we use the constraint solver Z3[14] integrated in Pex. Z3 is a high-performance theorem prover being developed at Microsoft Research. The constraint solver Z3 supports linear real and integer arithmetic, fixed-size bit-vectors, extensional arrays, uninterpreted functions, and quantifiers. In practice, the result R could be a set of values. For example, the execution of the auxiliary query returns a set of satisfying zip code values. If multiple program input values are needed, we can repeat the same constraint solving process to produce each returned value in R.

4.3.2 Dealing with Aggregate Calculation

Up to now, we have investigated how to generate program inputs through auxiliary query construction. Our algorithm exploits the relationships among program inputs,

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

program variables, executed queries, and path conditions in source code. Database applications often deal with more than one returned record. In many database applications, multiple records are iterated from the query's returned result set. Program variables that retrieve values from the returned result set further take part in aggregate calculations. The aggregate values then are used in the path condition. In this section, we discuss how to capture the desirable aggregate constraints on the result set returned for one or more specific queries issued from a database application. These constraints play a key role in testing database applications but previous work [13, 22] on generating database states has often not taken them into account.

Consider the following code after the query's returned result set has been iterated in our preceding example shown in Figure 2:

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

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

formulation. Our idea is to extend the auxiliary query with the GROUP BY and HAVING clauses. For example, we learn that the variable `totalBalance` is a summation of all the values from the attribute `M.balance`. The variable `totalBalance` can be transformed into an aggregation function `sum(M.balance)`. We include `C.zipcode` in the GROUP By clause and `sum(M.balance)` in the HAVING clause of the extended auxiliary query:

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

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

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

Here we can use a special DSE technique [25] for dealing with input-dependent loops. With this technique, we can learn that the subpath with the conditions in Lines 14 and 18 being `true` has to be invoked at least three times in order to cover the branch condition `count >= 3` in Line 20c. Hence we need to have at least three records iterated into Line 18 so that true branches of Lines 14, 18, and 20c can be covered. In our auxiliary query, we can simply add `COUNT(*) >= 3` in the HAVING clause to capture this cardinality constraint.

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

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

4.3.3 Dealing with Complex Queries

SQL queries embedded in application program code could be very complex. For example, they may involve nested subqueries with aggregation functions, union, distinct, and group-by views, etc. The fundamental structure of a SQL query is a query block, which consists of SELECT, FROM, WHERE, GROUP BY, and HAVING clauses. If a predicate or some predicates in the WHERE or HAVING clause are of the form

[C_k op Q] where Q is also a query block, the query is a *nested query*. A large body of work exists on query transformation in databases. Various decorrelation techniques (e.g., [34]) have been explored to unnest complex queries into equivalent single level canonical queries and recent work [2] showed that almost all types of subqueries can be unnested.

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

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

and CPNF with the WHERE clause consisting of a conjunction of disjunctions (such as `(A11 OR... OR A1n) AND ... AND (Am1 OR... OR Amn)`). Note that DPNF and CPNF can be transformed mutually using DeMorgan's rules[15].

Next we present our algorithm on how to formulate auxiliary queries and determine program input values given a general DPNF query. Our previous Algorithm 3 deals with only a special case of DPNF where the query's WHERE clause contains only one `A11 AND ... AND A1n`. We show the algorithm details in Algorithm 5. Our idea is to decompose the DPNF query Q_{dpnf} into m simple queries Q_i $(i = 1, \cdots, m)$. The WHERE clause of each Q_i contains only one disjunction in the canonical form, `Ai1 AND ... AND Ain`. We apply Algorithm 3 to generate its corresponding auxiliary query $\tilde{Q}_i$ and apply Algorithm 4 to generate program input values R_i. The union of

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

Table 7: Symbolic query processing: $t1$ for path P5 and $t2$ for path P6.

(a) Symbolic customer

	SSN	zipcode	income	...
$t1$:	$a1	$b1	$c1	...
$t2$:	$a2	$b2	$c2	...

(b) Symbolic customer after SQP

	SSN	zipcode	income	...
$t1$:	$a1	$b1=:zip+1	$c1 - 1.5*$f1<=100000	...
$t2$:	$a2	$b2=:zip+1	$c2 - 1.5*$f2>100000	...

(c) Symbolic mortgage

	SSN	year	balance	...
$t1$:	$d1	$e1	$f1	...
$t2$:	$d2	$e2	$f2	...

(d) Symbolic mortgage after SQP

	SSN	year	balance	...
$t1$:	$d1=$a1	$e1=:years	$c1 - 1.5*$f1<=100000	...
$t2$:	$d2=$a2	$e2=:years	$c2 - 1.5*$f2>100000	...

(e) Instantiated customer

	SSN	zipcode	income	...
$t1$:	003	28223	50000	...
$t2$:	004	28223	150000	...

(f) Instantiated mortgage

	SSN	year	balance	...
$t1$:	003	30	10000	...
$t2$:	004	30	20000	...

R_is then contains all appropriate program input values.

4.3.4 Modifying Database State

Although constructing auxiliary queries provides a way of deriving effective program inputs based on the existing database state, executing the constructed auxiliary queries may still return an empty result set, which indicates that the current database state is lack of qualified records. To deal with such situation, we need to generate new records and populate them back to the database.

For the preceding example code in Figure 2, choosing Line 06 to be `true` will make true the condition `M.year = 30` for the WHERE clause from the query in Line 11, where we observe that the database in Table 6 does not contain sufficient records. Thus, to cover two new paths P5 and path P6 as shown below, we need to have at least one record that satisfies constraints on the query result set.

P5:(type != 0) && (results.Read() == true) &&(diff <= 100000)

P6:(type != 0) && (results.Read() == true) &&(diff > 100000)

In our approach, we generate new records by invoking the data instantiation component. We build symbolic databases consisting of symbolic tuples. We use *symbolic query processing* by parsing the SQL statement and substitute the symbolic tuples in the symbolic databases with symbols that reflect the constraints from the SQL statement.

Tables 7(a) and 7(c) show the symbolic tables of `customer` and `mortgage`, respectively. For example, tuple $t1$ in Figure 7(a) is a symbolic tuple of symbolic relation `customer` to cover path P5; symbol $a1 represents any value in the domain of attribute `SSN` and symbol $c1 represents any value in the domain of attribute `income`. Similarly, tuple $t2$ is a symbolic tuple to cover path P6. After symbolic query processing, the symbolic tables shown in Tables 7(b) and 7(d) have captured all the constraint requirements specified in the symbolic case to cover paths P5 and P6 but without concrete data. We can see that $t1$ involves the constraint $\$c1 - 1.5 * \$f1 \leq 100000$ while $t2$ involves the constraint $\$c2 - 1.5 * \$f2 > 100000$.

In our approach, we also collect basic constraints at the database schema level (e.g., not-NULL, uniqueness, referential integrity constraints, domain constraints, and semantic constraints). For example, attribute `age` in table `customer` must be in the range (0, 100) and attribute `balance` must be greater than 1000. The symbolic tables together with the basic constraints are then sent to a constraint solver,

which can instantiate the symbolic tuples with concrete values. Tables 7(e) and 7(f) show the instantiated records from a constraint solver. For example, the record $t1$ is instantiated as "`C.SSN=003, C.zipcode=28223, C.income=50000`" in table `customer` and "`M.SSN=003, M.year=30, M.balance=10000`" in table `mortgage`. From constraints such as "`$b1=:zip+1`", the constraint solver also returns the value `28222` for input parameter `zip`. Hence, a test with program input {`type=1, zip=28222`} on the newly populated database state will lead to coverage of path P5.

To cover the preceding paths P5 and P6 for the example code in Figure 2, generating one new record for each path could be enough. However, in practice, satisfying cardinality constraints usually requires a large number of records. Cardinality constraints significantly affect the total cost of generating new database records. Consider the example related with cardinality constraints as discussed in Section 4.3.2. Suppose that we have another piece of code appended to Line 20 as

```
20c:  if (count >= 1000)
20d:    computeSomething();
```

Note that here we have a more extensive cardinality constraint `count >= 1000` for the qualified records than the previous cardinality constraint (i.e., `count >= 3`). As discussed in Section 4.3.2, we can first construct an auxiliary query as below to choose values for program input `zip`.

```
SELECT   C.zipcode
FROM     customer C, mortgage M
WHERE    M.year = 15 AND C.SSN = M.SSN AND C.income - 1.5 * M.balance > 100000
GROUP BY C.zipcode
HAVING   COUNT(*) >= 1000
```

However, executing this auxiliary query may return an empty result, indicating that the current existing database does not contain sufficient qualified records. Naively, we can generate at least 1000 new records from scratch and populate them back to the database, requiring a huge cost. In practice, an empty result returned by the auxiliary query could be caused by the insufficient size of existing qualified records. For example, if the existing database has already contained 900 such records and in that case, we need to generate only 100 other new records, requiring much lower cost than generating all the 1000 records from scratch.

Based on the required cardinality constraints, to reduce the cost of generating new records from scratch, we conduct database record generation in the following way. First, we check whether we need to generate new records if the constructed auxiliary query returns an empty result. If no, our technique does not need to help with such situation; if yes, second, we check whether the constructed auxiliary query contains cardinality constraints. If so, we remove the required cardinality constraints and get another auxiliary query by selecting the size of current qualified records. Third, we run this modified auxiliary query on the existing database and get the value for the size of current qualified records. Fourth, we compare the size with the required cardinality constraints. We can derive how many more records are needed and then conduct the data generation.

For example, when we observe that running the auxiliary query returns an empty result, we remove the cardinality constraints `HAVING COUNT(*) >= 1000`, add `COUNT(*)`

to the SELECT clause, and get a new auxiliary query as

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

Running this auxiliary query will return a value 900 for `COUNT(*)` if the existing database has already contained 900 qualified records within a specific zipcode. Comparing with the required cardinality constraints `COUNT(*) >= 1000`, we detect that at least 100 other records are needed. Hence, we generate new records by invoking the data instantiation component. In this way, we can significantly reduce the cost when the current existing database has already contained a large number of qualified records and only a few more new records are needed.

4.3.5 Dealing with Multiple Queries

In practice, a database application may contain multiple queries to interact with the existing database where the problem contexts become more complex. In this section, we discuss about two typical cases of multiple queries and investigate the relationship between these queries and program inputs. Figure 3 shows the brief logical structures of the two cases. In the first case, the application contains two queries and one program input, where the conditions from the first query are data-dependent on the program input and the conditions from the second query are data-dependent on the first query's result-manipulation code. In the second case, the application contains two queries and two program inputs, where the conditions from the first

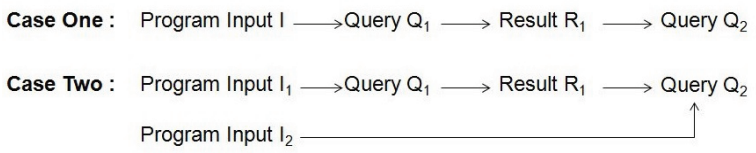

Figure 3: Two typical cases of database applications including two queries

```
01:public int calcStat(int zip) {
02:  int totalIncome = 0, count1 = 0, count2 = 0;
03:  SqlConnection sc = new SqlConnection();
04:  sc.ConnectionString = "..";
05:  sc.Open();
06:  string query1 =
     "SELECT C.SSN, C.income FROM customer C "
     "WHERE C.gender = 'male' AND C.zipcode='" + zip + "'";
07:  SqlCommand cmd1 = new SqlCommand(query1, sc);
08:  SqlDataReader results1 = cmd1.ExecuteReader();
09:  while (results1.Read()){
10:     totalIncome += int.Parse(results["income"]);
11:     count1++;}
12:  int average = totalIncome/count1;
13:  string query2 =
     "SELECT * FROM mortgage M WHERE M.year = 15 AND '"
     + average + "' - 1.5 * M.balance > 100000";
14:  SqlCommand cmd2 = new SqlCommand(query2, sc);
15:  SqlDataReader results2 = cmd2.ExecuteReader();
16:  while (results2.Read()){
       ...
17:     count2++;}
18:  return count2;}
```

Figure 4: An example code snippet involving two queries and one program input

query are data-dependent on one program input and the conditions from the second query are data-dependent on both the first query's result-manipulation code and the other program input. For other minor cases, we introduce some examples and briefly present the solutions.

