2013

Testing database applications using coverage analysis and mutation analysis

Tanmoy Sarkar
Iowa State University

Testing database applications using coverage analysis and mutation analysis

by

Tanmoy Sarkar

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:

Samik Basu, Co-major Professor

Johnny S. Wong, Co-major Professor

Arka P. Ghosh

Shashi K. Gadia

Wensheng Zhang

Iowa State University

Ames, Iowa

2013

DEDICATION

I would like to dedicate this thesis to my parents Tapas Sarkar, Sikha Sarkar and my girlfriend Beas Roy, who were always there to support me to move forward. I would also like to thank my friends and family in USA and in India for their loving guidance and all sorts of assistance during the writing of this work.

TABLE OF CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

ACKNOWLEDGEMENTS viii

ABSTRACT ix

CHAPTER 1. SOFTWARE TESTING FOR DATABASE APPLICATIONS 1

1.1 Background 1

1.2 Driving Problem 6

1.3 Our Solution 7

1.4 Overall Contributions 9

1.5 Organization 10

CHAPTER 2. RELATED WORK 11

2.1 Automated Test Case Generation 12

2.2 Mutation Testing 15

2.3 Database Application Testing 17

CHAPTER 3. ConSMutate: SQL MUTANTS FOR GUIDING CONCOLIC TESTING OF DATABASE APPLICATIONS 19

3.1 Introduction 19

3.1.1 Driving Problem 19

3.1.2 Motivating Example 19

3.1.3 Problem Statement 21

3.1.4 Individual Contributions 21

3.2 ConSMutate Test Case Generator for DB-Applications 21

3.2.1 Generation of Test Cases and Associated Path Constraints Using Application Branch Analyzer 22

3.2.2 Deployment of Mutation Analyzer 23

3.2.3 Deployment of Constraint Solver: Finding Satisfiable Assignment for θ . 27

3.2.4 Correctness Criteria of ConSMutate 28

3.3 Experimental Results 30

3.3.1 Evaluation Criteria 30

3.3.2 Evaluation Test-Bed 31

3.3.3 Summary of Evaluation 31

3.3.4 Execution Time Overhead 35

CHAPTER 4. SynConSMutate: CONCOLIC TESTING OF DATABASE APPLICATIONS VIA SYNTHETIC DATA GUIDED BY SQL MUTANTS 36

4.1 Introduction 36

4.1.1 Driving Problem 36

4.1.2 Motivating Example 36

4.1.3 Problem Statement 38

4.1.4 Individual Contributions 38

4.2 Approach Overview 39

4.2.1 Discussion: Dealing with Nested Queries 47

4.3 Experimental Results 48

4.3.1 Evaluation Criteria 48

4.3.2 Evaluation Test-Bed 49

4.3.3 Summary of Evaluation 49

CHAPTER 5. CONCOLIC TESTING OF DATABASE APPLICATIONS WHILE GENERATING MINIMAL SET OF SYNTHETIC DATA 53

5.1 Introduction 53

5.1.1 Driving Problem 53

5.1.2 Motivating Example 53

5.1.3 Problem Statement 56

5.1.4 Individual Contributions 56

5.2 Approach 57

5.2.1 Approach Overview 58

5.3 Future Work 71

CHAPTER 6. CONCLUSIONS AND FUTURE WORK 73

6.1 Summary 73

6.2 Uniqueness 74

6.3 Discussion 74

6.4 Future Directions 76

BIBLIOGRAPHY 80

LIST OF TABLES

Table 1.1 Mutation Operation Example 4

Table 3.1 Table **coffees** 20

Table 3.2 Sample mutant generation rules and mutant killing-constraints 24

Table 3.3 Mutants and results for test case (11, 1) 28

Table 3.4 Method names and corresponding Program Identifiers 30

Table 4.1 **coffees** Table schema definition 37

Table 4.2 Updated **coffees** Table in the database 37

Table 4.3 **coffees** Table with new synthetic data 46

Table 4.4 Mutants and new Results for test case $(x = 0)$ 47

Table 5.1 New Table **coffees** 55

Table 5.2 Table **distributor** 55

Table 5.3 Updated Table **coffees** 62

Table 5.4 Mutants and results for test case (8) 68

Table 5.5 Final Updated Table **coffees** 71

Table 5.6 Mutants and new Results for test case $(x = 10)$ 71

LIST OF FIGURES

Figure 1.1 General Control Flow for Assessing Test Input Quality using Mutation Analysis . . . 5
Figure 1.2 Broader Problem Scenario in the field of Database Application Testing 7
Figure 1.3 Our Solution Approach . . . 7

Figure 2.1 Sample Code Fragment . . . 13

Figure 3.1 Framework for ConSMutate . . . 22
Figure 3.2 Comparison between Pex and ConSMutate in terms of quality . . . 32
Figure 3.3 Execution time comparison between Pex and ConSMutate . . . 34

Figure 4.1 Framework For SynConSMutate . . . 40
Figure 4.2 Actual code snippet of the Pseudocode from Section 4.1.2 . . . 41
Figure 4.3 Transformed code snippet produced by SynDB for the code in Figure 4.2 42
Figure 4.4 Synthesized Database State . . . 42
Figure 4.5 Comparison among SynDB, Emmi et. al.'s approach and SynConSMutate for UnixUsage . . . 50
Figure 4.6 Comparison among SynDB, Emmi et. al.'s approach and SynConSMutate for RiskIt . . . 51

Figure 5.1 New Framework for Testing Database Applications . . . 61

Figure 6.1 Overall Impact of Our Work . . . 78

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Samik Basu and Dr. Johnny S. Wong for their guidance, patience and support throughout this research and the writing of this thesis. Their insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Arka P. Ghosh, Dr. Shashi K. Gadia and Dr. Wensheng Zhang. I would additionally thank my lab-mates Michelle Ruse, Chris Strasburg, Zachary Oster and Debasis Mandal for helping and supporting me in all stages of my graduate career.

I want to thank all my wonderful professors Dr. Samik Basu, Dr. Hridesh Rajan, Dr. David Fernández-Baca, Dr. Jack Lutz, Dr. Giora Slutzki, Dr. Lu Ruan, Dr. Doug Jacobson for teaching some of the best and interesting computer science courses in the most effective manner.

Moreover, I would like to thank all my teaching instructors, Dr. Johnny Wong, Dr. David M. Weiss, Dr. Simanta Mitra, Dr. Shashi K. Gadia, Dr. Yan-bin Jia, Dr. Andrew S. Miner, Dr. Steven Kautz for their wonderful support. Their advice has not only helped me to do my job effectively but also I have learnt a lot about teaching. Because of them I enjoyed my teaching duty all throughout my PhD. career.

Thanks to Abigail Andrews, Darlene Brace, Maria-Nera Davis, Linda Dutton, Cindy Marquardt, and Laurel Tweed for always are being so helpful, approachable, and friendly. Finally, I would like to thank my parents, my family and friends in India for their patience and support to complete my PhD study. Special thanks to my girlfriend, Beas. She has been extremely supportive during the writing of this work and has patiently helping me through the final stretch. I am thankful to God, because of whom all things are possible.

ABSTRACT

Database applications are built using two different programming language constructs: one that controls the behavior of the application, also referred to as the *host language*; and the other that allows the application to access/retrieve information from the back-end database, also referred to as the *query language*. The interplay between these two languages makes testing of database applications a challenging process. Independent approaches have been developed to evaluate test case quality for host languages and query languages. Typically, the quality of test cases for the host language (e.g., Java) is evaluated on the basis of the number of lines, statements and blocks covered by the test cases. High quality test cases for host languages can be automatically generated using recently developed concolic testing techniques, which rely on manipulating and guiding the search of test cases based on carefully comparing the concrete and symbolic execution of the program written in the host language. Query language test case quality (e.g., SQL), on the other hand, is evaluated using mutation analysis, which is considered to be a stronger criterion for assessing quality. In this case, several mutants or variants of the original SQL query are generated and the quality is measured using a metric called mutation score. The score indicates the percentage of mutants that can be identified in terms of their results using the given test cases. Higher mutation score indicates higher quality for the test cases. In this thesis we present novel testing strategy which guides concolic testing using mutation analysis for test case (which includes both program input and synthetic data) generation for database applications. The novelty of this work is that it ensures that the test cases are of high quality not only in terms of coverage of code written in the host language, but also in terms of mutant detection of the queries written in the query language.

CHAPTER 1. SOFTWARE TESTING FOR DATABASE APPLICATIONS

1.1 Background

Database systems play a central role in the operations of almost every modern organization. Commercially available database management systems (DBMSs) provide organizations with efficient access to large amounts of data, while both protecting the integrity of the data and relieving the user of the need to understand the low-level details of the storage and retrieval mechanisms. To exploit this widely used technology, an organization will often purchase an off-the-shelf DBMS, and then design database schemas and application programs to fit its particular business needs. It is essential that these database systems function correctly and provide acceptable performance. The correctness of database systems have been the focus of extensive research. The correctness of business applications, though, depends as much on the database management system implementation as it does on the business logic of the application that queries and manipulates the database. While Database Management Systems are usually developed by major vendors with large software quality assurance processes, and can be assumed to operate correctly, one would like to achieve the same level of quality and reliability to the business critical applications that use them. Given the critical role these systems play in modern society, there is clearly a need for new approaches to assess the quality of the database application programs.

There are many aspects of the correctness of a database system, some of them are:

- Does the application program behave as specified?
- Does the database schema correctly reflect the organization of the real world data being modeled?

- Are security and privacy protected appropriately?
- Are the data in the database sufficient?

All of these aspects of database system correctness, along with various aspects of system performance, are vitally important to the organizations that depend on the database system.

Many testing techniques have been developed to help assure that application programs meet their specifications, but most of these have been targeted towards programs written in traditional imperative languages. New approaches, targeted specifically towards testing database applications, are needed for several reasons. A database application program can be viewed as an attempt to implement a function, just like programs developed using traditional paradigms. However, consider in this way, the input and output spaces include the database states as well as the explicit input and output parameters of the application. This has substantial impact on the notion of what a test case is, how to generate test cases, and how to check the results produced by running the test cases. Furthermore, database application programs are usually written in a semi-declarative language, such as SQL, or a combination of an imperative language (which determines the control flow of the application, we call them host language) and a declarative language (we call them embedded language) rather than using a purely imperative language. Most existing program-based software testing techniques are designed explicitly for imperative languages, and therefore are not directly applicable to the database application programs.

The usual technique of quality assurance is testing: run the program on many test inputs and check if the results conform to the program specifications (or pass programmer written assertions). The success of testing highly depends on the quality of the test inputs. A high quality test suite (that exercises most behaviors of the application under test) may be generated manually, by considering the specifications as well as the implementation, and directing test cases to exercise different program behaviors. Unfortunately, for many applications, manual and directed test generation is prohibitively expensive, and manual tests must be augmented with automatically generated tests. Automatic test generation has received a lot of research attention, and there are several algorithms and implementations that generate test suites. For example, white-box testing methods such as symbolic execution may be used to generate good quality test inputs. However, such test input generation techniques run into certain problems when dealing with database-driven programs. First, the test input generation algorithm has to treat the database as an external environment. This is because the behavior of the program depends not just on the inputs provided to the current run, but also on the set of records stored in the database. Therefore, if the test inputs do not provide suitable values for both the program inputs and the database state, the amount of test coverage obtained may be low. Second, database applications are multi-lingual: usually, an imperative program implements the application logic, and declarative SQL queries are used for retrieving data from database. Therefore, the test input generation algorithm must faithfully model the semantics of both languages and analyze the mixed code under that model to generate tests inputs. Such an analysis must cross the boundaries between the application and the database.