Case One: We start from the basic case where the application contains two SQL queries and one input. Consider the example code in Figure 4. The program has one input `zip` and it is combined into the first query `query1`. `query1` selects customers' in-

come values within a given zipcode from the `customer` table. Then, `query1` is executed and the returned records are iterated where the sum of each record's `C.income` value is calculated. An `average` variable is calculated as the average income within the input zipcode. The `average` value is then combined into another query `query2`'s WHERE clause, where `query2` selects qualified records from the `mortgage` table. After `query2` is executed, the returned records are iterated for some further calculations. The program finally returns the calculated results. Note that in this example, the program input (i.e., `zip`) affects the first query's execution while the result manipulation (i.e., `average`) affects the later query's execution.

The challenge exists in that the relationship between `query2` and `query1`'s result-manipulation cannot be directly or naturally captured by DSE unless `query1`'s result-manipulation code has been fully explored. To generate effective `zip` values so that both `query1` and `query2` can return non-empty results, we first apply our preceding auxiliary-query-construction technique discussed in Section 4.3.1 on `query1` and construct an auxiliary query as

```
SELECT   C.zipcode
FROM     customer C
WHERE    C.gender = 'male'
```

Executing this auxiliary query can help derive appropriate values for `zip` so that DSE can reach the first query's result-manipulation part. Then, DSE can capture the data-dependency between conditions from the later query (i.e., `query2`) and the program variables derived from the first query's result-manipulation part (i.e., `average`).

We observe that `average` performs as a "local" external input to `query2`. At this point, we again apply our preceding technique in Section 4.3.1 on `query2` and construct another auxiliary query as

```
SELECT   M.balance
FROM     mortgage M
WHERE    M.year = 15
```

Running this auxiliary query will return values for the attribute `M.balance`. From the constraint `average - 1.5 * M.balance > 100000` in `query2`, we replace `M.balance` with these returned values. Thus, we are able to derive appropriate and fixed values (e.g., a value set Val) for the variable `average`. Then, based on the previously captured relationship between `average` and its corresponding database attribute (i.e., `C.income`) by DSE, we modify the auxiliary query for `query1` as

```
SELECT   C.zipcode, AVG(C.income)
FROM     customer C
WHERE    C.gender = 'male'
GROUP BY C.zipcode
HAVING   AVG(C.income) IN Val
```

Running this modified auxiliary query on the existing database will return effective values for the program input `zip`. Using the returned values as input can guarantee that both `query1` and `query2` return non-empty results. To formally summarize the preceding technique, we derive program inputs as follows.

Assume we have a database application with an input I and the program contains two SQL queries Q_1 and Q_2, where conditions in Q_1 are data-dependent on I

and conditions in Q_2 are data-dependent on the variables derived from Q_1's result-manipulation code. To generate effective program inputs so that executions of both Q_1 and Q_2 can return non-empty results, we first construct an auxiliary query $\tilde{Q}_1$ for Q_1. We run $\tilde{Q}_1$ on the existing database and get concrete values for I. We pick an arbitrary value for I and let DSE reach Q_1's result-manipulation code. DSE is also able to capture the variable v involved in Q_2 that is data-dependent on preceding variables derived from Q_1's result-manipulation code. Then, for Q_2, we treat v as a "local" external input and construct another auxiliary query $\tilde{Q}_2$ for Q_2. Running $\tilde{Q}_2$ on the existing database and using the condition from Q_2 that contains v , we get a value set Val for v. We then modify $\tilde{Q}_1$ by adding an extra constraint. The extra constraint is derived from the information collected by DSE, using the value set Val for v and the captured relationship between variables derived from Q_1's result-manipulation code and these variables' corresponding database attributes. For the example in Figure 4, v in `query2` is derived as `average` and we get a value set Val for `average` using the constructed auxiliary query for `query2`. We also derive that `average` is related with the database attribute `C.income`. We then convert the preceding data-dependency `average = totalIncome/count1` in Line 12 to be `AVG(C.income) in` Val and add this extra constraint into the auxiliary query $\tilde{Q}_1$ for `query1`. The modified $\tilde{Q}_1$ can help derive effective values for I.

Case Two: Based on the example in Figure 4, consider another typical case in

```
01:public int calcStat(int zip, int inputDiff) {
02:  int totalIncome = 0, count1 = 0, count2 = 0;
03:  SqlConnection sc = new SqlConnection();
04:  sc.ConnectionString = "..";
05:  sc.Open();
06:  string query1 =
     "SELECT C.SSN, C.income FROM customer C "
     "WHERE C.gender = 'male' AND C.zipcode='" + zip + "'";
07:  SqlCommand cmd1 = new SqlCommand(query1, sc);
08:  SqlDataReader results1 = cmd1.ExecuteReader();
09:  while (results1.Read()){
10:     totalIncome += int.Parse(results["income"]);
11:     count1++;}
12:  int average = totalIncome/count1;
13:  string query2 =
     "SELECT * FROM mortgage M WHERE M.year = 15 AND '"
     + average + "' - 1.5 * M.balance >'" + inputDiff + "'";
14:  SqlCommand cmd2 = new SqlCommand(query2, sc);
15:  SqlDataReader results2 = cmd2.ExecuteReader();
16:  while (results2.Read()){
       ...
17:     count2++;}
18:  return count2;}
```

Figure 5: Another example code snippet involving two queries and two program inputs

Figure 5. In this example, the program contains two inputs where one input affects the first query while both the second input and the first query's result-manipulation affect the second query. To deal this category of multiple queries, we modify the preceding technique as follows.

Assume that we have a database application with two inputs I_1 and I_2 and the program contains two SQL queries Q_1 and Q_2, where conditions in Q_1 are data-dependent on I_1 and conditions in Q_2 are data-dependent on both I_2 and the variables derived from Q_1's result-manipulation code. To generate effective program inputs so that executions of both Q_1 and Q_2 can return non-empty results, we first construct an auxiliary query $\tilde{Q}_1$ for Q_1. We run $\tilde{Q}_1$ on the existing database and get concrete values for I_1. Using these values, we let DSE reach Q_1's result-manipulation code.

DSE is also able to capture the variable v involved in Q_2 that is data-dependent on preceding variables derived from Q_1's result-manipulation code. For Q_2, DSE could set concrete values for v using preceding values derived from Q_1's result-manipulation code. Then, for I_2 that appears in Q_2, we treat I_2 as a "local" external input and construct another auxiliary query $\tilde{Q}_2$ for Q_2. We choose I_2's related database attribute C_{I_2} as the selection in $\tilde{Q}_2$. Running $\tilde{Q}_2$ on the existing database, we get values Val for C_{I_2}. Then, using the condition from Q_2 that contains I_2, we replace C_{I_2} with Val and derive the final values for I_2. For the example in Figure 5, we construct auxiliary query for `query1` as

```
SELECT   C.zipcode
FROM     customer C
WHERE    C.gender = 'male'
```

We can derive values for the input `zip`. Then, DSE can reach `query1`'s result-manipulation code and the variable `average` is later set with concrete values. We treat the input `inputDiff` as a "local" external input for `query2` and we derive that `inputDiff` is related with the database attribute `M.balance`. We construct another auxiliary query for `query2` as

```
SELECT   M.balance
FROM     mortgage M
WHERE    M.year = 15
```

We can derive values for the attribute `M.balance`. Then, we replace both `average` and `M.balance` in the condition `average - 1.5 * M.balance > inputDiff` with those

derived values. We send the modified condition to a constraint solver and get final values for `inputDiff`.

Other Minor Cases: The previously discussed techniques for dealing with the preceding two typical cases containing two queries could be extended to dealing with more than two queries by solving each individual query iteratively. In practice, we may have other minor categories that involve multiple queries. For example, assume that the program contains only one input and two queries, where the first query is static and the second query contains conditions that are data-dependent on the input. In that case, since the first query is static, DSE could directly reach the second query. We can straightforwardly apply our auxiliary-query-construction technique on the second query and derive values for the program input. For another example, assume that the program contains two input and two queries, where each query is only data-dependent on one input and the conditions in an individual query is not data-dependent on one another. In that case, since there is no data-dependency among each individual query, we can apply the auxiliary-query-construction technique separately on each individual query and derive values for each program input.

4.4 Evaluation

Our approach can provide assistance to DSE-based test-generation tools (e.g., Pex [1] for .NET) to improve code coverage in database application testing. In our evaluation, we seek to evaluate the benefit and cost of our approach from the following

two perspectives:

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

RQ2: What is the cost of our approach's assistance?

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

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

and then record new statistics.

We conduct an empirical evaluation on two open source database applications `RiskIt` and `UnixUsage`. The introductions to these two applications are previously presented in Chapter 1.

4.4.1 Code Coverage

We show the evaluation results in Table 8 and Table 9. For each table, the first part (Columns 1-2) shows the index and method names. The second part (Columns 3-6) shows the code coverage result. Column 3 "total(blocks)" shows the total number of blocks in each method. Columns 4-6 "covered(blocks)" show the number of covered blocks by Pex without our approach's assistance, the number of covered blocks by Pex together with our approach's assistance, and the percentage increase, respectively.

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

The `UnixUsage` application contains 28 methods whose program inputs are related

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

4.4.2 Cost

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

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

branches not covered by Pex with relatively low cost.