Mutation Testing (or Mutation Analysis) is a fault-based testing technique which [1] has been proven to be effective for assessing the quality of the generated test inputs. The history of Mutation Analysis can be traced back to 1971 in a student paper by Lipton [2]. The birth of the field can also be identified in papers published in the late 1970s by DeMillo et al. [3] and Hamlet [4]. In mutation testing, the original program is modified slightly based on typical programming errors. The modified version is referred to as the *mutant*. Mutation Analysis provides a criterion called the *mutation score*. The mutation score can be used to measure the effectiveness of a test set in terms of its ability to detect faults. The general principle underlying Mutation Analysis work is that the faults used by Mutation Analysis represent the mistakes that programmers often make. By carefully choosing the location and type of mutant, we can also simulate any test adequacy criteria. Such faults are deliberately seeded into the original program by simple syntactic changes to create a set of faulty programs called mutants, each containing a different syntactic change. To assess the quality of a given test set, these mutants are executed against the input test set. If the result of running a mutant is different from the result of running the original program for any test cases in the input test set, the seeded fault denoted by the mutant is detected. The outcome of the mutation testing process is measured using mutation score, which indicates the quality of the input test set. The mutation score is

Table 1.1 Mutation Operation Example

Actual Program p	Mutant p'
...	...
if(a==1 && b==1)	if(a==1 ||| b==1)
return 1;	return 1;
...	...

the ratio of the number of detected faults over the total number of the seeded faults.

In mutation analysis, from a program p, a set of faulty programs p', called mutants, is generated by a few single syntactic changes to the original program p. As an illustration, Table 1.1 shows the mutant p', generated by changing the *and* operator of the original program p, into the *or* operator, thereby producing the mutant p'. A transformation rule that generates a mutant from the original program is known as mutant operators. Table 1.1 contains only one mutant operator example; there are many others [3, 4, 1].

The traditional process of Mutation Analysis is to assess the quality of the test cases for a given program p, illustrated in Figure 1.1.

For a given program p, several mutants i.e. p's are created depending on predefined rules. In the next step, a test set T is supplied to the system. The program p and each mutant i.e. all p's are executed against T. If the result of running p' is different from the result of running p for any test case in T, then the mutant p' is said to be *killed*; otherwise, it is said to be *alive*.

After all test cases have been executed, there may still be a few surviving mutants. Then the metric *mutation score* is calculated. It is the *percentage* of number of mutants killed divided by total number non-equivalent mutants (mutants which are both syntactically and semantically different). If the mutation score value is above than predefined threshold (which may be 100%) then we can say the test case is good enough identifying programming faults. If not, surviving mutants can further be analyzed to improve the test set T. However, there are some mutants that can never be killed because they always produce the same output as the original program. These mutants are called Equivalent Mutants. They are syntactically different but semantically equivalent to the original program. Automatically detecting all equivalent mutants

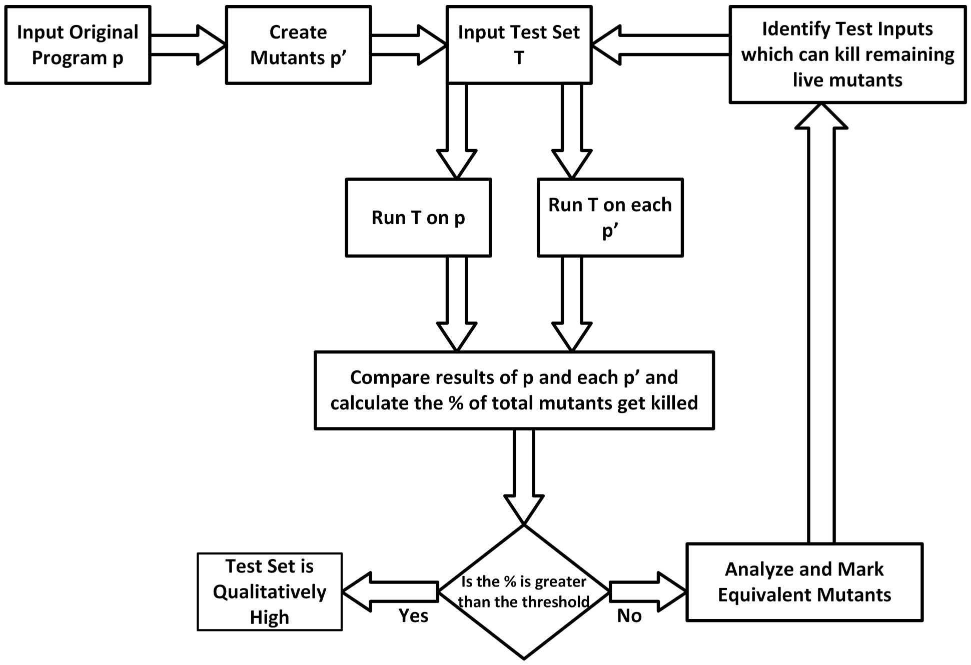
Input Original Program p
Create Mutants p'
Input Test Set T
Identify Test Inputs which can kill remaining live mutants
Run T on p
Run T on each p'
Compare results of p and each p' and calculate the % of total mutants get killed
Test Set is Qualitatively High
Yes
Is the % is greater than the threshold
No
Analyze and Mark Equivalent Mutants

Figure 1.1 General Control Flow for Assessing Test Input Quality using Mutation Analysis

is impossible [5] because program equivalence is undecidable.

Mutation Analysis can be used for testing software at the unit level, the integration level, and the specification level. It has been applied to many programming languages as a white box unit test technique, for example, Fortran programs, $C\#$ code, SQL code, AspectJ programs [6, 7, 8]. Mutation Testing has also been used for integration testing [9, 10, 11]. Besides using Mutation Testing at the software implementation level, it has also been applied at the design level to test the specifications or models of a program.

In database applications, mutation testing has been applied to SQL code to detect faults. The first attempt to design mutation operators for SQL was done by Chan et al. [12] in 2005. They proposed *seven* SQL mutation operators based on the enhanced entity-relationship model. Tuya et al. [8] proposed another set of mutant operators for SQL query statements. This set of mutation operators is organized into *four* categories: mutation of SQL clauses, mutation of operators in conditions and expressions, mutation handling NULL values, and mutation of identifiers. They also have developed a tool named SQLMutation [13] that implements this set of SQL mutation operators and have shown an empirical evaluation concerning results using SQLMutation [13]. A development of this work targeting Java database applications can be found in [14]. [15] has also proposed a set of mutation operators to handle the full set of SQL statements from connection to manipulation of the database. This paper introduced *nine* mutation operators and implemented them in an SQL mutation tool called *MUSIC*.

1.2 Driving Problem

With advances in the Internet technology and ubiquity of the web, applications relying on data/information processing and retrieval from database form the majority of the applications being developed and used in the software industry. Therefore, it is important that such applications are tested adequately before being deployed. There are two main approaches to generate test cases for database applications:(a) generating database states from scratch [16, 17, 18] and (*b*) using existing database states [19]. These approaches try to achieve a common goal, *high branch coverage*. Test cases achieving high block or branch coverage certainly increases the confidence on the quality of the application under test; however, coverage cannot be argued as a sole criterion for effective testing. Mutation testing [1] has been proven effective for assessing the quality of the generated test inputs.

Typically, test case generation for database applications include both program inputs and synthetic data (if required) generation to ensure a high degree of (code, block or branch) coverage. Mutation testing is performed separately for analyzing quality of the generated test cases in terms of identifying SQL related faults. If the mutation score of the generated test cases is low, new test cases are generated and mutation analysis is performed again. This results in unnecessary delay and overhead in identifying the high quality test cases, where quality is attributed to both coverage and mutation scores.

Figure 1.2 demonstrates the broader problem scenario in the field of database application testing. In one hand researchers have developed automated test data generation techniques for database application programs which generate test cases automatically for the application. These techniques guarantee to achieve high structural coverage of the given program but may suffer from low quality in terms of identifying SQL faults that might present in the embedded

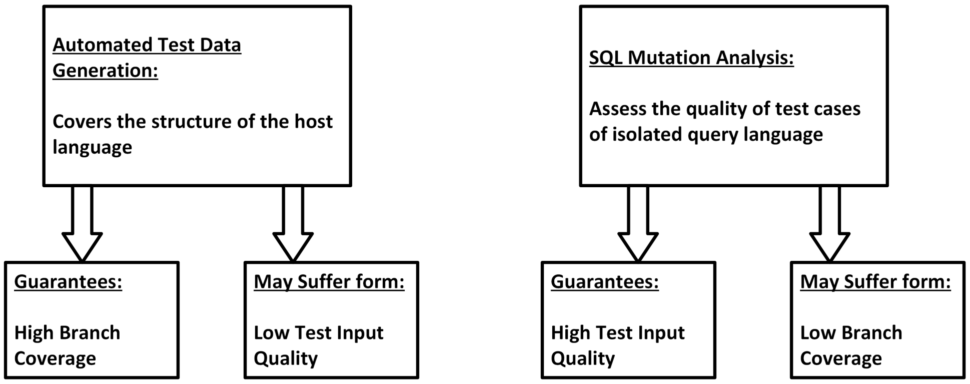

Figure 1.2 Broader Problem Scenario in the field of Database Application Testing

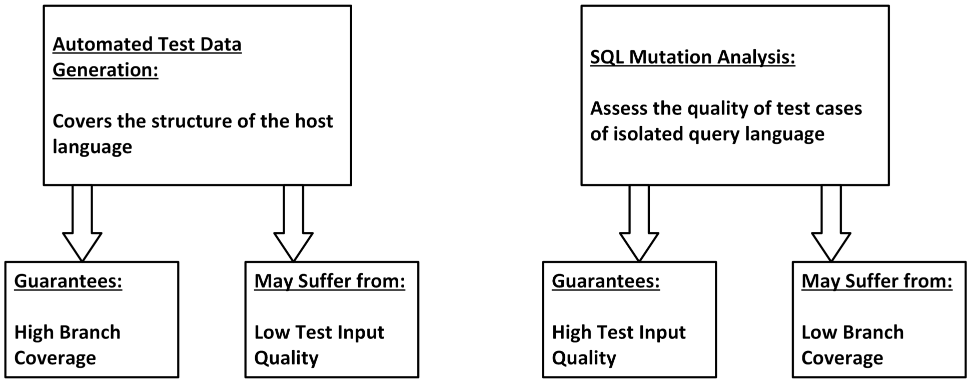

Figure 1.3 Our Solution Approach

query. On the other hand researchers have developed SQL Mutation Analysis to assess quality of the test inputs for isolated SQL queries. These techniques surely identifies test cases with high mutation score for individual SQL queries, but they might not guarantee to achieve high structural coverage for the host language (imperative language) in which the embedded queries are used.

1.3 Our Solution

Figure 1.3 demonstrates our overall solution to address the broader problem scenario as shown in Figure 1.2. We combine *coverage analysis* and *SQL mutation analyis* to generate test cases which include both program inputs and synthetic data for database applications. The generated test cases will guarantee two things,

- High structural coverage, and
- High mutation score.

In this work, we propose and develop a constraint-based test case generation technique for database applications achieving both high structural coverage and high mutation score. The approach works as follows. First our technique tries to cover possible branches of the given program. If not, then synthetic data will be created to improve coverage. After covering a new branch of a program, for every newly generated test case, we measure the mutation score of the test case. If the mutation score of the test case is below the pre-specified threshold, our technique analyzes the path constraints (necessary for coverage) and mutant-killing constraints (necessary for high mutation score), and uses a constraint solver to automatically identify a new test case whose quality is likely to be high. If no new test case can improve the mutation score with respect to the present database state (including the generated synthetic data), a new constraint will be generated to update database state (i.e. identify new synthetic data). If the constraint is solvable by the solver, new synthetic data will be created for the database. With respect to the new database state (which includes newly generated data), previously generated test case can achieve high quality in terms identifying SQL faults. Finally, the whole process is iterated to generate new test cases that explore new execution paths of the program. This iteration continues until all possible branches are covered.

Apart from improving test input quality, we also address an important common challenge in the field of database application testing. Most of the existing test strategies do not consider the relationship between the application program and the current database state while generating program inputs for the application. This leads to unnecessary database state generation for improving test input quality. We leverage our basic solution and propose a new approach for generating test cases for database applications. The novelty of the new technique is it maximizes the usage of current database state to identify program inputs achieving both high coverage and high mutation score. This will eliminate the overhead of generating unnecessary synthetic data at each iteration. Therefore only minimal set of synthetic data will be generated to help test cases achieve high quality both in terms of structural coverage and SQL mutation score.

1.4 Overall Contributions

The contributions of our work are summarized as follows:

1. To the best of our knowledge, this is the *first approach* that combines coverage analysis and mutation analysis in automatic test case generation for database applications which involve two different languages: host language and embedded query language.

2. The impact of our proposed technique is that it reduces the overhead of high quality test case generation by avoiding test cases with low coverage and low mutation scores.

3. Synthetic data generation strategy in database application testing not only helps improve coverage but also mutation score.

4. We propose a new approach to avoid unnecessary generation of synthetic data and maximizes the usage of current database state during test case generation. Thus only minimal set of data will be generated.

5. We evaluate the practical feasibility and effectiveness of our proposed framework by applying it on two real database applications. We compare our method against other existing tools like Pex [20], a white-box testing tool for .NET from Microsoft Research, SynDB [17], an automated test case generation approach for database applications developed on top of Dynamic Symbolic Execution technique, Tool developed by Emmi et al. [16], an automated test case generation approach for database application developed on top of Concolic Execution technique, and show that applications. We compare our method against other existing tools like Pex [20], a white-box testing tool for .NET from Microsoft Research, SynDB [17], an automated test case generation approach for database applications developed on top of Dynamic Symbolic Execution technique, Tool developed by Emmi et al. [16], an automated test case ganeration approach for database application

devloped on top of Concolic Execution technique, and show that our method generates test cases with higher code coverage and higher mutation score compared to the ones generated by aforementioned existing approaches.

1.5 Organization

The rest of the thesis is organized as follows. Chapter 2 discusses about several aspects of Software Testing followed by existing works done in the field of Automated Test Case Generation, Mutation Testing and Testing Database Applications which are related to this work. In chapter 3, we discuss about our novel framework which combines coverage analysis and mutation analysis to generate high quality test cases. We also demonstrate the validation of our approach by providing experimental results. In chapter 4 we leverage our work from chapter 3 and propose a technique to generate test cases even when associated database state insufficient or absent. This approach also generates database states (we call them synthetic data), if required, so that generated test cases can achieve high quality with respect to the generated data. We also provide experimental results in this chapter to validate our approach. In chapter 5, we have proposed a new helper method which will leverage our overall testing strategy to reuse the current database state to the fullest. This approach will help our overall technique to reduce the overhead of generating unnecessary synthetic data while generating high quality test cases both in terms of coverage and mutation score. Chapter 6 summarizes our work proposed in chapters 3, 4 and 5. We conclude the chapter by demonstrating the overall unique impact of this work in the field of Software Testing and propose a brief picture of extending our current work to solve other problem scenario in the field of software testing.

CHAPTER 2. RELATED WORK

Substantial research in systematic design and development practices increasingly improves in building reliable software; errors are still present in the software. The aim of software testing is to expose/identify such bugs/errors by executing the software on a set of test cases. In the basic form, a test case will consist of program inputs and corresponding expected outputs. After the software has successfully passed the testing phase, we have a greater degree of confidence in the software (in terms of reliability). Typically Software Testing is labor intensive, therefore also expensive. Research has shown that testing can account for 50% of the total cost of software development [21]. Therefore a need for automated testing strategies caught attention of the researchers. Tools which can automate one or several aspects of testing can hugely help in reducing the overall cost of testing. There are different aspects/directions of testing techniques, but broadly categorized into two categories, *functional testing* and *structural testing.* Functional testing is mainly used to verify the functionality of the program, implementation of the program is not important in this case. Therefore it involves comparing multiple input output conditions. On the other hand, structural testing is concerned with testing the implementation of the program. The primary focus of our work is structural testing. Before we move into more details, we will discuss some key aspects of structural testing.

Test requirements:

As a first step, specific program identities need to be found in terms of which the *test requirements* of a given program can be identified. So, when the program is executed on a test case, program execution can be analyzed to determine the set of test requirements that are exercised by the test case. Some example test requirements are: program paths, program statements etc.

Test Coverage Criteria:

In order to determine the completeness of the testing process, a term called *test coverage criteria* is defined. A test coverage criterion specifies a minimal set of test requirements that must be achieved by the test cases on which the program is executed during the testing process. Minimal set of test requirements depends on the criterion. It is also helpful to guide the testing process. At any point during testing, a typical goal is to run the program on test cases that cover the test requirements that yet to be covered by any of test cases that have already been executed.

Testing Strategy:

Typically, first unit testing is employed to test all the modules in the program individually and then integration testing is performed to test the interfaces among the modules. Different ways are used to organize integration testing process. Initially the program is fully tested. During maintenance stages, regression testing technique is employed. This technique monitors only the test requirements that are impacted by the program changes. Finally the generation of test cases based on the test requirements can be done manually or automatically. Concolic Testing Strategy, Dynamic Symbolic Execution technique are some of the popular automated test case generation techniques.
Structural Testing strategy can be categorized into three main parts: *control flow based testing*, *data flow based testing* and *mutation testing*. In control flow based testing, test coverage is criteria is measured in terms of nodes, edges, paths in the program control flow graph. In Data flow based testing: test coverage is measured in terms of definition-use associations present in the program. In Mutation testing, numbers of variants are created from the original program using some predefined rules. The variants are called mutants. The goal of mutation testing is to identify test cases that distinguish original program from its mutants. In our work, we will mainly discuss about control flow based testing and mutation testing.

2.1 Automated Test Case Generation

Automating test case generation is an active area of research. In the last several years, over a dozen of techniques have been proposed that automatically increase test coverage or generate test inputs. Among them, random generation of test cases (concrete values) have been proven to be the simplest and very effective [22, 23, 24, 25]. It could actually be used to generate input values for any type of program since, ultimately, a data type such as integer, string, or heap is just a stream of bits. Thus, for a function taking a string as an argument, we can just randomly generate a bit stream and let it represent the string. On the contrary, random testing mostly does not perform well in terms of coverage. Since it merely relies on probability, it has quite low chances to identify semantically small faults [23], and thus accomplish high coverage. A semantically small fault is such a fault that is only revealed by a small percentage of the program input. Consider the code in Figure 2.1,

```
void test1(int i, int j){
  if(i == j)
   Method1();
else
   Method2();
}
```

Figure 2.1 Sample Code Fragment

The probability of reaching *Method1()* statement is $1/n$, where n is the maximum integer value, since in order to execute the statement, i and j must be equal. This tells us the fact that generating even more complex structures than simple integer equalities will give even worse probability.

The goal-oriented approach is much stronger than random generation, provides a guidance towards a certain set of paths. Instead of letting the generator generate input that traverses from the entry to the exit of a program, it generates input that traverses a given path. Because of this, it is sufficient to find input for any path. This in turn reduces the risk of encountering relatively infeasible paths and provides a way to direct the search for input values as well. Two methods using this technique have been found: *the chaining approach* and *assertion-oriented approach*. The latter is an interesting extension of the chaining approach. They have all been implemented in the TESTGEN system [26, 27].

Path-oriented generation is strongest among the three approaches. It does not provide the generator with a possibility of selecting among a set of paths, but just one specific. In this way it is the same as a goal-oriented test data generation, except for the use of specific paths. Successively this leads to a better prediction of coverage. On the contrary, sometimes it is harder to find test data. Substantial amount of research works have been done in all these areas. We are specifically interested in path-oriented test data generation. Rest of this section talks about related works in path-oriented test data generation.

An important technique to mention is bounded-exhaustive concrete execution [28, 29] that tries all values from user-provided domains to cover the paths of the program. Even though these tools can achieve high code coverage, but they require the user to carefully choose the values in the domains to ensure high coverage. Microsoft Research has also developed a white box testing tool named Pex [20] based on Dynamic Symbolic Execution technique, performs path-bounded model-checking of .NET programs. Pex search strategies try to find individual execution paths in a sequence which depends on chosen heuristics; the strategies are complete and will eventually exercise all execution paths. This is important in an environment such as .NET where the program can load new code dynamically, and not all branches and assertions are known ahead of time. The core of Dynamic Symbolic Execution strategy is same as Concolic Execution, but Pex has its added advantages. Pex is language independent, and it can symbolically reason about pointer arithmetic as well as constraints from object oriented programs. Pex search strategies aim at achieving high coverage fast without much user annotations. Other notable tools are Randoop [30] and Agitar [31]. Randoop generates new test cases by composing previously found test case fragments, supplying random input data. Agitar generates test cases for Java by analyzing the source code, using information about program invariants.

Another popular technique is symbolic Execution. It uses variety of approaches like abstraction based model checking [32], explicit state model checking [33], symbolic sequence exploration [34], and static analysis [35]. Essentially, all these techniques either try to detect potential bugs or test inputs. They inherit the incompleteness from their underlying reasoning engines like theorem provers and constraint solvers. For example, tools using precise symbolic execution [33, 34] cannot analyze any code with non-linear arithmetic or array indexing with non-constant expressions. Typically in these tools, symbolic execution proceeds separately from the concrete execution. Techniques like CUTE [36], DART [37] combines concrete and symbolic execution. Even though the core techniques are same, some improvements are seen in CUTE. As an example, DART tests each function in isolation and without preconditions, whereas CUTE targets related functions with preconditions such as data structure implementations. DART handles constraints only on integer types and cannot handle programs with pointers and data structures, whereas CUTE handles such scenarios.