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

Algorithm 3 Auxiliary Query Construction

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

Find variables $V = \{V_1, V_2, ..., V_t\}$ data-dependent on I;
Decompose Q_{abs} with a SQL parser for each clause;
Construct a predicate set $A = \{A_1, A_2, ..., A_n\}$ from Q's WHERE clause;
Construct an empty predicate set $\tilde{A}$, an empty attribute set C_V, and an empty query $\tilde{Q}$;
for each predicate $A_i \in A$ **do**
  if A_i does not contain program variables **then**
    Leave A_i unmodified and check the next predicate;
  else
    if A_i does not contain program variables from V **then**
      Substitute A_i's program variables with their corresponding concrete values in Q;
    else
      Substitute the variables from V with the expression expressed by I;
      Substitute the variables not from V with their corresponding concrete values in Q;
      Copy A_i to $\tilde{A}$;
      Add A_i's associated database attributes to C_V;
    end if
  end if
end for
Append C_V to $\tilde{Q}$'s SELECT clause;
Copy Q's FROM clause to $\tilde{Q}$'s FROM clause;
Append $A - \tilde{A}$ to $\tilde{Q}$'s WHERE clause;
Find variables $U = \{U_1, U_2, ..., U_u\}$ coming directly from Q's result set;
Find U's corresponding database attributes $C_U = \{C_{U1}, C_{U2}, ..., C_{Uw}\}$;
for each branch condition $pc_i \in PC$ after Q's execution **do**
  if pc_i contains variables data-dependent on U **then**
    Substitute the variables in pc_i with the expression expressed by the variables from U;
    Substitute the variables from U in pc_i with U's corresponding database attributes in C_U;
    Add the branch condition in pc_i to $\tilde{PC}$;
  end if
end for
Flip the last branch condition in $\tilde{PC}$;
Append all the branch conditions in $\tilde{PC}$ to $\tilde{Q}$'s WHERE clause;
return $\tilde{Q}$;

Algorithm 4 Program Input Generation

Input: an auxiliary query $\tilde{Q}$, program inputs I
intermediate results C_V and $\tilde{A}$ from Algorithm 3
Output: program input values R for I

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

Algorithm 5 Program Input Generation for DPNF Query

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

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

Table 8: Evaluation results on RiskIt

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

Table 9: Evaluation results on UnixUsage

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

4.5 Conclusions

In this research, we have presented an approach that takes database applications and a given database as input, and generates appropriate program input values to achieve high code coverage. In our approach, we employ dynamic symbolic execution to analyze the code under test and formulate auxiliary queries based on extracted constraints to generate program input values. We incorporate data instantiation component in our framework to deal with the case that no effective program input values can be attained. We determine how to generate new records and populate them in the new database state. We also extend our program-input-generation approach to database applications including multiple queries. Empirical evaluations on two open source database applications showed that our approach can assist Pex, a state-of-the-art DSE tool, to generate program inputs that achieve higher code coverage than the program inputs generated by Pex without our approach's assistance. Part of this research was published in ASE 2011 [44]. The extended work was submitted to a journal [47] and is under review when preparing this dissertation.

In future work, we plan to extend our technique to construct auxiliary queries directly from embedded complex queries (e.g., nested queries), rather than from their transformed norm forms. An execution path of an application can involve the execution of a sequence of SQL statements including both SELECT queries and state-modifying SQL statements such as INSERT, UPDATE and DELETE. We aim to explore how to extend our approach to state-modifying SQL statements.

CHAPTER 5: GENERATE TEST BY SYNTHESIZED INTERACTIONS

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, 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.

5.1 Illustrative Example

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

```
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;}
```

Figure 6: A code snippet from a database application in C#

The code snippet in Figure 6 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 10. 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 vari-

Table 10: 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	∈ {F, M}	year	Int	∈ {10, 15, 30}
zipcode	Int	[00001, 99999]			
age	Int	(0, 100]	balance	Int	[2000, Max)
income	Int	[100000, Max)			

ables: a local variable `zip` and a program-input argument `inputYear`. The returned result records are then iterated (Lines 09-15). For each record, a variable `diff` is calculated from the values of the fields `C.income`, `M.balance`, and `M.year`. If `diff` is greater than `100000`, a counter variable `count` is increased (Line 15). The method then returns the final result (Line 16). To achieve high structural coverage of this program, we need appropriate combinations of database states and program inputs.

Typically, a database application communicates with the associated database through four steps. First, the application sets up a connection with the database (e.g., construct a `SqlConnection` object). Second, it constructs a query to be executed and combines the query into the connection (e.g., construct a `SqlCommand` object using the database connection and the string value of the query). Third, if the query's execution yields an output, the result is returned (e.g., construct a `SqlDataReader` object by calling the API method `ExecuteReader()`[16]). Fourth, the returned query result is manipulated for further execution.

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

```
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;}
```

Figure 7: Transformed code produced by SynDB for the code in Figure 6

To test the preceding code, for existing DSE-based test generation approaches [38, 53] where the application interacts with either real database or mock database, in the first run, DSE chooses random or default values for `inputYear` (e.g., `inputYear = 0` or `inputYear = 1`). For generation of a database state, the constraint for the attribute `M.year` in the concrete query becomes `M.year = 0` or `M.year = 1`. Here, the query-construction constraints are simply *true*. However, we observe from the schema in Table 10 that the randomly chosen values (e.g., `inputYear = 0 or 1`) violate a database schema constraint: `M.year` can be chosen from only the set {`10, 15, 30`}. Query constraints (constraints derived from the WHERE clause of the concrete query) thus conflict with the database schema constraints. As aforementioned, the violation of database schema constraints would cause the generation of invalid database states.

```
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*/}}
```

Figure 8: Synthesized database state

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., the condition in Line 14) from sub-paths for manipulating the query result. 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.

To address the preceding problems in testing database applications, our approach replaces the original database interactions by constructing synthesized database interactions. For example, we transform the example code in Figure 6 into another form shown in Figure 7. 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 10, its synthesized database state is shown in Figure 8. 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.

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

Table 11: 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(SynSqlConnection SSConn,string q) ->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`

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 12 (e.g., `inputYear = 15` and the record with `C.SSN = 001`). In the next run, the execution of the query whose WHERE clause has been updated as `C.SSN = M.SSN AND C.zipcode = 28223 AND M.year = 15` yields a record so that DSE's exploration enters the `while` loop (Lines 09-15). Straightforwardly, the transformed code can also guide DSE's exploration to collect later constraints (Line 14) from sub-paths for manipulating the query result to generate new inputs. For

example, to cover `Line 14 == true`, the collected new constraint `(income - 1.5 * balance) * year` is combined with previous constraints to generate new inputs (e.g., the input `inputYear = 15` and the record with `C.SSN = 002` as shown in Table 12).

Table 12: Generated program inputs and database states to cover the paths `Line09 = true, Line14 = false` and `Line09 = true, Line14 = true`

inputYear	dbState								
	dbState.Customer						dbState.Mortgage		
	SSN	name	gender	zipcode	age	income	SSN	year	balance
15	001	AAA	F	28223	45	150000	001	15	100000
15	002	BBB	M	28223	55	150000	002	15	50000

5.2 Problem Formalization and Proposed Solution

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

Considering the preceding four parts of constraints, applying DSE on testing database applications faces great challenges for generating both effective program inputs and sufficient database states. For existing DSE-based approaches of testing database applications, it is difficult to correlate program-execution constraints and environment constraints. Performing symbolic execution of database interaction API methods would face a significant problem: these API methods are often implemented in either native code or unmanaged code, and even when they are implemented in managed code, their implementations are of high complexity; existing DSE engines have difficulty in exploring these API methods. In practice, existing approaches [38, 53] 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 [38] or mock database [53]) so that it returns a non-empty query result for query-result-manipulation code to iterate through.

A problem of such design decision made in existing approaches is that values for variables involved in the query issued to the database system could be prematurely concretized. Such premature concretization could pose barriers for achieving structural coverage because query constraints (constraints from the conditions in the WHERE clause of the prematurely concretized query) may conflict with later constraints, such as database schema constraints and query-result-manipulation con-

straints. In particular, the violation of database schema constraints could cause the generation of invalid database states, thus causing low code coverage of database application code in general. On the other hand, the violation of query-result-manipulation constraints could cause low code coverage of query-result manipulation code. Basically, there exists a gap between program-execution constraints and environment constraints, caused by the complex black-box query-execution engine. Treating the connected database (either real or mock) as an external component isolates the query constraints with later constraints such as database schema constraints and query-result-manipulation constraints.

In our research, we develop a DSE-based test generation approach called *SynDB* to deal with the preceding challenges. 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. Through this way, we correlate aforementioned four parts of

constraints within a database application, and bridge the gap of program-execution constraints and environment constraints. Then, based on the transformed code, we guide DSE's exploration through the 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. In our approach, we use a state-of-the-art tool called Pex [1] for .NET from Microsoft Research as the DSE engine and also use it to conduct the test generation. 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.

5.3 Approach

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

database states. In this way, we generate sufficient database states as well as effective program inputs.

5.3.1 Code Transformation

For the code transformation, we transform the code under test into another form upon which our synthesized database interactions can execute. Basically, we replace the standard database interactions with renamed API methods. We mainly deal with the statements or stored procedures to execute against a SQL Server database [40]. We identify relevant method calls including the standard database API methods. We replace the original database API methods with new names (e.g., we add "Syn" before each method name). Note that replacing the original database API methods is a large body of work. Even a single class could contain many methods and their relationships could be very complex. In our *SynDB* framework, we mainly focus on the classes and methods that are commonly used and can achieve the basic functionalities of database applications. Table 11 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 10, the corresponding synthesized database state is shown in Figure 8. Meanwhile, we check database schema constraints for each table and each attribute by transforming the database schema constraints into normal program code for checking these constraints. Note that we are also able to capture complex constraints at the schema level such as constraints

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

Figure 9: Synthesized `SqlConnection`

across multiple tables and multiple attributes. We then add the synthesized database state as an input to the transformed code. Through this way, we force DSE to track the associated database state symbolically and guide DSE's exploration to collect constraints of the database state.

5.3.2 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 11, 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

```
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 6
          return synReader;}
   public SynSqlDataReader ExecuteNonQuery(){ //execute the modify operation;
          DatabaseState synDB = this.getDB();
          ModifyExe(synDB,this.getQuery());} //details in Algorithm 7
   public DatabaseState getDB() {return dbStateComm;} //new method
   public string getQuery() {return query;} //new method}
```

Figure 10: Synthesized `SqlCommand`

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 9 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 10 gives the details of `SynSqlCommand` whose methods `ExecuteReader()` and `ExecuteNonQuery()` are used to execute queries. The details of algorithms for `ExecuteReader()` and `ExecuteNonQuery()` are discussed later in Section 5.3.3 (Algorithms 6 and 7, 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` [39] 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.

5.3.3 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).

Select Operation

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

Figure 11: A simple query

We first discuss how to deal with the SELECT statement for a simple query. A simple query (shown in Figure 11) consists of three parts. In the FROM clause, there is a from-list that consists of a list of tables. In the SELECT clause, there is a

list of column names of tables named in the FROM clause. In the WHERE clause, there is a qualification that is a boolean combination of conditions connected by logical connectives (e.g., AND, OR, and NOT). A condition is of the form *expression op expression*, where *op* is a comparison operator (=, <>, >, >=, <, <=) or a membership operator (IN, NOT IN) and *expression* is a column name, a constant, or an (arithmetic or string) expression. We leave discussion for complex queries in Section 5.3.4.