2.2 Mutation Testing

Since Mutation Testing was proposed in the 1970s, it has been applied to test both program source code (Program Mutation) [38] and program specification (Specification Mutation) [39]. Program Mutation belongs to the category of white-box-based testing, in which faults are seeded into source code, while Specification Mutation belongs to blackbox-based testing, where faults are seeded into program specifications, but in which the source code may be unavailable during testing. There has been more work on Program Mutation than Specification Mutation. Notably more than 50% of the work has been applied to Java [40, 41], Fortran [6, 42] and C [43, 44]. Fortran features highly because a lot of the earlier work on Mutation Testing was carried out on Fortran programs.

Program based mutation testing consists of generating a large number of alternative programs called mutants, each one having a simple fault that consists of a single syntactic change in the original program. Mutants are created by transforming the source code using a set of defined rules (mutation operators) that are developed to induce simple syntax changes based on errors that programmers typically make. Each mutant is executed with the test data and when it produces an incorrect output (the output is different to that of the original program), the mutant is said to be killed. A test case is said to be effective if it kills some mutants that have not yet been killed by any of the previously executed test cases. Some mutants always produce the same output as the original program, so no test case can kill them. These mutants are said to be equivalent mutants. After executing a test set over a number of mutants, the mutation score is defined as the percentage of dead mutants divided by the number of non-equivalent mutants. A study has shown mutation testing to be superior to common code coverage in evaluating effectiveness of test inputs [19].

Program Mutation has been applied to both the unit level [45] and the integration level [10] of testing. For unit-level Program Mutation, mutants are generated to represent the faults that programmers might have made within a software unit, while for the integration-level Program Mutation, mutants are designed to represent the integration faults caused by the connection or interaction between software units. Applying Program Mutation at the integration level is also known as Interface Mutation, which was first introduced by Delamaro et al. [10]. Interface Mutation has been applied to C programs by [11, 10] and also to CORBA programs by [46, 47]. Empirical evaluations of Interface Mutation can be found in [48, 49]

Mutation testing has been further extend to programming languages like $C\#$ [7, 50], SQL [8, 51, 14, 13, 15]. The primary goal of developing SQL mutation operators is to measure the quality of the generated test inputs and generate quality test inputs for isolated SQL queries. But in database application, SQL query is embedded as a string inside the host language. Therefore measuring the quality of test inputs and generating high quality test inputs for database application involves including mutation analysis of embedded SQL queries, which has not been done before. Our work in this thesis combines the mutation analysis technique as a quality measurement guidance criterion for automated test generation technique. Therefore newly generated test inputs will achieve both high coverage and high SQL mutation score.

Although Mutation Testing was originally proposed as a white box testing technique at the implementation level, it has also been applied at the software design level. Mutation Testing at design level is often referred to as *Specification Mutation* which was first introduced by Gopal and Budd [39] In Specification Mutation, faults are typically seeded into a state machine or logic expressions to generate *specification mutants*. A specification mutant is said to be killed if its output condition is falsified. Specification Mutation can be used to find faults related to missing functions in the implementation or specification misinterpretation.

2.3 Database Application Testing

Database application programs play a central role in operation of almost every modern organization. Recently, database application testing [52, 18, 53, 16] has attracted much attention of researchers. All these approaches try to achieve a common goal, *high branch coverage*. Therefore automatic generation of test inputs has been regarded as the main issue in database application testing. Along with high branch coverage, assessing the goodness of test data has not been considered as a criterion while generating test inputs. Mutation testing has been proven to be a powerful method in this regard. For database application, SQL mutation operators have been developed [8, 13] and then coverage criteria of isolated SQL statements [51] have been defined separately. Our work [54] combines the coverage criteria and mutation analysis in such a way that test cases with high coverage and high mutation score are generated automatically. The primary challenge addressed in our work is the consideration of database applications where the coverage criteria depends on the application language while the mutation score relies only on the embedded query language.

Test input generation for database applications primarily depends on the current database state. Before generating test inputs for database application, testers need to generate sufficient number of entries for the tables present in the database. Therefore, generating test database in an optimized/sufficient manner for a given application is a challenging problem which has concentrated some research efforts [55, 56]. A tool [57] has been defined for data generation incorporating Alloy specifications both for the schema and the query. Each table is modeled as a separate n-arity relation over the attribute domains and the query is specified as a set of constraints that models the condition in the WHERE clause. However, this approach cannot handle tables with a larger number of attributes due to the arity of the table relations.[58] propose a technique named *reverse query processing* for generating test databases that takes the query and the desired output as input and generates a database instance (using a model checker) that could produce that output for the query. This approach supports one SQL query and therefore generates one test database for each query. A further extension to this work [58] supports a set of queries and allows to specify to the user the output constraints in the form of SQL queries. However, the creation of these constraints could be difficult if the source specification is not complete. There are other works which use general purpose constraint solvers to populate the test database [59, 16, 60]. As in preceding works, the coverage criterion for generating the test database is not specifically tailored for SQL queries but rather for predicates or user constraints and therefore, the generated test database does not provide enough confidence to exercise the target query and also the corresponding database application from a testing point of view.

In recent works [55, 52], researchers use coverage criteria in conjunction with database constraints to populate databases with test data. But the main disadvantage of these approaches is *considering isolated SQL statements*. Therefore, while executing the actual query or it's mutants from a particular database application, the test database might not find any result. In this thesis, we leverage our basic approach [54] and develop a new approach [61] which automatically generates test cases and synthetic data if required. The generated data, in our approach, will help to improve both structural coverage and mutation score of the generated test cases.

As an extension of our work we exploit the existing database state and generate test cases covering maximum number of branches. This technique is also accompanied by our mutation analysis so that only high quality test inputs are generated. By using this approach we are able to bypass unnecessary mock data generation and reduce overhead. If only our approach cannot find any test case using existing database entries to cover a particular branch of a given program, our framework will guide the tester to generate mock dataset which will help to generate test data covering the uncovered branch. The mock dataset will be selected in such a way that the test data can also achieve high mutation score along with high branch coverage.

CHAPTER 3. ConSMutate: SQL MUTANTS FOR GUIDING CONCOLIC TESTING OF DATABASE APPLICATIONS

3.1 Introduction

3.1.1 Driving Problem

With advances in the Internet technology and ubiquity of the Web, applications relying on data/information processing and retrieval from database form the majority of the applications being developed and used in the Software industry. Therefore, it is important that such applications are tested adequately before being deployed. A typical database application consists of two different programming language constructs: the control flow of the application depends on procedural languages, *host language* (e.g., Java); while the interaction between the application and the backend database depends on specialized *query languages* (e.q., SQL) that are constructed and embedded inside the host language. Automatically generating test cases along with assessing their quality, therefore, pose an interesting and important challenge for such applications.

3.1.2 Motivating Example

Consider the pseudo code in the above procedure CHOOSECOFFEE. It represents a typical database application; it takes as two input parameters x and y, creating different query string depending on the valuation of the parameters which guides the control path in the application. Assume that one of the database tables `coffees` contains three entries as shown in Table 3.1. Pex generates three test cases, e.g., (0, 0), (11, 0) and (11, 2), taking into consideration the branch conditions in the application program. The first and the second values in the tuple represent the valuations of x and of y respectively. These test cases cover all branches present

```
1:  procedure CHOOSECOFFEE($x$, $y$)
2:      String q = " ";
3:      if x>10 then
4:          y++;
5:          if y$\leq$ 2 then
6:              q = "SELECT cof_name FROM coffees WHERE price =" + y + ";";
7:          else
8:              q = "SELECT cof_name FROM coffees WHERE price $\leq$" + y + ";";
9:          end if
10:     end if
11:     if q != " " then
12:         executeQuery(q);
13:     end if
14:     return;
15: end procedure
```

Algorithm 1 **Sample Pseudo code for Database Application**

Table 3.1 Table **coffees**

cof_name	sup_id	price
Colombian	101	1
French_Roast	49	2
Espresso	150	10

in the program. However, as the database is not taken into consideration for the test case generation, the test cases are unlikely to kill all mutants corresponding to the query being executed. For instance, the test case (11, 0) results in the execution of the query generated at Line 6.

The executed query

$$\text{SELECT cof\_name FROM coffees WHERE price} = 1$$

generates the result `Colombian` using the *coffees* table. A mutant of this query

$$\text{SELECT cof\_name FROM coffees WHERE price} \leq 1$$

is generated by slightly modifying the "WHERE" condition in the query (mimicking typical programming errors). The result of the mutant is also `Colombian`. That is, if the programmer makes the error of using the equal-to-operator in the "WHERE" condition instead of the intended less-than-equal-to operator, then that error will go un-noticed if test case $(11, 0)$ is used. Note that there exists a test case $(11, 1)$ which can distinguish both the mutants from the original query without compromising branch coverage. We will show in section 3.2 that our framework successfully identifies such test cases automatically.

3.1.3 Problem Statement

How to automatically generate test cases for database applications such that: *test cases not only ensure high coverage of the control flow described in host language, but also allow for adequate testing of the embedded queries by attaining high mutation scores where mutants are generated from embedded queries*?

3.1.4 Individual Contributions

The contributions of our work described in this chapter are summarized as follows:

1. To the best of our knowledge, this is the first approach that combines coverage analysis and mutation analysis in automatic test case generation for database applications which involve two different language interaction.

2. The impact of our proposed framework is that it reduces the overhead of high quality test case generation by avoiding test cases with low coverage and low mutation scores.

3. We evaluate the practical feasibility and effectiveness of our proposed framework by applying it on two real database applications. We compare our method against Pex [20], a white-box testing tool for .NET from Microsoft Research, and show that our method generates test cases with higher code coverage and higher mutation score compared to the ones generated by Pex.

3.2 ConSMutate Test Case Generator for DB-Applications

Figure 3.1 presents the overall architecture of our framework named *ConSMutate*. It has two main modules, *Application Branch Analyzer* and *Mutation Analyzer*. The Application Branch

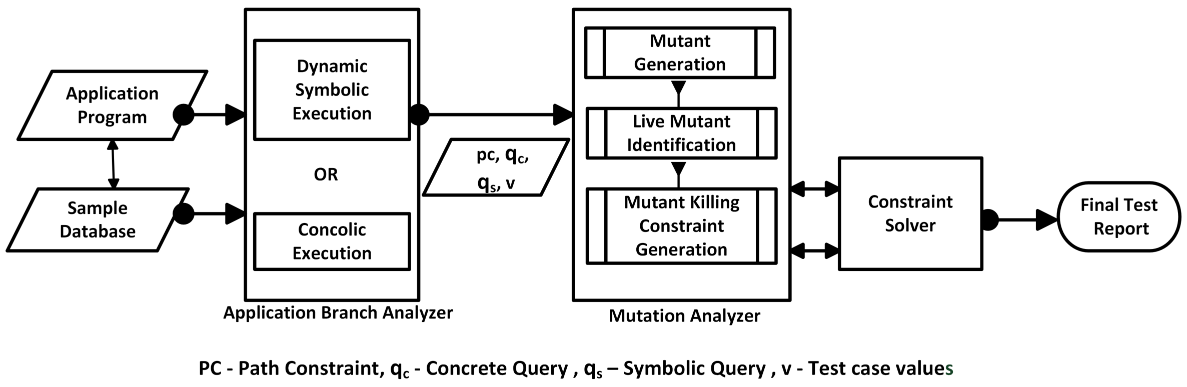
Application Program
Sample Database
Dynamic Symbolic Execution
OR
Concolic Execution
Application Branch Analyzer
pc, q_c, q_s, v
Mutant Generation
Live Mutant Identification
Mutant Killing Constraint Generation
Mutation Analyzer
Constraint Solver
Final Test Report
PC - Path Constraint, q_c - Concrete Query , q_s – Symbolic Query , v - Test case values

Figure 3.1 Framework for ConSMutate

Analyzer takes the program under test and the sample database as inputs, and generates test cases and the corresponding path constraints. It uses Pex [20], a dynamic symbolic execution engine (other engines like concolic testing tools [36] can also be used), to generate test cases by carefully comparing the concrete and symbolic execution of the program. After exploring each path, the Mutation Analyzer module performs mutation quality analysis using mutation score. If the mutation score is low, Mutation Analyzer generates a new test case for the same path whose *quality* is likely to be high. The steps followed in our framework for generating test cases are presented in the following subsections.