In our approach, we rewrite the method `ExecuteReader()` (shown in Figure 10) 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 6. 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 aforementioned, we deal with the SELECT statement for a simple query whose WHERE clause contains a qualification that is a boolean combination of conditions connected by logical connectives (e.g., AND, OR, and NOT). In this step, we transform the evaluation of the conditions specified in the WHERE clause into normal program code in the following way. From the WHERE clause, we replace the database attributes in the conditions with their corresponding column names in `DataTable resultSet`. We also map those SQL logical connectives (e.g., AND, OR, and NOT) to program logical operators (e.g., &&, ||, and !), thus keeping the original logical relations unchanged. After these transformations, we push the transformed logical conditions into parts of a path condition (e.g., realized as an assumption recognized by the DSE engine). Third, after scanning all the rows, we remove unnecessary columns from `DataTable resultSet` based on the SELECT clause (Lines 23-28). For each column c in `DataTable resultSet`, if it appears in the SELECT clause, we keep this column; otherwise, we remove c. After the preceding three steps, the field `DataTable resultSet` contains all rows with qualified values that the SELECT statement should return.

Through this way, we construct a `SynSqlDataReader` object to relate the previous

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

Modify Operation

To deal with queries that modify database states, we rewrite the API method called `ExecuteNonQuery()` (shown in Figure 10). The pseudocode is shown in Algorithm 7. The method also operates on the field `dbStateComm` that represents the symbolic database state. We first check the modification type of the query (e.g., INSERT, UPDATE, and DELETE). For the INSERT statement (Lines 1-8), from the table in the INSERT INTO clause, we find the corresponding table in `dbStateComm`. From the VALUES clause, we then check whether the values of the new row to be inserted satisfy database schema constraints. We also check after this insertion, whether the whole database state still satisfy database schema constraints. If both yes, we add this new row to the target table in `dbStateComm`, by mapping the attributes from the INSERT query to their corresponding fields. For the UPDATE statement (Lines 9-19), from the UPDATE clause, we find the corresponding table in `dbStateComm`. We scan the table with the conditions from the WHERE clause and locate target rows. For each row, we also check whether the specified values satisfy the schema

constraints. If qualified, we set the new values to their corresponding columns based on the SET clause. For the DELETE statement (Lines 20-30), from the DELETE FROM clause, we find the corresponding table in `dbStateComm`. We locate the target rows using conditions from the WHERE clause. We then check whether this deletion would violate the schema constraints; otherwise, we remove these rows.

5.3.4 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.

Dealing with Complex Queries

Note that SQL queries embedded in the program code could be very complex. For example, they may involve nested sub-queries with aggregation functions, union, distinct, and group-by views, etc. The syntax of SQL queries is defined in the ISO standardization[17]. The fundamental structure of a SQL query is a query block, which consists of SELECT, FROM, WHERE, GROUP BY, and HAVING clauses. If a predicate or some predicates in the WHERE or HAVING clause are of the form [C_k op Q] where Q is also a query block, the query is a *nested query*. A large body of work [34, 17, 2] on query transformation in databases has been explored to unnest complex queries into equivalent single level canonical queries. Researchers showed that almost all types of subqueries can be unnested except those that are correlated

[17] 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)
```

Figure 12: A canonical query in DPNF

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 12, 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[18]. 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.

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 object with

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

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

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

Dealing with Cardinality Constraints

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

Consider the example code in Figure 13. Inside the `while` loop after the result set is returned, a variable `count` is updated every time when a condition `balance > 50000` is satisfied. Then, outside the `while` loop, branch conditions in Lines 10a and 10c depend on the values of `count`. Manually, we can observe that the value of `count` depends on how many records satisfy the branch condition in Line 09b. We may generate enough database records so that branches in Lines 10a and 10c could be entered. However, since there is a `while` loop, applying DSE to hunt for enough database records (thus covering Lines 10a) faces significant challenges: the size of `results` can range to a very large number, of which perhaps only a small number of records can satisfy the condition in Line 09b. Hence, this problem is reduced to a traditional issue [68]: to explore a program that contains one or more branches with relational conditions (here, we have `(count > 10)`) where the operands are scalar values (integers or floating-point numbers) computed based on control-flow decisions connected to program inputs through data flow (here, we have `if (balance > 50000) count++;`).

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

Figure 13: An example where cardinality constraints come from the query result manipulation

In the literature, Xie et al. proposed an approach *Fitnex* [68] 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 [68], so that generating enough database records with high efficiency becomes feasible. For the example code in Figure 13, 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.

5.4 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 [38, 53] in testing database applications?

RQ2: What is the running cost of our approach compared with existing approaches on generating tests?

5.4.1 Subject Applications

We conduct an empirical evaluation on three open source database applications: `iTRUST`, `RiskIt`, and `UnixUsage`. The introductions to these applications are previously presented in Chapter 1.

For these applications, we focus on the methods whose SQL queries are constructed dynamically. For the `iTRUST` application, from the sub-package 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.

5.4.2 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 13. During our search for the subject applications to be evaluated, we found that most database applications do not contain very complex schema constraints. We observe that the predefined database schema constraints for `RiskIt` and `UnixUsage` are over-simplified, such as 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 14.

Algorithm 6 *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;

Algorithm 7 *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
```

Table 13: 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 14: 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]

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

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

We choose one method `filterZipcode` from the `RiskIt` application to illustrate the

evaluation process. The method accepts an input `zip` to form an query, searches corresponding records from the `userrecord` table, and conduct some calculation from the returned records. After the code transformation, we get a new method `SynfilterZipcode` as shown in Figure 14. 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 15 (Lines 01-21). Running this test covers the path where `Line 11 = true, and Line 16 = true` in Figure 14. For the test in Figure 15, 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 15, we construct a new test shown in Figure 16. 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 [38], we make use of our `SynDB` framework. Basically, we simulate the approach [38] by generating database records using constraints from a concrete

```
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;}
```

Figure 14: `SynfilterZipcode`:Transformed code of method `filterZipcode`

```
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)}
```

Figure 15: Tests generated by Pex on `SynfilterZipcode`

query. We seek to compare the performance with our approach from two perspectives. First, constraints from the WHERE clause of a concrete query may conflict with database schema constraints. In this step, we do not transform database schema constraints into normal program code. We generate database records and corresponding program input values. We insert these generated database records back to the real database, use the generated program input values to form test cases, and measure the code coverage. Note that the insertion of these generated database records may be rejected by the database due to conflicts with database schema constraints. By doing this, we see how this omission of transformation regarding database schema constraints would impact code coverage. Second, constraints from the WHERE clause of a concrete query may conflict with later query-result-manipulation constraints. In this step, at each query-issuing location, we get constraints from the WHERE clause of the concrete query. We generate records based on these constraints and insert these records to the real database. Then, DSE's exploration can enter the query-result iteration and we get additional constraints from the query-result-manipulation code. We generate new records and insert them back to the real database again and see how the addition of such new records would impact code coverage. For both perspectives, we still conduct code transformation on the original code under test and then run Pex to generate tests, populate the generated records back to the real database, and construct new tests for the original code under test. After running the new tests, we measure code coverage achieved by these existing approaches. Such simulated results

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

Figure 16: Constructed tests for `filterZipcode`

are expected to be equivalent to the results produced with the original implementations of the existing approach [38].

To compare our approach with another existing test-generation approach for database application testing [53], we apply the MODA tool on the subject applications. Since the initial version of MODA did not consider various kinds of database schema constraints during its preparation for the mock database, to fairly compare MODA with our approach, we assist MODA by incorporating the database schema constraints into the suitable location (i.e., in its MockDBMS.cs file). We then run MODA on each method under test to conduct test generation. We measure the code coverage achieved by MODA and meanwhile record its running time.

5.4.3 Results

We report the evaluation results in Tables 15, 16, and 17, from the perspectives of code coverage and cost. The evaluation is conducted on a machine with hardware configuration Intel Pentium 4 CPU 3.0 GHz, 2.0 GB Memory and OS Windows XP SP2.

Code Coverage

In Tables 15, 16, and 17, the first part (Columns 1-2) shows the index and method names. The second part (Columns 3-6) shows the code coverage result. Column 3 "total(blocks)" shows the total number of blocks in each method. Columns 4-6 "covered(blocks)" show the number of covered blocks using tests generated by our approach, the number of covered blocks using tests generated by existing approaches, and the percentage increase, respectively. We find that for existing approaches [38, 53], they achieve the same code coverage for all the methods. The reason is that they use the same design decision when conducting the generation for database records, interacting with either a real database [38] or a mock database [53]. 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 running time consumed 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 is that the generated records violate the database schema constraints. When populating such records back into the real database, the insertion operations are rejected by the database. Take the aforementioned example method `SynfilterZipcode` shown in Figure 14 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 14. 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 generate tests to cover more blocks for all the 14 methods under test. For example, for the No.7 method `getRole` shown in Table 15, the original code is shown in Figure 17. 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 14, where the value can be chosen from only a predefined string set. Existing approaches fail to generate effective test inputs to

```
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);}}
```

Figure 17: Method `getRole` from `iTRUST`

cover Line 12 while our approach has captured such 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 case is different from the method shown in Figure 14. For the method shown in Figure 14, 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 16, 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.

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 16, 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.

Table 15: 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

Cost

We observe that the major factor that impacts the analysis time for test generation is the complexity of the query embedded in a method. If a query joins multiple tables, the exploration of checking database schema constraints for each table is linearly increased. Meanwhile, if a table contains a large number of attributes, high cost is also incurred. In the implementation of our approach, we apply one of Pex's API methods called `PexAssume()` to reduce the cost of exploring constraints. `PexAssume()` is to filter out undesirable test inputs. By using `PexAssume()`, it is beneficial to guarantee that database schema constraints are always enforced without unnecessary negations. Complexity of the qualification in a query also influences the analysis time as evaluating the conditions has been transformed into normal program code in our approach.

Another observation from the running cost that we would like to point out is related to Pex's path exploration. As aforementioned, we evaluate the qualification in a query by transforming it into normal program code. For example, the qualification in a query is expressed by a boolean combination of conditions connected by program logical connectives. A generated record that satisfies the whole qualification should satisfy all the conditions. However, when Pex explores a branch, it neglects to explore any subsequent boolean condition but starts a new run, if it finds that the first condition does not hold. Thus, to make all the conditions true, Pex takes more runs, whose number is linear to the number of conditions. In practice, to improve the

efficiency, we force Pex to consider all the conditions together in one time, still using Pex's API method `PexAssume()`.

Since the existing approach [38] 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 15, 16, and 17. 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 the three applications, the results show that our approach use much less running time than MODA does.

Table 16: 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

Table 17: Evaluation results on UnixUsage

No.	method	total (blocks)	covered(blocks)			time(seconds)	
			others	SynDB	increase	SynDB	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

5.5 Conclusions

In this research, we have developed a DSE-based approach called *SynDB* for testing database applications. The approach synthesizes new database interactions to replace the original ones. Through this way, we bridge the gap between program-execution constraints and environment constraints. Existing test-generation techniques treat the database as an external component and may face problems when considering constraints within a database application in an insufficient way. Our approach considers both query constraints and database schema constraints, and transform them to normal program code. We use a state-of-the-art DSE engine called Pex to generate effective tests consisting of both program inputs and database states. Empirical evaluations show that our approach achieves higher program code coverage than existing approaches. Part of this work was submitted to a journal and was accepted with major revision [46].

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.