3.2.1 Generation of Test Cases and Associated Path Constraints Using Application Branch Analyzer

In the first step, the framework uses the *Application Branch Analyzer* module to generate a test case value v and the associated path constraints. It results in a specific execution path constraint (say, PC) of the application, which in turn results in a database query execution (if the path includes some query). The executed query is referred to as the concrete query q_c and the same without the concrete values (with the symbolic state of the input variable) is referred to as the symbolic query q_s. The path constraints refer to the conditions which must be satisfied for exploring the execution path in the application.

Going back to the example in Section 3.1.2, Application Branch Analyzer (Pex in our case) generates a test case $v = (11, 0)$, i.e., $x = 11$ and $y = 0$. This results in an execution path with path constraints $PC = (x > 10) \wedge (y + 1 \leq 2)$. It also results in a symbolic query and a corresponding concrete query:

Symbolic q_s: SELECT cof_name FROM coffees WHERE price = y_s.

Concrete q_c: SELECT cof_name FROM coffees WHERE price = 1.

where y_s is related to program input y as $y_s = y + 1$ at line 6 (see the example program in Section 3.1.2).

3.2.2 Deployment of Mutation Analyzer

After exploring a path of the program under test, ConSMutate forwards PC, q_c, q_s and v to *Mutation Analyzer* to evaluate the quality of the generated test case in terms of mutation score.

3.2.2.1 Generation of Mutant Queries

In Mutation Analyzer, the obtained concrete query q_c is mutated to generate several mutants q_m(s). The mutations are done using pre-specified mutation functions in the *Mutant Generation* module.

It is generally agreed upon that a large set of mutation operators may generate too many mutants which, in turn, exhaust time or space resources without offering substantial benefits. Offutt et al. [62] proposed a subset of mutation operators which are approximately as effective as all 22 mutation operators of Mothra, a mutation testing tool [38]. They are referred as *sufficient set of mutation operators*. In our context, we are specifically focused on SQL mutants. We have identified *five* mutation operators by comparing SQL mutation operators developed in [8] with the sufficient set of mutation operators mentioned in [62]. We refer to these six rules as the *sufficient set of SQL mutation operators*, sufficient to identify logical errors present in the `WHERE` and `HAVING` clauses.

ConSMutate uses these mutation operators in generating mutants. It should be noted here that new mutation operators can be considered and incorporated in mutation generation

Table 3.2 Sample mutant generation rules and mutant killing-constraints

Mutation Rule	**Original**	**Mutant**	**Mutant Killing-constraint**
Relational Operator Replacement (ROR),$\alpha, \beta \in$ ROR and $\alpha \neq \beta$	$C_1\ \alpha\ C_2$	$C_1\ \beta\ C_2$	$((C_1\ \alpha\ C_2) \wedge \neg(C_1\ \beta\ C_2))$ $\|$ $(\neg(C_1\ \alpha\ C_2) \wedge (C_1\ \beta\ C_2)))$
Logical Operator Replacement (LOR),$\alpha, \beta \in$ LOR and $\alpha \neq \beta$	$C_1\ \alpha\ C_2$	$C_1\ \beta\ C_2$	$(C_1\ \alpha\ C_2) \neq (C_1\ \beta\ C_2)$
Arithmetic Operator Replacement (AOR),$\alpha, \beta \in$ AOR and $\alpha \neq \beta$	$C_1\ \alpha\ C_2$	$C_1\ \beta\ C_2$	$(C_1\ \alpha\ C_2) \neq (C_1\ \beta\ C_2)$
Unary Operator Insertion (UOR), $\forall$u $\in$ UOI	C_1	u(C_1)	$C_1 \neq$ u(C_1)
Absolute Value Insertion (ABS), $\forall$u $\in$ ABS	C_1	u(C_1)	$C_1 \neq$ u(C_1)

module in ConSMutate as and when needed. Table 3.2 (first three columns) presents those mutation generation rules. Going back to the example in section 3.1.2 one of the mutants of the symbolic q_s is

$$q_m\text{: SELECT cof\_name FROM coffees WHERE price} \leq y_s.$$

In the above transformation, α is "=" (equality relational operator) and β is "$\leq$" (less-than-equal-to relational operator) as per the rule in the first row, second and third columns of Table 3.2.

3.2.2.2 Identification of Live Mutants

Using the test case under consideration, the live mutants are identified. Live mutants are the ones whose results do not differ from that of the concrete query in the context of the given database table. The above mutant q_m is live under the test case $v = (11, 0)$ as it results in a concrete query

$$\text{SELECT cof\_name FROM coffees WHERE price} \leq 1.$$

Recall that $y_s = y + 1$ and $y = 0$ for the test case $(11, 0)$ when the query is constructed in Line 6 (see program in Section 3.1.2). The above query and the concrete query q_c produce the same result for the given database table (Table 3.1). Therefore, q_m is live under the test case $(11, 0)$.

3.2.2.3 Generation of Mutant Killing Constraints

A new set of constraints θ is generated in *Mutant Killing Constraint Generation* module in two steps:

1. Generation of constraint from queries,
    - the symbolic query q_s and its concrete version q_c,
    - the live mutants (q_m's) computed in the previous step,
    - the concrete and symbolic state of the program inputs which is affected by the test cases.
2. Incorporation of path constraints (PC) to ensure the same path is explored and therefore the same set of queries are executed.

Generation of Constraint from Queries. We proceed by capturing the concrete and symbolic queries executed in the path explored by the given test case. This is done using Pex API methods `PexSymbolicValue.ToString(..)` and `GetRelevantInputNames(..)`. We decompose concrete and symbolic query using a simplified SQL parser and get their `WHERE` conditions, which we assume to be in conjunctive normal form.

Identification of Query Conditions. We then identify the conditions that resulted in a mutant query and their relationship with the test inputs (or program inputs). We refer to the conditions obtained from the original query as the *original query-condition* and, likewise, the conditions obtained from the mutant query as the *mutant query-condition.*

For the *concrete versions* of the original and the mutant query-condition, we identify the satisfiable valuations of the database attribute. For instance, in our running example, the original query-condition is $price = 1$ and the mutant query-condition is $price \leq 1$. We query the database to find one valuation of *price* which satisfies these conditions. Note that the same valuation of *price* will satisfy both the conditions as we are considering the live mutants. In our running example, the original query-condition and the mutant query-condition are satisfied when the value of *price* is set to 1 (see Table 3.1).

Using the above and the *symbolic versions* of the original and the mutant query-conditions, we identify the relationship between the valuations of database attributes and the test inputs. For instance, in our running example, the original symbolic query-condition is $price = y_s$ and the mutant symbolic query-condition is $price \leq y_s$. We also know that y_s is set to $y + 1$ (y is one of the test inputs) and *price* is set to 1. Therefore, the relationship between the valuations of the database attribute *price* and the test input y is $1 = y + 1$ in the original query-condition and $1 \leq y + 1$ in the mutant query-condition. We will use these relationships/conditions for generating the mutant killing constraint; we refer to them as the *original input-condition* and the *mutant-input condition.*

Identification of Mutation Points. The original and the mutant input-conditions are compared to identify the mutation point (the point at which the original input-condition and the mutant input-condition differ). Depending on the mutation point, a corresponding mutant killing constraint rule is triggered.

For complex conditions, ConSMutate uses a binary search algorithm to identify the mutation point. As an example, the original condition $(C_1 \leq C_2) \wedge (C_3 \leq C_4)$ can have a mutant $(C_1{=}C_2) \wedge (C_3 \leq C_4)$. ConSMutate first looks at the outmost level and finds that the logical operators remain the same for both of these expressions. It recursively looks at the left and right sub-conditions of these expressions and identifies the mutation point. In this case the mutation point is at left-hand side i.e., $(C_1 \leq C_2)$ and $(C_1{=}C_2)$.

Identifyication of Mutant Killing Constraints for Conditions: Finally, for the original input-condition and its mutant, a mutant killing constraint is generated following the rules in Table 3.2 (4^{th} column). Satisfaction of the mutant killing constraint results in an assignment to the test inputs which satisfies (resp. does not satisfy) the original input-condition and does not satisfy (resp. satisfies) the mutant input-condition. For instance, for our running example, the mutant killing constraint is $[(1 = y + 1) \wedge (1 \not\leq y + 1)] \vee [(1 \neq y + 1) \wedge (1 \leq y + 1)]$ (using ROR rule from Table 3.2).

Incorporation of Path Constraints. We extract *path constraints* (PC) from Pex and conjunct them with the mutant killing constraint generated above to construct θ. This is necessary to ensure that any satisfiable assignment of test inputs results in exploration of the same execution path. In our running example, the path constraint is $(x > 10) \wedge (y + 1 \leq 2)$. The conjunction result will be θ as shown below.

$$\theta : (x > 10) \wedge (y + 1 \leq 2) \wedge$$
$$[((1 = y + 1) \wedge (1 \nleq y + 1)) \vee ((1 \neq y + 1) \wedge (1 \leq y + 1))]$$

3.2.3 Deployment of Constraint Solver: Finding Satisfiable Assignment for θ

The constraint θ is checked for satisfiability to generate a new test case. We use the SMT solver named Yices[1] for this purpose. Other high performance constraint solvers like Z3[2] can be used in the constraint solver module (Figure 3.1). If θ is satisfied, then certain valuations of the inputs to the application are identified, which is the new test case v'. This new test case v' is guaranteed to explore the same execution path as explored due to test case v. Furthermore, some mutants that were left "live" by v are *likely to be* "killed" by v'. Therefore, it is necessary to check whether v' indeed kills the live mutants; if not, SMT solver is used again to generate a new satisfiable assignment for θ (including the negation of the previously generated value), which results in a new test case v''. This iteration is terminated after certain pre-specified times (e.g., 10) or after all live mutants are killed (whichever happens earlier). It should be noted that if the live mutant is equivalent to the original query in the context of the database table, then no new test case can differentiate between the mutant and the original query. Therefore, we use a pre-specified limit to the number of iterations after which we terminate the process.

Going back to our running example, when the SMT solver generates a satisfiable assignment $x = 11, y = 1$ for the mutant killing constraint θ (see above), the new test case $v' = (11, 1)$ successfully kills the live mutant q_m by distinguishing its result from the original query result, as shown in Table 3.3.

The above steps (starting from Section 3.2.1) are iterated to generate new test cases that

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

Table 3.3 Mutants and results for test case (11, 1)

Query	**Concrete Query**	**Result**
q_c	SELECT cof_name FROM coffees WHERE price = 2	French_Roast
q_m	SELECT cof_name FROM coffees WHERE price $\leq$ 2	Colombian, French_Roast

explore different execution paths of the program. This iteration continues until all possible branches are covered following the method used by Pex.