CHAPTER 6: GENERATE TEST FOR MUTATION TESTING

To assure high quality of database applications, testing database applications remains the most popularly used approach. In testing database applications, tests consist of both program inputs and database states. Assessing the adequacy of tests allows targeted generation of new tests for improving their adequacy (e.g., fault-detection capabilities). Comparing to code coverage criteria, mutation testing has been a stronger criterion for assessing the adequacy of tests. Mutation testing would produce a set of mutants (each being the software under test systematically seeded with a small fault) and then check how high percentage of these mutants are killed (i.e., detected) by the tests under assessment. However, existing test-generation approaches for database applications do not provide enough support for killing mutants in database applications (in either program code or its embedded or resulted SQL queries). In this research, we develop an approach called *MutaGen* that conducts test generation for mutation testing on database applications. In our approach, we first correlate various constraints within a database application through constructing synthesized database interactions and transforming the constraints from SQL queries into normal program code. Based on the transformed code, we generate program-code mutants and SQL-query mutants, and then derive and incorporate

query-mutant-killing constraints into the transformed code. Then, we generate tests to satisfy query-mutant-killing constraints. Evaluation results show that *MutaGen* can effectively kill mutants in database applications, and *MutaGen* outperforms existing test-generation approaches for database applications in terms of strong mutant killing.

6.1 Illustrative Example

Mutation testing is a fault-based software testing technique that is intensively studied for evaluating the adequacy of tests [19]. The original program under test is mutated into a set of new programs, called *mutants*, caused by syntactic changes following a set of rules. The mutants are *(strongly) killed* if running the mutants against given tests produces different results than the results of the original program. Killing more mutants reflects better adequacy and higher reliability of the tests under assessment.

However, automatically producing tests that can kill mutants could be very time-consuming and even intractable [20], because a short program may contain a large number of mutants. To deal with the expensiveness of mutation testing, Howden et al. [32] proposed *weak mutation testing* that focuses on intermediate results or outputs from components of the program under test. Instead of checking mutants after the execution of the entire program, the mutants need only to be checked immediately after the mutated components. Researchers also developed subsets of mutation operators to reduce time or space resources exhausted by large number of mutants.

For example, Offutt et al. [42] proposed that five mutation operators (ABS, AOR, ROR, LCR, and UOI) can perform as effectively as all the 22 mutation operators [21].

Mutation testing has also been applied to detect faults in SQL queries. Tuya et al. proposed a set of mutation operators [59] and developed a tool called SQLMutation [57] that implements this set of mutation operators to generate SQL-query mutants. Briefly, the mutation operators are organized into four categories:

SC - SQL clause mutation operators: mutate the most distinctive features of SQL (e.g., clauses, aggregate functions.).

OR - Operator replacement mutation operators: extend the expression modification operators.

NL - NULL mutation operators: mutants related with incorrect treatment of NULL values.

IR - Identifier replacement mutation operators: replacement of operands and operators (e.g., replacement of columns or constants.).

In our approach, for database applications, SQL queries are considered as components of the program under test. Thus, applying *weak mutation testing* by seeding faults [59] to the queries can reflect the adequacy of the associated test database states.

We give a motivating example to illustrate the necessity of generating sufficient

```
01:public int calcStat(int inputAge) {
02:  int zip = 28223, count = 0;
03:  SqlConnection sc = new SqlConnection();
04:  sc.ConnectionString = "..";
05:  sc.Open();
06:  string query = "SELECT C.SSN, C.income,"
   +" M.balance FROM customer C, mortgage M"
   +" WHERE C.age='" + inputAge +"' AND"
   +" C.zipcode='"+ zip + "' AND C.SSN = M.SSN";
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 diff = income - balance;
13:     if (diff > 50000){
14:        count++;}}
15:  return count;}
```

Figure 18: A code snippet from a database application in C#

Table 18: 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, 20, 30}
zipcode	Int	[00001, 99999]			
age	Int	[0, 100]	balance	Int	[2000, Max)
income	Int	[100000, Max)			

database states for mutation testing on database applications.

The code snippet in Figure 18 gives a portion of C# code from a database application that calculates some statistics related to customers' mortgages. The schema of the associated database is shown in Table 18. The method `calcStat` sets up database connection (Lines 03-05), constructs a query (Line 06), and executes the query (Lines 07-08). The query contains two program variables: a local variable `zip` and a program-input parameter `inputAge`. The returned records are then iterated

Table 19: Program inputs and database states to cover paths for program code in Figure 18

input	customer table						mortgage table		
inputAge	SSN	zipcode	name	gender	age	income	SSN	year	balance
30	001	28223	Alice	F	30	100000	001	20	30000
40	002	28223	Bob	M	40	150000	002	20	30000

(Lines 09-14). For each record, a variable `diff` is calculated from the values of the columns `C.income` and `M.balance`. If `diff` is greater than `50000`, a counter variable `count` is increased (Line 14). The method finally returns the value of `count` (Line 15).

To test the preceding method in Figure 18 for achieving high structural coverage, existing test-generation approaches [38, 53] can generate both program inputs and database states to cover feasible paths. For example, the generated values for input `inputAge` and corresponding database records shown in Table 19 could achieve full code coverage: a default value `inputAge = 0` and an empty database state covers the path where `Line 09 = false`; `inputAge = 30` and the record whose column `SSN = 001` covers the path where `Line 09 = true, Line 13 = false`; `inputAge = 40` and the record whose column `SSN = 002` covers the path where `Line 09 = true, Line 13 = true`.

However, in terms of mutation testing, tests in Table 19 are not enough. Killing mutants in database applications requires more program inputs and multiple database records so that executing the program with these inputs against the database could produce different results. For example, in Figure 18, for a mutant in Line 13 where `diff > 50000` is mutated to `diff >= 50000`, none of the values for `inputAge` in Table 19

could kill this mutant as the final program outputs are same. Similarly, for a mutant of the query in Line 06 where the condition `C.age = 'inputAge'` is mutated to `C.age <= 'inputAge'`, all values for `inputAge` still could not kill this SQL-query mutant [19]. Hence, for database applications, achieving valid mutant-killing performance requires both effective program inputs and sufficient database states.

6.2 Problem Formalization and Proposed Solution

Comparing to code coverage criteria (a popular type of testing requirements), mutation testing [19] has been a stronger criterion for assessing the adequacy of tests. Mutation testing would produce a set of mutants (each being the software under test systematically seeded with a small fault) and then check how high percentage of these mutants are killed (i.e., detected) by the tests under assessment. Other than traditional mutation testing where mutants exist in normal program code, Tuya et al. [59, 57] proposed a set of mutation operators for SQL queries and a tool called SQLMutation that implements these mutation operators to generate SQL-query mutants. To assess the adequacy of tests for Java database applications, Zhou et al. [72] developed a tool called JDAMA based on the mutation operators for SQL queries [59].

To kill generated mutants, test generation for mutation testing has been addressed [20, 70]. However, for mutation testing on database applications, tests consist of both program inputs and database states. Thus, these approaches become inapplicable for database applications since sufficient and supportive back-end database states are

[19] Although for `inputAge = 40`, the mutant `C.age <= 'inputAge'` is weakly killed because executions of the original query and this mutant on Table 19 produce different result sets.

required for generated tests. Focusing on test generation for database application testing, some recent approaches [38, 53, 44] have been proposed to automatically generate database states and program inputs to achieve various testing requirements such as high code coverage. However, these approaches do not consider mutation testing as the main goal and cannot provide effective support for killing mutants in database applications.

For a database application, a mutant may occur in either normal program code or SQL queries. Generating appropriate program inputs and sufficient database states to kill a mutant requires collecting and satisfying constraints for killing that mutant. Typically, within a database application, a mutant in normal program code can affect the query-construction constraints (where constraints come from the sub-paths explored before the query execution) and query-result-manipulation constraints (where constraints come from the sub-paths explored for iterating through the query result), while a mutant in SQL queries can affect the query constraints (where constraints come from conditions in a query's WHERE clause). Test generation by applying a constraint solver on the collected constraints faces great challenges because a constraint solver can deal with *program-execution constraints* (e.g., query-construction constraints and query-result-manipulation constraints) but cannot directly handle *environment constraints* (e.g., query constraints).

Existing test-generation approaches [38, 53] for database applications choose to consider *program-execution constraints* and *environment constraints* separately. Thus,

when applying existing approaches [38, 53] for mutation testing on database applications, the design decision of these approaches requires a whole constraint system for each mutant's killing, making the whole process very costly or even infeasible [46]. On the other hand, although a recent approach called PexMutator [70] incorporates all the mutant-killing constraints into the program under test, the approach still cannot directly correlate *program-execution constraints* and *environment constraints* for database applications, thus not being able to generate sufficient database states.

To address these issues, in this paper, we propose a new approach called *MutaGen* (Test Generation for Mutation Testing on Database Applications) for killing mutants in database applications based on our previous SynDB framework [46]. The SynDB framework is based on Dynamic Symbolic Execution (DSE) [24, 49, 55] and correlates program-execution constraints and environment constraints in a database application. It constructs synthesized database interactions and transforms the original program under test into another form that the synthesized database interactions can operate on. Meanwhile, a synthesized object is constructed to replace the physical database state and the query constraints are transformed into normal program code. The framework focuses on generating program inputs and database states to achieve high program code coverage. In *MutaGen*, we leverage SynDB as a supporting mechanism for mutation testing on database applications.

To generate mutants that occur in the program code, we apply an existing code-mutation tool [70] on the code transformed with the SynDB framework. To generate

```
01:public int calcStat(int inputAge,
                       DatabaseState dbState) {
02:  int zip = 28223, count = 0;
03:  SynSqlConnection sc = new SynSqlConnection(dbState);
04:  sc.ConnectionString = "..";
05:  sc.Open();
06:  string query = "SELECT C.SSN, C.income,"
   +" M.balance FROM customer C, mortgage M"
   +" WHERE C.age='" + inputAge + "' AND"
   +" C.zipcode='" + zip + "' AND C.SSN = M.SSN";
07:  SynSqlCommand cmd = new SynSqlCommand(query, sc);
08:  SynSqlDataReader results = cmd.ExecuteReader();
09:  while (results.Read()){
10:     int income = results.GetInt(1);
11:     int balance = results.GetInt(2);
12:     int diff = income - balance;
13:     if (diff > 50000){
14:        count++;}}
15:  return count;}
```

Figure 19: Code transformation for example code in Figure 18

SQL-query mutants, we apply an existing SQL-query-mutation tool [57] to generate SQL-query mutants at query-issuing points. We then derive query-mutant-killing constraints considering both the original query and its mutants. We finally incorporate the derived constraints into the transformed code. Specifically, solving these query-mutant-killing constraints helps produce a database state on which running the original query and its mutants can cause different query results, thus killing the corresponding SQL-query mutants. The transformed code is able to guide DSE to collect constraints for both program inputs and database states. By applying a constraint solver on the collected constraints, we generate effective tests for killing both program-code mutants and SQL-query mutants.

```
public class customerTable {
    public class customer {//define attributes;}
    public List<customer> customerRecords;
    public void checkConstraints() {
        /*method for checking schema constraints*/;}}
public class mortgageTable {
    public class mortgage {//define attributes;}
    public List<mortgage> mortgageRecords;
    public void checkConstraints() {
        /*method for checking schema constraints*/;}}

public class DatabaseState {
  public customerTable customerT = new customerTable( );
  public mortgageTable mortgageT = new mortgageTable( );
  public void checkConstraints(){
        /*check constraints for each table*/;}}
```

Figure 20: Synthesized database state

6.3 Approach

In this section, we present details of our *MutaGen* approach. The approach is developed based on our previous SynDB framework [46]. The framework transforms the original program under test into another form to correlate program-execution constraints and environment constraints. It constructs new synthesized database interactions to replace the original ones for the program under test. For example, the transformed code of the example code in Figure 18 is shown in Figure 19. According to the schema in Table 18, we construct a synthesized database state shown in Table 20. In SynDB framework, we mainly focus on generating tests to achieve high program code coverage. In *MutaGen*, we leverage SynDB as a supporting mechanism for mutation testing.