3.2.4 Correctness Criteria of ConSMutate

For any path explored by a test case t_0 with path constraint PC, if the symbolic query executed along the path is q_s and if the live mutant is q_m, the set of satisfiable assignments for the mutant killing constraint θ as obtained by ConSMutate is a superset of the test cases that can kill the mutant.

Proof. A test case can be viewed as a mapping of variables (inputs to programs) to values. We will denote this mapping as $t : [\bar{x} \mapsto \bar{v}]$, where t is a test case, $\bar{x}$ is an ordered set of inputs/variables and $\bar{v}$ is an ordered set of valuations[3].

We prove the above theorem by contradiction. We assume that there exists a test case t that can kill the mutant q_m; however it is not a satisfiable assignment for θ, denoted by $t \not\models \theta$.

As the test case t can kill the mutant q_m, it must satisfy the path constraint PC, which is necessary to explore the path where the original query q_s is generated and executed. Recall that θ contains a conjunct PC. Therefore, $t \not\models \theta_1$, where $\theta = PC \ \wedge \ \theta_1$.

Next, let us consider the construction of θ_1. WLOG, consider that there is one mutation point in the `WHERE` clause of q_s and q_m. Let the `WHERE` clause be $db_{var} \ \mathcal{R} \ x$, where db_{var} is a database variable, $\mathcal{R}$ is a relational operator and x is an input to the program (x can be a program variable dependent indirectly on the program input). Let the mutant q_m has the `WHERE` clause transformed by altering $\mathcal{R}$ to $\mathcal{R}'$. The original test case t_0 results in the valuation

[3]When the ordered set contains one variable, we denote test case t as $t : [x \mapsto v]$

of x for which the `WHERE` clauses of q_s and q_m, i.e., $db_{var}\ \mathcal{R}\ x$ and $db_{var}\ \mathcal{R}'\ x$, produces the same set of results.

Therefore, $\theta_1 = \theta_{11} \vee \theta_{12}$, where

$$\theta_{11} = (v_0\ \mathcal{R}\ x) \wedge \neg(v_0\ \mathcal{R}'\ x)$$
$$\theta_{12} = \neg(v_0\ \mathcal{R}\ x) \wedge (v_0\ \mathcal{R}'\ x)$$

and $t_0 : [x \mapsto v_0]$.

As per our assumption, $t \not\models \theta_1$, i.e., $t \not\models \theta_{11}$ and $t \not\models \theta_{12}$. In other words, for $t : [x \mapsto v]$, both

$$\begin{aligned} &(v_0\ \mathcal{R}\ v) \wedge \neg(v_0\ \mathcal{R}' v) \\ &\neg(v_0\ \mathcal{R}\ v) \wedge (v_0\ \mathcal{R}' v) \end{aligned} \tag{3.1}$$

evaluate to false.

Case-Based Argument: Consider that $\mathcal{R}$ is the equality relation $=$. Let the mutation rule result in $\mathcal{R}'$ equal to $\neq$ relation. It is immediate that at least one of the formulas in Equation 3.1 must be satisfiable (specifically the second formula must be satisfiable when $\mathcal{R}$ is $=$ relation). Therefore, our assumption that t is not a satisfiable assignment of θ is contradicted.

Next consider that the mutation rule resulted in $\mathcal{R}'$ to be $\leq$ relation. Note that $db_{var} = v_0$ and $db_{var} \leq v_0$ in the `WHERE` clause of the original and the mutant queries, respectively, produced equivalent/ indistinguishable results for the test case t_0; on the other hand, $db_{var} = v$ and $db_{var} \leq v$ in the `WHERE` clause of the original and the mutant queries, respectively, produced non-equivalent/indistinguishable results for the test case t. As $v \neq v_0$ (in which case the test cases will become identical), there are two possibilities: $v < v_0$ and $v_0 < v$.

If $v < v_0$, then the `WHERE` clause conditions $db_{var} \leq v$ would have produced results equivalent to the ones produced by $db_{var} = v$. This is because $db_{var} \leq v_0$ and $db_{var} = v_0$ produce equivalent results. However, as t can kill the mutant, the results produced by the valuation v for the original and the mutant clauses must be different. Therefore, $v < v_0$ does not hold. Proceeding further, $v_0 < v$ implies that the second formula in Equation 3.1 is satisfied, which leads to contradiction of our assumption that t does not satisfy θ.

Similar contradictions can be achieved, and the theorem statement can be proved for other operations. $\square$

□

3.3 Experimental Results

3.3.1 Evaluation Criteria

ConSMutate can utilize any DSE-based test generation tools (e.g., Pex [20] in .NET applications) to generate high quality test cases for database applications, where quality is attributed to both coverage criteria and mutation score. We evaluate the benefits of our approach from the following two perspectives:

1. What is the percentage increase in code coverage by the test cases generated by Pex compared to the test cases generated by ConSMutate in testing database applications?

2. What is the percentage increase in mutation score of test cases generated by Pex compared to the ones generated by ConSMutate in testing database applications?

We first run Pex to generate test cases (different valuations for program inputs) for methods with embedded SQL queries in two open source database applications. We record the mutation score and code coverage percentage achieved by Pex. Next we apply ConSMutate to generate test cases for the same methods and record the corresponding mutation score and code coverage statistics. The experiments are conducted on a PC with a 2GHz Intel Pentium CPU and 2GB memory running the Windows XP operating system.

Table 3.4 Method names and corresponding Program Identifiers

UnixUsage		RiskIt	
Program Identifier(s)	**Method(s)**	**Program Identifier(s)**	**Method(s)**
1	courseIdExists	10	getOneZipcode
2	courseNameExists	11	filterMaritalStatus
3	getCourseIDByName	12	filterZipcode
4	getCourseNameByID	13	getValues
5	isDepartmentIdValid		
6	isRaceIdValid		
7	getDeptInfo		
8	deptIDExists		

3.3.2 Evaluation Test-Bed

Our empirical evaluations are performed on two open source database applications. They are *UnixUsage*[4] and *RiskIt*[5]. UnixUsage is a database application where queries are written against the database to display information about how users (students), who are registered in different courses, interact with the Unix systems using different commands. The database contains 8 tables, 31 attributes, and over a quarter million records. RiskIt is an insurance quote application which makes estimates based on users' personal information, such as zipcode. It has a database containing 13 tables, 57 attributes and over 1.2 million records[6]. Both applications are written in Java with backend Derby. To test them in the Pex environment, we convert the Java source code into C# code using a tool called Java2CSharpTranslator[7]. Since Derby is a database management system for Java and does not adequately support C#, we retrieve all the database records and populate them into Microsoft Access 2010. We also manually translate those original database drivers and connection settings into C# code.

Table 3.4 presents the methods in each of the test applications. The program identifiers 1–8 and 10–13 will be used to present our results in the rest of the sections.

3.3.3 Summary of Evaluation

Figure 3.2 shows the results of our evaluation. The graph compares the performances of Pex and ConSMutate in terms of achieving quality. The x-coordinates in the graph represent the Program Identifiers for different methods for Unix-Usage and RiskIt as mentioned in Table 3.4. The y-axis represents the Quality(%) in terms of Block Coverage and Mutation Score achieved by Pex and ConSMutate for various program identifiers.

3.3.3.1 Evaluation Criterion 1: Coverage Benefit.

Figure 3.2 (points shown in square) demonstrates the block coverage achieved by Pex and ConSMutate for both of the applications. Although Pex has achieved good block coverage as

[4]http://sourceforge.net/projects/se549unixusage
[5]https://riskitinsurance.svn.sourceforge.net
[6]http://webpages.uncc.edu/ kpan/coverageCriteria.html
[7]http://sourceforge.net/projects/j2cstranslator/

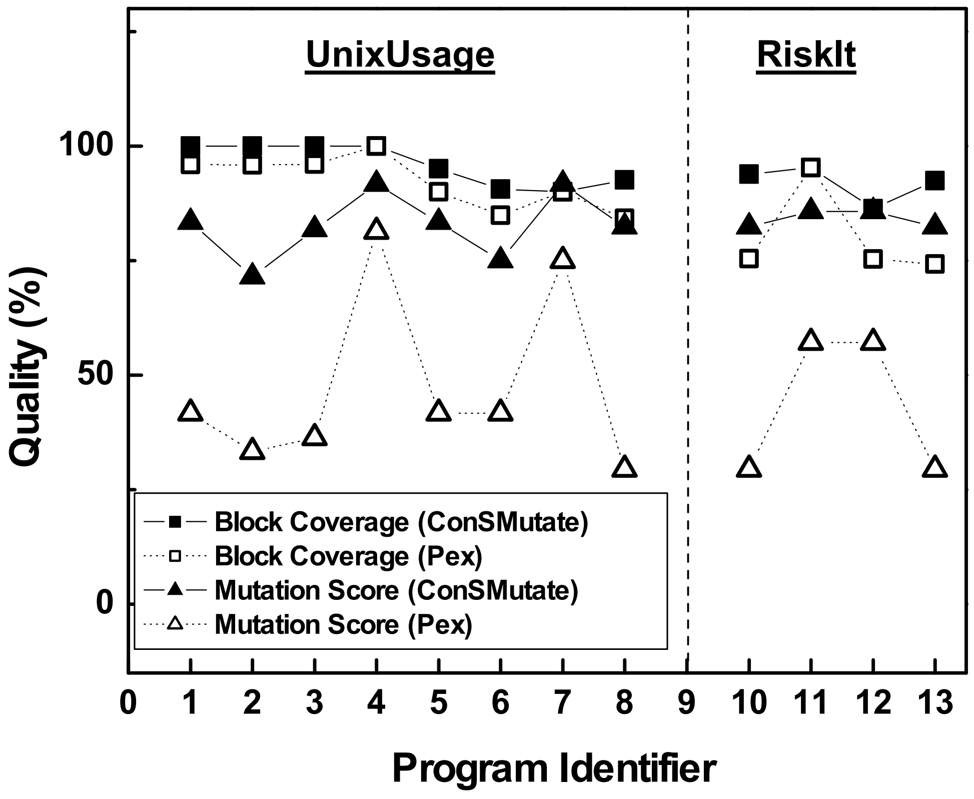

Figure 3.2 Comparison between Pex and ConSMutate in terms of quality

expected, ConSMutate has successfully achieved more than 10% improvement in coverage in case of various methods (program identifiers in figure 3.2). The reason for this is that Pex cannot generate sufficient program inputs to achieve higher code coverage, especially when program inputs are directly or indirectly involved in embedded SQL statements. ConSMutate does not suffer from this drawback, as it considers database states and the results of generated queries and their execution results.

3.3.3.2 Evaluation Criterion 2: Mutation Score Benefit.

Figure 3.2 (points shown in triangle) also demonstrates the mutation score achieved by Pex and ConSMutate for the test applications. The mutation score of test cases generated by ConSMutate is always higher than the mutation score of test cases generated by Pex. The increase in mutation score ranges from around 10% to 50%. We can see less increase in mutation score for methods like getCourseNameByID, getDeptInfo in Unix-Usage (program identifiers 4 and 7 in figure 3.2). Manual inspection reveals the fact that the improvement in mutation score is less for methods where the number of generated mutants are fewer than other methods.

The mutation scores achieved by ConSMutate are sometimes less than 100%, because the test cases generated by ConSMutate are *likely* to kill mutants and therefore may not be always successful. Figure 3.2 presents the mutation score achieved by ConSMutate by just performing constraint solving once (see Section 3.2.2). If the mutant is not killed by the test case obtained after one iteration of constraint solving, additional iterations of constraint solving can be done.

In our evaluation we do not eliminate equivalent mutants. We calculated mutation score as the number of mutants killed divided by total number of generated mutants. Note that there are a number of equivalent mutants for most of the cases and if we exclude these equivalent mutants, ConSMutate could achieve even higher mutant-killing ratios. Manual inspections show that the mutation scores achieved by ConSMutate are less than 100% because of the existence of equivalent mutants and because the database tables provided are not always sufficient to kill all the mutants.

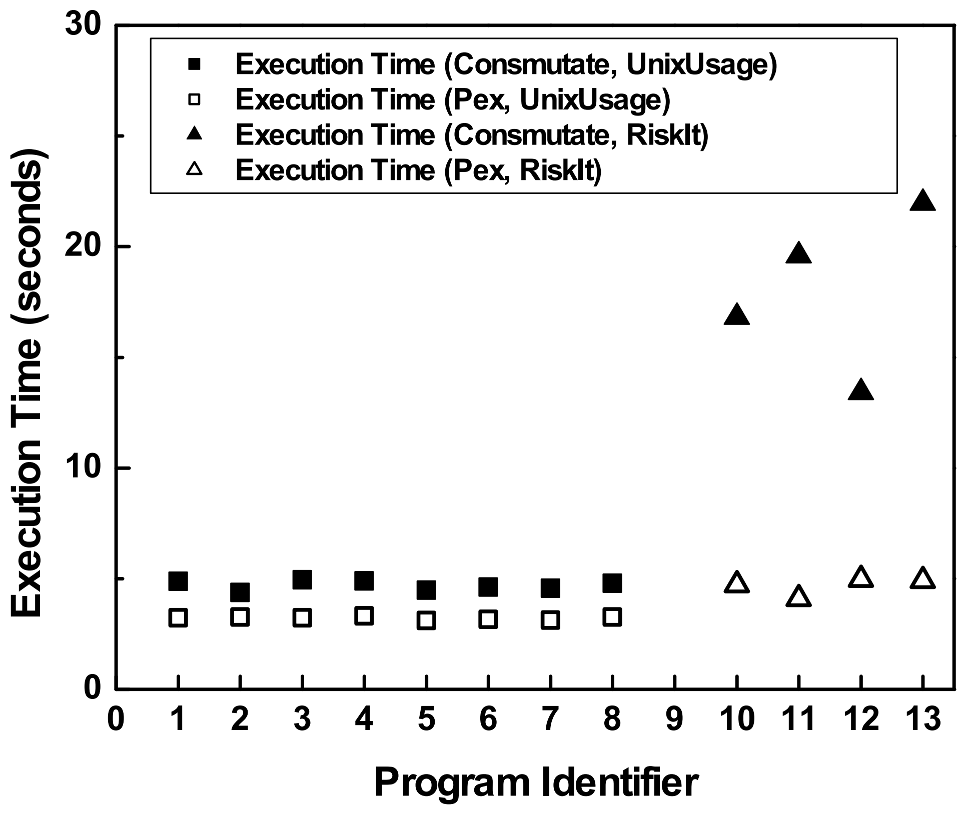

Figure 3.3 Execution time comparison between Pex and ConSMutate

3.3.4 Execution Time Overhead

As ConSMutate involves the database and utilizes a constraint solver for generating high quality test cases, there is obviously a penalty in terms of execution time. In this section, we show that the execution time overhead is not prohibitively large and therefore ConSMutate can be used effectively for test case generation for practical applications.

Figure 3.3 compares the execution times of Pex and ConSMutate. The x-coordinates in the graph represent the different Program Identifiers for Unix-Usage and RiskIt as mentioned in Table 3.4. The y-axis represents time. For UnixUsage, the execution time of ConSMutate is approximately 1.3 times that of Pex. The increase in time is due to multiple mutant query execution and subsequent comparison of the large result sets returned by them from the back-end database (more than 0.25 million records for UnixUsage). Multiple mutant executions are required in our framework in order to identify live mutants.

In the case of RiskIt, the increase in database size is *five*-times more than Unix-Usage. As a result, the total execution time increases by five times (maximum for the method identified by program 13). Optimizing multiple query execution is an open research problem and several research works in this area [63, 64] propose effective techniques which can reduce the total execution time by a considerable amount. Incorporating such techniques in our framework is not in the scope of our current objective but can be done easily to further improve the execution time.

Our evaluation results demonstrate the fact that ConSMutate successfully generates test cases for database application (where associated database state is given) which achieve high code coverage and mutation score as compared to the test cases generated by standard DSE engine.

CHAPTER 4. SynConSMutate: CONCOLIC TESTING OF DATABASE APPLICATIONS VIA SYNTHETIC DATA GUIDED BY SQL MUTANTS

4.1 Introduction

4.1.1 Driving Problem

Database applications are built using two different programming language constructs: one that controls the behavior of the application (host language); and the other that allows the application to access/retrieve information from the backend database (query language). In such applications, several branches are dependent on the query result; therefore current database state is an important factor to achieve high quality both in terms of coverage and mutation score in such applications. Existing works like [16, 17, 18] propose techniques to generate test cases and synthetic data to improve branch coverage but the generated test cases may suffer from low mutation score with respect to the same data. Auto generation of test cases and corresponding synthetic data to improve both coverage and quality; therefore, pose another important and interesting challenge.

4.1.2 Motivating Example

We present here a simple database application to illustrate the problem scenario. Consider the pseudo-code named CalculateTotalCost (shown in algorithm 2), which represents a typical database application. It takes available packets in stock as input (parameter x) and calculates total cost for `coffees` which has number of packets equal to x. The program creates a query string depending on the valuation of the parameter x, which also guides the control path in the program. Assume that the database table `coffees` contains no entry and its schema

```
1:  procedure CalculateTotalCost($x$)
2:      String q = " ";
3:      if x==0 then
4:          x++;
5:      end if
6:      q = "SELECT * FROM coffees WHERE packets =" + x + ";";
7:      result=executeQuery(q);
8:      while (result.next()) do
9:          totalCost = calculateCost(result.getInt(price), result.getInt(packets));
10:     end while
11:     return;
12: end procedure
```

Algorithm 2 Sample Pseudo code for Database Application

is shown in Table 4.1. To satisfy the branch condition at *line 3* ($x == 0$), test case ($x = 0$) is generated by concolic execution module of [16, 17].

Table 4.1 **coffees** Table schema definition

Column	Data Type	Constraint
cof_id	Int	Primary Key
cof_name	String	
price	Int	> 0
packets	Int	≥ 0

The concrete query executed at *line 6* will be,

Q: SELECT * FROM coffees WHERE packets = 1.

Table 4.2 Updated **coffees** Table in the database

cof_id	cof_name	price	packets
1	abc	1	1

Since the query will return an empty result and the program control cannot satisfy the true condition at *line 8*. To overcome such scenario, new synthetic data is created for the `coffees` table by [16, 17]. The synthetic data in Table 4.2 is created by solving the condition extracted from the query (packets = 1) in conjunction with `coffees` table schema constraint. This new entry improves the coverage (by satisfying the condition at *line 8*) of the generated test case ($x = 0$). Even though this new database state improves the structural coverage, it does not guarantee killing all the mutants corresponding to the executed query. For instance, for ($x = 0$) the query Q generates the tuple ($\langle 1, abc, 1, 1\rangle$) using the updated `coffees` table as shown in Table 4.2. A mutant of this query is,

$$\text{SELECT * FROM coffees WHERE packets} \leq 1.$$

This is generated by slightly modifying the $WHERE$ condition in the query (mimicking typical programming error). The result of the mutant is also ($\langle 1, abc, 1, 1\rangle$). That is, if the programmer makes a typical error of using $\leq$ in the $WHERE$ condition instead of the intended $=$ (or vice versa), then that error will go un-noticed for test case ($x = 0$) with respect to the updated `coffees` Table 4.2. We will show that our framework successfully identifies new synthetic data which will help test cases to identify such errors.

4.1.3 Problem Statement

How to automatically generate test cases and corresponding synthetic data for database applications where current database state is insufficient or absent? The test cases will not only ensure high coverage of the control flow described in host language, but will also allow for adequate testing of the embedded queries by attaining high mutation scores with respect to the generated data.

4.1.4 Individual Contributions

The contributions of our work described in this chapter are summarized as follows:

- We leverage our previous framework CoSMutate to develop an automatic test case generation approach that combines coverage analysis and mutation analysis for testing database applications even when associated physical database entries are absent (or insufficient).
- Our ew framework generates high quality test cases both in terms of structural coverage and mutation score.

- The framework also generates synthetic data to help improve the quality of the generated test cases.
- We demonstrate an empirical evaluation to show the effectiveness of our approach.

4.2 Approach Overview

Figure 4.1 shows the salient features of our framework, *SynConSMutate*. It has two main parts, *Application Branch Analyzer* and *Mutation Analyzer*. Application Branch Analyzer takes the program under test and the sample database (can be empty) as inputs, and generates test cases and synthetic data (if required) to satisfy any branch condition. It uses *SynDB* [17], built on top of DSE engine Pex [20], to generate test cases and synthetic data. After exploring each path by Application Branch Analyzer, the Mutation Analyzer performs quality analysis using mutation analysis. If the mutation score is low, the mutation analyzer generates a new test case (and corresponding synthetic data if required to satisfy branch condition) for the same path, whose quality is likely to be high. For the given (or newly generated) database state, if no new test case for the same path can improve the quality, then the mutation analyzer generates new synthetic data to help achieve the test case high quality. The steps followed in our framework are as follows:

Step 1: Generate Test Case and Associated Path Constraints using Application Branch Analyzer. In the first step, the framework uses the *Application Branch Analyzer* module to generate a test case value v, synthetic data if required and the associated execution path, called path constraint (PC). It may result a query execution (if the path includes some query). The executed query is referred to as the concrete query q_c and the query without the concrete values is referred to as the symbolic query q_s.