Base on the transformed code with SynDB framework [46], *MutaGen* conducts mutant killing for database applications from two aspects: killing mutants in original

normal program code and killing SQL-query mutants. Based on the transformed code, *MutaGen* seeds code-mutant-killing constraints by applying an existing mutant-generation tool [70]. To kill SQL-query mutants, *MutaGen* calls a query-mutant-generation tool [57] to generate SQL-query mutants at query-issuing points, derives query-mutant-killing constraints, and inserts the constraints into the transformed code. Thus, applying a DSE engine on the modified transformed code to satisfy the weak-mutant-killing constraints is able to generate both effective program inputs and sufficient database states to weakly kill program-code mutants and SQL-query mutants.

6.3.1 Killing Program-Code Mutants

Mutants in original program code may affect test generation of database states because variables in the mutated statements may be data-dependant on the database attributes of the returned query result. For example, in Figure 18, the variable `diff` in Line 13 is derived from database attributes `C.income` and `M.balance`. Hence, mutants of the statement in Line 13 will cause changes to the constraints for generating database states.

In our approach, *MutaGen* applies a tool called PexMutator [70] on the transformed code of the original program under test. PexMutator is a mutant-generation tool that constructs weak-mutant-killing constraints to guide test generation using *sufficient mutation operators*. In the literature, Offutt et al. [42] proposed that five mutation operators (called *sufficient mutation operators*: ABS, AOR, ROR, LCR, and UOI)

```
12:     ...
13a:    if(((diff>50000) && !(diff>=50000)) ||
           (!(diff>50000) && (diff>=50000)));
           //to weakly kill the mutant diff>=50000
13b:    if(((diff>50000) && !(diff==50000)) ||
           (!(diff>50000) && (diff==50000)));
           //to weakly kill the mutant diff==50000
13c:    if(((diff>50000) && !(diff!=50000)) ||
           (!(diff>50000) && (diff!=50000)));
           //to weakly kill the mutant diff!=50000
        ...
13:     if (diff > 50000){
14:       count++;}}
15:  return count;}
```

Figure 21: A code snippet of applying PexMutator on the transformed code in Figure 19

could achieve as effective performance as all the 22 mutation operators [21].

Note that in the transformed code, program-execution constraints affected by mutants of original program code have been correlated with query constraints. Thus, satisfying these generated weak-mutant-killing constraints will provide sufficient constraints for generating database states to help kill corresponding program-code mutants. In *MutaGen*, applying PexMutator on the transformed code will not affect the implementations of our constructed synthesized database interactions, because PexMutator only focuses on the specific program (e.g., the program under test) indicated by *MutaGen*. After introducing the weak-mutant-killing constraints, we apply a DSE engine (e.g., Pex for .NET [55]) on the transformed code to generate database records.

Figure 21 gives a code snippet of applying PexMutator on the transformed code shown in Figure 19. For example, at the mutation point in Line 13, the generated weak-mutant-killing constraints (we list three of them) for the statement

Table 20: Generated tests for program code in Figure 19 to weakly kill the three mutants shown in Figure 21

int inputAge	DatabaseState dbState								
	dbState.Customer						dbState.Mortgage		
	SSN	name	gender	zipcode	age	income	SSN	year	balance
50	003	AAA	F	28223	50	100000	003	30	50000
50	004	BBB	M	28223	50	100000	004	30	40000
50	005	CCC	M	28223	50	100000	005	30	60000

`if(diff>50000)` are inserted before Line 13. The variable `diff` is calculated from the attributes `C.income` and `M.balance`. Then, applying a DSE engine on the modified transformed code will generate appropriate values for program inputs `inputAge` and `dbState` to cover the true branches of Lines 13a, 13b, and 13c, weakly killing the corresponding three mutants `diff>=50000`, `diff==50000`, and `diff!=50000`. For example, tests to kill the three mutants are shown in Table 20.

Note that although PexMutator provides a general way of inserting weak-mutant-killing constraints into the program code, combining PexMutator with existing test-generation approaches [38, 53] cannot help directly generate tests to kill program-code mutants in database applications. Program-execution constraints and query constraints are still not correlated causing that a whole constraint system is needed for each mutant killing.

6.3.2 Killing SQL-Query Mutants

Mutants occurring in SQL queries will directly affect constraints for generating database states. To weakly kill a SQL-query mutant, *MutaGen* generates database

records to expose the difference between the original query and the mutant so that their executions produce different results.

In *MutaGen*, the transformed code has incorporated the query constraints into normal program code. We first identify query-issuing points by finding corresponding method signatures (e.g., `SynSqlCommand.ExecuteReader()`). Then, at each query-issuing point, we get the symbolic query and call the tool SQLMutation [57] to generate its mutants. SQLMutation is developed by Tuya et al. [57] that automatically generates SQL-query mutants (providing each mutant's form, type, and generation rule) based on a set of mutation operators [59] for SQL queries. As aforementioned, the mutation operators are organized into four categories of which the SC operators mainly focus on the main clauses (e.g., SELECT clause) and the other operators (OR, NL, and IR) focus on the conditions in the WHERE clause. For example, one of the mutants generated by the OR operators using SQLMutation for the query in Figure 19 is shown in Figure 22, where the condition `C.age = 'inputAge'` is mutated to `C.age >= 'inputAge'`.

Next, we derive query-mutant-killing constraints based on the original query and its mutants and insert these constraints into the transformed code [20]. Algorithm 8 gives details of how to derive the query-mutant-killing constraints. The algorithm mainly deals with mutants generated by OR, NL, and IR operators (e.g., mutating operators or column names in the WHERE clause). In Algorithm 8, the inputs consist

[20]To avoid causing syntactic errors, in the transformed code, we insert these constraints before the original query.

of a constructed synthesized database state $SynDB$ and a symbolic query Q and the output is a set of program statements that contain conditions whose exploration helps derive constraints for killing mutants of a given query. In Algorithm 8, we construct an empty statement set S (Line 1) and a SQL-query mutant set Q_m by calling $SQLMutation(Q)$ (Line 2). We retrieve Q's WHERE clause $s1$ using a SQL parser (Line 4). In Lines 5-17, for each mutant q in Q_m, if q is generated by the mutation operators OR, NL, or IR, we retrieve its WHERE clause $s2$ and construct a query-mutant-killing constraint $s = (!s1$ AND $s2)$ OR $(s1$ AND $!s2)$. Note that if a record r satisfies conditions in s, r can only satisfy either $s1$ or $s2$, causing different execution results when executing Q and q against r. We then check the expressions in s and replace the columns in s with their corresponding names from the constructed synthesized database state $SynDB$. We add the query-mutant-killing constraint s to the set S. After dealing with all the mutants in Q_m, the algorithm finally returns the set S. For example, to weakly kill the mutant shown in the top of Figure 22, the constructed query-mutant-killing constraints based on the query's WHERE clause are shown in the bottom of Figure 22.

To deal with SQL-query mutants generated by the SC operators, we mainly focus on cardinality constraints as killing mutants generated by SC operators requires different sizes of qualified records. For example, a LEFT OUTER JOIN keyword requires the two joined tables contain different numbers of qualified records for conditions in the WHERE clause. To kill such mutants, we specify different cardinality constraints in

Algorithm 8 *QMutantGen*: Generate query-mutant-killing constraints

Input: Synthesized database state $SynDB$, a symbolic query Q
Output: A set of program statements S

```
1: Statement set S = ∅;
2: Query mutant set Q_m = SQLMutation(Q);
3: Mutation operator set OP = {OR, NL, IR};
4: String s1 = Q.whereClause;
5: for each query q in Q_m do
6:    if q.type ∈ OP then
7:       String s2 = q.whereClause;
8:       String s = (!s1 AND s2) OR (s1 AND !s2);
9:       for each expression e in s do
10:         for each column c in e do
11:            Variable v = findColumn(c, SynDB);
12:            Replace(c, v);
13:         end for
14:      end for
15:      S = S ⋃ s;
16:   end if
17: end for
18: return S;
```

the transformed code for the query results.

6.4 Evaluation

In our evaluation, we seek to evaluate the effectiveness of *MutaGen* by investigating the following research questions:

RQ1: What is the performance of *MutaGen* in generating tests to kill mutants in database applications?

RQ2: What is the performance of *MutaGen* comparing with existing test-generation approaches [38, 53] in terms of mutant killing and code coverage for database application testing?

```
A mutant generated by OR operators using SQLMutation:
OR(query) = "SELECT C.SSN, C.income, M.balance
          FROM customer C, mortgage M
          WHERE C.age >= 'inputAge'
          AND C.zipcode = 'zip' AND C.SSN = M.SSN

Constructed query-mutant-killing constraints:
((SynDB.customerTable.age == 'inputAge' AND
SynDB.customerTable.zipcode == 'zip'
AND SynDB.customerTable.SSN == SynDB.mortgageTable.SSN) &&
!(SynDB.customerTable.age >= 'inputAge' AND
SynDB.customerTable.zipcode == 'zip'
AND SynDB.customerTable.SSN == SynDB.mortgageTable.SSN))||
((!(SynDB.customerTable.age == 'inputAge' AND
SynDB.customerTable.zipcode == 'zip'
AND SynDB.customerTable.SSN == SynDB.mortgageTable.SSN)) &&
(SynDB.customerTable.age >= 'inputAge' AND
SynDB.customerTable.zipcode == 'zip'
AND SynDB.customerTable.SSN == SynDB.mortgageTable.SSN))
```

Figure 22: Generating query-mutant-killing constraints for the query shown in Figure 19

6.4.1 Subject Applications and Setup

We conduct the empirical evaluation on two open source database applications `RiskIt` and `UnixUsage`. The introductions to these two applications are previously presented in Chapter 1. Both applications contain existing records in their databases but we do not use them because our approach is able to conduct test generation from scratch. We use Pex [55] as the DSE engine.

The experimental procedure is as follows. To evaluate how *MutaGen* performs in killing program-code mutants, we generate compiled file for the program under test and apply the tool PexMutator [70] on the compiled file to generate meta-program that has incorporated weak-mutant-killing constraints. We then generate compiled files for the other programs (e.g., synthesized database interfaces constructed by our SynDB framework). We send these compiled files together with the meta-program of

the program under test to Pex for test generation. We insert the generated database records back to the real database, run the original program under test using the generated program inputs, and record the number of weakly killed program-code mutants at each mutation point. To evaluate how *MutaGen* performs in killing SQL-query mutants, we call the tool SQLMutation [57] to generate SQL-query mutants at each query-issuing point and use *MutaGen* to generate tests. We insert the generated database records back to the real database and run the original program with our generated program inputs. To measure the number of weakly killed SQL-query mutants, we compare the returned result sets from executions of the original query and its mutants by checking the metadata (e.g., number of rows, columns, and contents). For both two kinds of mutants, we also record the numbers of strongly killed mutants by comparing final results of the program.

Table 21: Evaluation Results(NOM: Number of Mutants, MG: *MutaGen*, EA: Existing Approaches, Inc%: Percentage Increase)

Subjects	Methods	Program-Code Mutants							SQL-Query Mutants							Coverage (covered blocks)			
			Weakly Killed			Strongly Killed				Weakly Killed			Strongly Killed						
		NOM	MG	EA	Inc%	MG	EA	Inc%	NOM	MG	EA	Inc%	MG	EA	Inc%	Total	MG	EA	Inc%
RiskIt	filterZipcode	24	22	18	16.7	16	13	12.5	14	11	6	35.7	8	6	14.3	42	38	28	23.8
	filterEducation	20	18	14	20.0	12	9	15.0	62	43	31	19.4	32	23	14.5	41	37	27	24.4
	filterMaritalStatus	20	18	14	20.0	12	9	15.0	14	11	6	35.7	8	6	14.3	41	37	27	24.4
	getAllZipcode	27	23	19	14.8	17	14	11.1	85	69	45	28.2	51	33	21.2	39	37	17	51.3
	filterEstimatedIncome	24	22	18	16.7	16	13	12.5	122	98	68	24.6	70	53	13.9	58	54	44	17.2
	getOneZipcode	32	28	25	9.4	21	17	12.5	14	11	6	35.7	8	6	14.3	34	32	23	26.5
	getValues	44	38	33	11.4	29	22	15.9	56	44	28	28.6	32	22	22.7	107	99	68	29.0
	userinformation	37	33	26	18.9	27	21	16.2	70	58	39	27.1	41	29	17.1	61	57	51	9.8
	updatestability	42	37	29	19.0	30	21	21.4	64	50	34	25.0	35	24	17.2	79	67	75	10.1
all methods (total)		270	239	196	16.3	180	140	14.7	501	395	263	28.9	285	202	16.7	502	458	360	21.3
UnixUsage	raceExists	12	9	7	16.7	6	5	8.3	16	13	8	31.3	9	6	23.1	11	11	7	36.4
	transcriptExist	12	9	7	16.7	6	5	8.3	16	13	8	31.3	9	6	23.1	11	11	7	36.4
	retrieveMaxLineNo	9	9	8	11.1	7	5	22.2	14	11	7	28.6	5	3	14.3	10	10	7	30.0
	getUserInfoBy	14	11	9	14.3	7	6	7.1	16	13	8	31.3	9	6	18.8	47	47	15	68.1
	doesUserIdExist	12	9	7	16.7	6	5	8.3	12	10	7	25.0	7	4	25.0	10	10	9	10.0
	getPrinterUsage	16	14	11	18.8	9	5	25.0	36	32	22	27.8	28	17	30.6	34	34	27	20.1
all methods (total)		75	61	49	15.7	41	32	13.2	110	92	60	29.2	67	42	22.5	123	123	75	33.5

To compare *MutaGen* with existing test-generation approaches [38, 53], we simulate these approaches using our SynDB framework [46], by not incorporating either program-mutant-killing constraints or query-mutant-killing constraints into the transformed code. We insert the database records generated in this step back to the real database and run the program under test with generated program inputs to measure the numbers of weakly and strongly killed mutants and the code coverage.

6.4.2 Results

We give detailed evaluation results in Table 21. In the table, Column 1 lists the subject applications and Column 2 lists method names; the remaining columns give comparisons of performance using tests generated by *MutaGen* and existing approaches [38, 53] from three perspectives: killing program-code mutants (Columns 3-9), killing SQL-query mutants (Columns 10-16), and code coverage (Columns 17-20), respectively. For mutant killing (Columns 3-9 and 10-16), we list the total number of mutants, the number of weakly killed mutants, the number of strongly killed mutants, and percentage increase. For code coverage (Columns 17-20), we list the number of total blocks, covered blocks, and percentage increase. For example, for the first method "filterZipcode" in `RiskIt`, there are 24 program-code mutants in total, of which our approach weakly kills 22 and strongly kills 16, achieving better mutant-killing ratio (16.7% and 12.5% increase, respectively) than existing approaches. For the total 14 SQL-query mutants, we also achieve better mutant-killing performance (35.7% increase for weak-killing and 14.3% increase for strong-killing). Meanwhile,

for the total 42 blocks, *MutaGen* achieves better code coverage (23.8% increase) than existing approaches.

In summary, to answer RQ1, *MutaGen* can effectively kill a large portion of both program-code mutants and SQL-query mutants for database applications, leaving a few hard-to-kill mutants. To answer RQ2, *MutaGen* outperforms existing test-generation approaches in terms of mutant killing. For example, for `RiskIt`, *MutaGen* achieves a 16.3% percentage increase on average in weakly killing program-code mutants and a 28.9% percentage increase on average in weakly killing SQL-query mutants, while the average increases are 14.7% and 16.7% in strong mutant killing for the aforementioned two kinds of mutants, respectively. Meanwhile, *MutaGen* achieves higher code coverage (21.3% increase for `RiskIt` and 33.5% increase for `UnixUsage`).

6.5 Conclusions

In our research, we have developed an approach called *MutaGen* that generates tests for mutation testing on database applications. In our approach, we leverage our previous SynDB framework [46] that relates program-execution constraints and query constraints within a database application. We incorporate weak-mutant-killing constraints for the original program code and query-mutant-killing constraints for the SQL queries into the transformed code, guiding DSE to generate both effective program inputs and sufficient database states to kill mutants. Evaluation results show that *MutaGen* achieves high effectiveness and outperforms existing test-generation approaches in killing both program-code mutants and SQL-query mutants. Part of

this work was included in a Technical Report at UNC Charlotte [45].

In future work, we plan to investigate how to generate program inputs based on a given database state for mutation testing. We also plan to investigate techniques of augmenting existing tests to detect logical faults in database applications.

CHAPTER 7: CONCLUSIONS

In this research, we investigate test generation for database application testing, including generating both program input values and database states from various testing aspects, under various testing requirements and environments.

Achieving higher block or branch coverage is a good indicator of the quality of test inputs, as more potential faults could be exposed during the execution. In the literature, advanced structural coverage criteria are enforced to ensure desired features of tests, so that testers can detect more faults that occur in boundaries or involve complex logical expressions. In this research, we complement the traditional block or branch coverage by developing an approach that generates database states to achieve advanced code coverage including boundary value coverage(BVC) and logical coverage(LC) for program under test. We examine the close relationships among program variables, embedded SQL queries, and branch conditions in source code. We then derive constraints such that tests satisfying those constraints can achieve the target coverage criteria. Evaluations on two real database applications show that our approach assists a state-of-the-art DSE tool called Pex for .NET to generate test database states that can effectively achieve both BVC and LC.

We then investigate the problem context that how to conduct program-input gen-

eration given an existing database state. We formally present the problem under this testing scenario, which is the first problem formalization for program-input generation given an existing database state to achieve high code coverage. Although it is desirable to use an existing database state when conduct test generation, there is a significant challenge that there exists a gap between program-input constraints derived from the program and those derived from the given existing database state. To address this problem, we develop a novel program-input-generation approach based on symbolic execution and query formulation for bridging the gap between program-input constraints from the program and from the given existing database state. In our research, we examine close relationships among program inputs, program variables, branch conditions, embedded SQL queries, and database states. We formulate auxiliary queries based on the collected various intermediate information during DSE's exploration. The constructed auxiliary queries treat those database attributes related with program inputs as the target selection and incorporate those path constraints related with query result sets into selection condition. Executing these auxiliary queries finally return effective program input values for achieving code coverage. We conduct evaluations on two real database applications to assess the effectiveness of our approach upon Pex and the results show that our approach assists Pex to generate program inputs that achieve higher code coverage.

We investigate the problem of considering various constraints within a database application in a insufficient way regarding existing test-generation approaches, caused by

premature concretization made in existing DSE-based approaches. Such premature concretization comes from the significant problem when performing symbolic execution of database interaction API methods, leading constraint conflicts among query constraints, database schema constraints, and query-result-manipulation constraints. In this research, we develop a DSE-based test generation approach called *SynDB* to deal with this problem, even when the associated physical database is not available. We comprehensively investigate the relations of program-execution constraints (e.g., query-construction constraints and query-result-manipulation constraints) and environment constraints (e.g., query constraints and database schema constraints). Our technique is the first work that uses a fully symbolic database. We bridge the gap between program-execution constraints and environment constraints within database applications by constructing synthesized database interactions, so that we solve the problem that existing test-generation approaches may face constraint conflicts when generating desirable database states. The synthesized database interactions incorporate query constraints into normal program code. We develop a test-generation approach based on DSE through code transformation, treating symbolically both the embedded query and the associated database state, correlating various parts of constraints. We implement our approach into a prototype. We conduct empirical evaluations on open source packages and the results show that our approach can generate effective program inputs and sufficient database states to achieve higher code coverage than existing DSE-based test generation approaches for database applications.

We also extend our research into test generation for mutation testing, which is another important testing aspect. Mutation testing is considered as a stronger criterion for assessing the adequacy of tests, by measuring the ratio of killed mutants using given tests. However, for database applications, mutants could appear on either program code or SQL queries, corresponding to program-execution constraints (e.g., query-construction constraints and query-result-manipulation constraints) and environment constraints (e.g., query constraints). Because a constraint solver cannot directly handle environment constraints, existing test-generation approaches [38, 53] for database applications choose to consider program-execution constraints and environment constraints separately, posing challenges when a whole constraint system is required for each mutant's killing. To address this problem, in our research, we leveraging our *SynDB* framework as a supporting mechanism. We develop an approach called *MutaGen* that can generate both effective program inputs and sufficient database states to kill mutants, by incorporating weak-mutant-killing constraints for original program code and query-mutant-killing constraints for SQL queries into the transformed code generated by *SynDB* framework. Then, solving these query-mutant-killing constraints collected during DSE's exploration helps produce a database state on which running the original query and its mutants can cause different query results, thus killing the corresponding SQL-query mutants. Experimental results show that *MutaGen* is able to achieve high effectiveness and outperforms existing test-generation approaches in killing both program-code mutants and SQL-query mutants.

Throughout our research, we use Dynamic Symbolic Execution (DSE) as a major background technique and a DSE tool called Pex for .NET as the experimental tool. The evaluation results show that our proposed approaches are able to achieve effective performance from the aforementioned testing aspects.

In future work, we plan to extend our research to performance testing. In the literature, little research has been conducted in connecting functional testing with performance testing. We have investigated how to track constraints over both program inputs and database attributes. Consequently, we are able to use those intermediate constraint results to develop novel technique for applying it on a further performance testing. A close estimate of the performance of a database application can be significant for organizations that need to quantify the potential gain in performance. The performance of a database application heavily depends on the statistical distribution of the database from two perspectives, including the execution time of the query evaluation in the DBMS and the execution time of manipulating the query result set in the application software. The database application can have substantial performance differences on two database states with the same size even if the query results satisfy the same cardinality constraints (which guarantees the same coverage for structural testing).

To achieve this goal, we statically analyze the program statements related to manipulating query result sets to collect parameters that can be configured by testers to allow the proposed approach to automatically generate desirable statistical distribu-

tion of query result sets. For each test objective, we generate several representative database states (e.g., corresponding to the worst, average, and best cases), which can help organizations understand an overall picture of their software performance under various situations. There are two challenges in generating database states for performance testing. First, we need to identify which database attributes control the performance. From technical point of view, we will use the intermediate results during DSE's exploration. Second, for performance testing, we usually need to generate database states with large numbers of tuples. We aim to minimize the number of calls to the constraint solver, by conducting additional optimization work here, such as how to combine two path conditions together, how to use existing database states to instantiate new database states by analyzing their constraints.

In future work, we also plan to extend our techniques to load testing for database applications. Load testing aims to examine program's execution performance under various testing intensity. To generate tests for load testing, we plan to investigate specific code portions that can cause potential performing differences within database applications. Under aforementioned problem contexts discussed in previous chapters, we plan to extend our techniques to deal with multiple queries as there are more complex cases regarding different appearing sequences of queries.

REFERENCES

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

[2] Ahmed, R., Lee, A. W., Witkowski, A., Das, D., Su, H., Zait, M., and Cruanes, T. Cost-based query transformation in oracle. In VLDB (2006), pp. 1026–1036.

[3] Ammann, P., Offutt, A. J., and Huang, H. Coverage criteria for logical expressions. In ISSRE (2003), pp. 99–107.

[4] Bati, H., Giakoumakis, L., Herbert, S., and Surna, A. A genetic approach for random testing of database systems. In VLDB (2007), pp. 1243–1251.

[5] Binnig, C., Kossmann, D., and Lo, E. Testing database applications. In SIGMOD Conference (2006), pp. 739–741.

[6] Binnig, C., Kossmann, D., and Lo, E. Multi-RQP: generating test databases for the functional testing of OLTP applications. In DBTest (2008), p. 5.

[7] Bruno, N., and Chaudhuri, S. Flexible database generators. In VLDB (2005), pp. 1097–1107.

[8] Cabal, M. J. S., and Tuya, J. Improvement of test data by measuring sql statement coverage. In STEP (2003), pp. 234–240.

[9] C.Binnig, D.Kossmann, and E.Lo. Reverse query processing. In ICDE (2007), pp. 506–515.

[10] C.Binnig, D.Kossmann, E.Lo, and M.T.Ozsu. QAGen: Generating Query-Aware Test Databases. In SIGMOD (2007), pp. 341–352.

[11] Chays, D., and Deng, Y. Demonstration of agenda tool set for testing relational database applications. In ICSE (2003), pp. 802–803.

[12] Chays, D., Deng, Y., Frankl, P. G., Dan, S., Vokolos, F. I., and Weyuker, E. J. An agenda for testing relational database applications. Softw. Test., Verif. Reliab. 14, 1 (2004), 17–44.

[13] Chays, D., Shahid, J., and Frankl, P. G. Query-based test generation for database applications. In DBTest (2008), p. 6.

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

[15] Clark, S. R., Cobb, J., Kapfhammer, G. M., Jones, J. A., and Harrold, M. J. Localizing SQL faults in database applications. In ASE (2011), pp. 213–222.

[16] Clarke, L. A. A system to generate test data and symbolically execute programs. In IEEE Trans. Softw. Eng.,2(3):215-222 (1976).

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

[18] de la Riva, C., Cabal, M. J. S., and Tuya, J. Constraint-based Test Database Generation for SQL Queries. In AST (2010), pp. 67–74.

[19] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. Hints on test data selection help for the practicing programmer. IEEE Computer 11, 4 (April 1978), 34–41.

[20] DeMillo, R. A., and Offutt, A. J. Constraint-based automatic test data generation. IEEE Trans. Software Eng. 17, 9 (1991), 900–910.

[21] DeMillo, R. A., and Spafford, E. H. The mothra software testing environment.

[22] Deng, Y., and Chays, D. Testing database transactions with agenda. In ICSE (2005), pp. 78–87.

[23] Farre, C., Rull, G., Teniente, E., and Urpi, T. SVTe: a tool to validate database schemas giving explanations. In DBTest (2008), p. 9.

[24] Godefroid, P., Klarlund, N., and Sen, K. DART: directed automated random testing. In PLDI (2005), pp. 213–223.

[25] Godefroid, P., and Luchaup, D. Automatic partial loop summarization in dynamic test generation. In ISSTA (2011), pp. 23–33.

[26] Gould, C., Su, Z., and Devanbu, P. T. Static checking of dynamically generated queries in database applications. In ICSE (2004), pp. 645–654.

[27] Grechanik, M., Csallner, C., Fu, C., and Xie, Q. Is data privacy always good for software testing? In ISSRE (2010), pp. 368–377.

[28] Haftmann, F., Kossmann, D., and Lo, E. Parallel execution of test runs for database application systems. In VLDB (2005), pp. 589–600.

[29] Haftmann, F., Kossmann, D., and Lo, E. A framework for efficient regression tests on database applications. VLDB J. 16, 1 (2007), 145–164.

[30] Halfond, W. G. J., and Orso, A. Command-form coverage for testing database applications. In ASE (2006), pp. 69–80.

[31] Hoag, J. E., and Thompson, C. W. A parallel general-purpose synthetic data generator. vol. 36, pp. 19–24.

[32] Howden, W. E. Weak mutation testing and completeness of test sets. IEEE Trans. Software Eng. 8, 4 (1982), 371–379.

[33] Kapfhammer, G., and Soffa, M. A family of test adequacy criteria for database-driven applications. In ESEC/SIGSOFT FSE (2003).

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

[35] King, J. C. Symbolic execution and program testing. In Commun. ACM, 19(7):385394 (1976).

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

[37] Li, C., and Csallner, C. Dynamic symbolic database application testing. In DBTest (2010), pp. 01–06.

[38] M.Emmi, R.Majumdar, and K.Sen. Dynamic test input generation for database applications. In ISSTA (2007), pp. 151–162.

[39] Microsoft. .NET Framework Class Library: DataTable. *http://msdn.microsoft.com/en-us/library/system.data.datatable.aspx* (2012).

[40] Microsoft. .NET Framework Data Provider for SQL Server. *http://msdn.microsoft.com/en-us/library/system.data.sqlclient.aspx* (2012).

[41] N.Bruno, S.Chaudhuri, and D.Thomas. Generating queries with cardinality constraints for dbms testing. In TKDE (2006).

[42] Offutt, A. J., Rothermel, G., and Zapf, C. An experimental evaluation of selective mutation. In ICSE (1993), pp. 100–107.

[43] Pan, K., Wu, X., and Xie, T. Database state generation via dynamic symbolic execution for coverage criteria. In DBTest (2011), pp. 01–06.

[44] Pan, K., Wu, X., and Xie, T. Generating program inputs for database application testing. In ASE (2011), pp. 73–82.

[45] Pan, K., Wu, X., and Xie, T. Automatic test generation for mutation testing on database applications. In Technical Report UNC Charlotte (2012).

[46] Pan, K., Wu, X., and Xie, T. Guided test generation for database applications via synthesized database interactions. In submission to TOSEM (2012).

[47] Pan, K., Wu, X., and Xie, T. Program input generation for testing database applications using existing database states. In submission to TSE (2012).

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

[49] Sen, K., Marinov, D., and Agha, G. CUTE: a concolic unit testing engine for C. In ESEC/SIGSOFT FSE (2005), pp. 263–272.

[50] Shahriar, H., and Zulkernine, M. Music: Mutation-based sql injection vulnerability checking. In QSIC (2008), pp. 77–86.

[51] S.Khalek, B.Elkarablieh, Laleye, Y., and Khurshid, S. Query-aware test generation using a relational constraint solver. In ASE (2008).

[52] Taneja, K., Grechanik, M., Ghani, R., and Xie, T. Testing software in age of data privacy: A balancing act. In ESEC/FSE (2011), pp. 201–211.

[53] Taneja, K., Zhang, Y., and Xie, T. MODA: Automated Test Generation for Database Applications via Mock Objects. In ASE (2010), pp. 289–292.

[54] Tang, E., Frankl, P. G., and Deng, Y. Test coverage tools for database applications. In Mid-Atlantic Student Workshop on Programming Languages and Systems (2006).

[55] Tillmann, N., and de Halleux, J. Pex-White Box Test Generation for .NET. In TAP (2008), pp. 134–153.

[56] Tillmann, N., and Schulte, W. Mock-object generation with behavior. In ASE (2006), pp. 365–368.

[57] Tuya, J., Cabal, M. J. S., and de la Riva, C. SQLMutation: A tool to generate mutants of SQL database queries. Mutation 2006.

[58] Tuya, J., Cabal, M. J. S., and de la Riva, C. A practical guide to sql white-box testing. SIGPLAN Notices 41, 4 (2006), 36–41.

[59] Tuya, J., Cabal, M. J. S., and de la Riva, C. Mutating database queries. Information and Software Technology 49, 4 (2007), 398–417.

[60] Tuya, J., Cabal, M. J. S., and de la Riva, C. Full predicate coverage for testing sql database queries. vol. 20, pp. 237–288.

[61] Veanes, M., Grigorenko, P., de Halleux, P., and Tillmann, N. Symbolic query exploration. In ICFEM (2009), pp. 49–68.

[62] Willmor, D., and Embury, S. M. A safe regression test selection technique for database-driven applications. In ICSM (2005), pp. 421–430.

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

[64] Wu, X., Sanghvi, C., Wang, Y., and Zheng, Y. Privacy aware data generation for testing database applications. In IDEAS (2005), pp. 317–326.

[65] Wu, X., Wang, Y., Guo, S., and Zheng, Y. Privacy preserving database generation for database application testing. Fundam. Inform. 78, 4 (2007), 595–612.

[66] Wu, X., Wang, Y., and Zheng, Y. Privacy preserving database application testing. In WPES (2003), pp. 118–128.

[67] Wu, X., Wang, Y., and Zheng, Y. Statistical database modeling for privacy preserving database generation. In ISMIS (2005), pp. 382–390.

[68] Xie, T., Tillmann, N., de Halleux, P., and Schulte, W. Fitness-guided path exploration in dynamic symbolic execution. In DSN (2009), pp. 359–368.

[69] Zhang, L., Ma, X., Lu, J., Tillmann, N., de Halleux, J., and Xie, T. Environment modeling for automated testing of cloud applications. IEEE Software, Special Issue on Software Engineering for Cloud Computing 29, 2 (2012), 30–35.

[70] Zhang, L., Xie, T., Zhang, L., Tillmann, N., de Halleux, J., and Mei, H. Test generation via dynamic symbolic execution for mutation testing. In ICSM (2010), pp. 1–10.

[71] Zhang, P., Elbaum, S. G., and Dwyer, M. B. Automatic generation of load tests. In ASE (2011), pp. 43–52.

[72] Zhou, C., and Frankl, P. G. Mutation Testing for Java Database Applications. In ICST (2009), pp. 396–405.

[73] Zhou, C., and Frankl, P. G. Inferential checking for mutants modifying database states. In ICST (2011), pp. 259–268.