For the example in Section 4.1.2, in Step 1, Application Branch Analyzer generates a test case $v = (x = 0)$ and synthetic data shown in Table 4.2. This results in an execution path with path constraint $PC = (x == 0) \ \wedge \ (result.next() = true)$. It also results in a symbolic query and corresponding concrete query:

$$q_s\text{:SELECT * FROM coffees WHERE packets} = x_s,$$

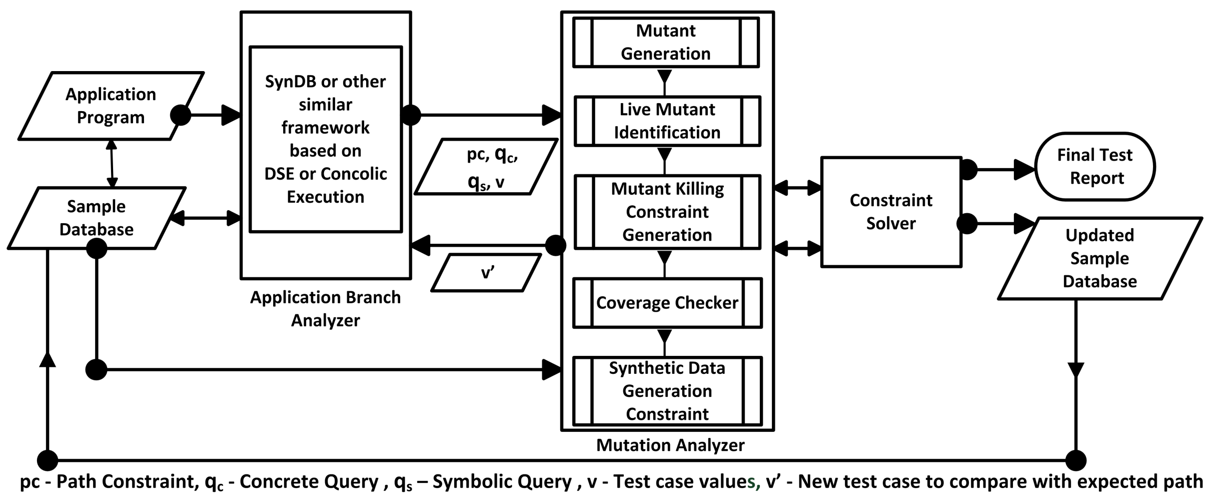
Application Program
Sample Database
SynDB or other similar framework based on DSE or Concolic Execution
Application Branch Analyzer
pc, qc, qs, v
v'
Mutant Generation
Live Mutant Identification
Mutant Killing Constraint Generation
Coverage Checker
Synthetic Data Generation Constraint
Mutation Analyzer
Constraint Solver
Final Test Report
Updated Sample Database
pc - Path Constraint, qc - Concrete Query , qs – Symbolic Query , v - Test case values, v' - New test case to compare with expected path

Figure 4.1 Framework For SynConSMutate

$$q_c\text{:SELECT * FROM coffees WHERE packets} = 1.$$

x_s is the symbolic state of the program input x, which is $x + 1$ in this case, at *line 4* (see program 2).

Application Branch Analyzer is loosely coupled with SynDB [17] which essentially depends on DSE engine Pex for exploring all possible branches of a given program. As an enhancement to DSE, SynDB tries to cover branches which depend on query result. To do that, SynDB treats symbolically both the embedded query and the associated database state by constructing synthesized database interactions. The original code under test is first transformed (instrumented) into another form that the synthesized database interactions can operate on. In order to force Pex to actively track the associated database state in a symbolic way, the concrete database state is converted into synthesized object, added it as an input to the program under test, and then passed it among synthesized database interactions. This results in integration of query constraints as normal constraints in the program code. Also database state is checked by incorporating database schema constraints into normal program code. Then, based on the instrumented code, SynDB guides Pex's exploration through the operations on the symbolic

```
public int calculatetotalCost(int x)
{
string query = " "‘;
int totalCost = -1;
SqlConnection sc = new SqlConnection();
sc.ConnectionString = "..";
sc.Open();
if(x == 0)
{
  x++;
}
query = "SELECT * FROM coffees WHERE packets =" + x;
SqlCommand cmd = new SqlCommand(query, sc);
SqlDataReader results = cmd.ExecuteReader();
while(results.Read())
{
  totalCost = calculateCost(result.getInt(3), result.getInt(4));
}
return totalCost
}
```

Figure 4.2   Actual code snippet of the Pseudocode from Section 4.1.2

database state to collect constraints for both program inputs and the associate database state. Then after applying Pex's constraint solver on the collected constraints, SynDB produces both program inputs and synthetic data to satisfy branch conditions which depend on query result.

For example, the pseudocode shown in Algorithm 2 can be written in actual *C#* code as shown in Figure 4.2. SynDB transforms the example code in Figure 4.2 into another form shown in Figure 4.3. In the instrumented code, SynDB adds 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. The schema as shown in Table 4.1 is represented as synthesized database state in Figure 4.4.

The program input *dbState* is then passed through synthesized database interactions. *SynSqlConnection,SynSqlCommand, SynSqlDataReader* are modified database interaction methods developed to mimic the actual *C#* database interations *SqlConnection,SqlCommand, Sql-*

```
public int calculatetotalCost(int x, DatabaseState dbState)
{
string query = " ";
int totalCost = -1;
SynSqlConnection sc = new SynSqlConnection(dbState);
sc.ConnectionString = "..";
sc.Open();
if(x == 0)
{
  x++;
}
query = "SELECT * FROM coffees WHERE packets =" + x;
SynSqlCommand} cmd = new SynSqlCommand(query, sc);
SynSqlDataReader results = cmd.ExecuteReader();
while(results.Read())
{
  totalCost = calculateCost(result.getInt(3), result.getInt(4));
}
return totalCost
}
```

Figure 4.3 Transformed code snippet produced by SynDB for the code in Figure 4.2

```
public class coffeesTable{
public class coffees {//define attributes;}
public List<coffees> coffeeRecords;
public void checkConstraints(){
   /*check constraints for each attributes */;
}
}

public class DatabaseState {
public coffeesTable coffee = new coffeesTable();
public void checkCOnstraints(){
/* check constraints for each table*/;
}
}
```

Figure 4.4 Synthesized Database State

DataReader. Meanwhile, at the beginning of the synthesized database connections, the framework ensures that the associated database state is valid by calling a method predened in *dbState* to check the database schema constraints for each table.

To synthesize database operations for the synthesized database interactions, it incorporates the query constraints as program-execution constraints in normal program code. To do so, within the synthesized method *ExecuteReader*, SynDB parses the symbolic query and transform the constraints from conditions in the **WHERE** clause into normal program code (if satisfied then leads to exploration of new branch 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. Then SynDB uses Pex for path exploration which eventually generates synthetic data if required.

Step 2: Execute Mutation Analyzer. After exploring a path of the program under test, SynConSMutate forwards PC, q_c, q_s and v to the *Mutation Analyzer* to evaluate the quality of the generated test case in terms of mutation score.

Step 2.1: Generate Mutant Queries. In Mutation Analyzer, the obtained q_c in Step 1 is mutated to generate several mutants in the *Mutant Generation* module. We have identified five rules which we call the *sufficient set of SQL mutation generation rules* from [8, 62] to identify logical errors present in the *WHERE* and *HAVING* clauses. Table 3.2 illustrates some of the rules. For instance, one of the mutants of the above query q_s is,

$$q_m\text{: SELECT * FROM coffees WHERE packets} \leq x_s.$$

Step 2.2: Identify Live Mutants. Using the test case under consideration, the live mutants are identified. Live mutants are those whose results do not differ from those of the concrete query in the context of the given database table.

The above mutant q_m is live under the test case $v = (x = 0)$ as q_c and q_m produces the same result for the database table (see Table 4.2).

Step 2.3: Generate Mutant Killing Contraints. A new set of constraints, θ is generated in *Mutant Killing Constraint Generation* module from

- the symbolic query q_s and its concrete version q_c,
- the live mutants (q_m's) from step 2.2,
- the path constraint of the execution (PC).

θ includes conditions on the inputs to the application. Due to the high cost of mutation analysis, we adopt the concept of weak mutation analysis [65]. Therefore, the test cases (if generated) do not guarantee killing the live mutants, but improve the probability of killing them.

θ is generated as follows. The mutant q_m is live because the WHERE clauses $packets = x_s$ and $packets \leq x_s$ do not generate two different result-sets. We also know that x_s is set to $x + 1$ (x is the test input) and $packets$ is set to 1. Therefore, the relationship between the valuations of the database attribute $packets$ and the test input x is $1 = x + 1$ in the original query-condition and $1 \leq x + 1$ in the mutant query-condition. We will use these relationships/conditions to generate the mutant killing constraint. In order to generate a different value of x to likely kill the mutant q_m, we need to choose a value for x such that $[(1 = x+1) \wedge (1 \nleq x+1)] \vee [(1 \neq x+1) \wedge (1 \leq x+1)]$. The last column of Table 3.2 demonstrates the general rules for generating these mutant killing constraints. Then we extract sub-path constraint pc_{pi}, which depends on program input (x in this case) from PC. The mutant killing constraint in conjunction with pc_{pi} (since the new test case should satisfy the executed path constraint) results in θ, the constraint which when satisfied is likely to generate a test case that can kill the mutant q_m.

$$\begin{aligned}\theta: \quad (x = 0) \wedge \ & [(1 = x + 1 \wedge 1 \nleq x + 1) \\ & \vee (1 \neq x + 1 \wedge 1 \leq x + 1)]\end{aligned}$$

Step 2.4: Find Satisfiable Assignment for θ and Corresponding Synthetic Data. The constraint θ is checked for satisfiability to generate a new test case in the *Constraint Solver* module (Z3[1] is used). If θ is satisfied then a new test case v' is identified by the framework.

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

In order to guarantee that v' explores the same execution path as was explored by v (see Step 1), new synthetic data may need to be generated to satisfy the branch condition which depends on query result (e.g., (result.next()=true) in PC in Step 1). To do that, *Coverage Checker* module first compares the executed path covered by v' with corresponding expected path (whichh is v in this case) and then generates the constraint expression (if required) by combining the $WHERE$ clause condition of the query executed by v' (i.e. concrete version of q_s with respect to the new test case v') in conjunction with the table (database) schema constraint (see Table 4.1). Constraint Solver module is again invoked to solve the generated expression. After updating the database state (if required) with the newly generated data, v' guarantees same structural coverage of the application as was achieved by v.

Furthermore, some mutants that were left "live" by v are now *likely to be* "killed" by v'. Therefore, it is necessary to check whether v' indeed kills the live mutants; if not, constraint solver is used again to solve θ for a program input (v'') and generate corresponding synthetic data (if required for coverage criterion). This iteration is terminated after pre-specified times (e.g., 10) or after all mutants are killed (whichever happens earlier). If the live mutants are killed, the control goes to **Step 3**. But there are situations where (a) θ becomes unsatisfiable or (b) the new test case valuations cannot kill the live mutants. This implies that for the given path PC and the given (or generated) database state there does not exist any new test case which can have higher mutation score than previous one. In order to improve mutation score, the control goes to **Step 2.5**.

In our example, θ becomes unsatisfiable, thus q_m is still alive. This means no new test case can be generated for the same path which can kill q_m with respect to the current database state (Table 4.2). Thus control goes to Step 2.5.

Step 2.5: Produce Synthetic Data Generation Constraint to Improve Mutation Score. To improve the mutation score of the generated test case, the *Synthetic Data Generation Constraint* module is triggered and a new set of constraints ψ is generated from

- the concrete query q_c from step 1,
- the sample database state from step 1,

- the live mutants (q_m's) from step 2.2.

ψ includes the database schema as a constraint expression. Otherwise, the generated synthetic data may become invalid with respect to the given database state, causing low quality test case generation for the database application.

In our example, ψ is generated as follows. The mutant q_m is live because there are not enough entries in the `coffees` table to generate different entries for WHERE clauses, $packets = 1$ (from q_C) and $packets \leq 1$ (from the concrete version of q_m). In order to improve the mutation score of the generated test case $x = 0$, we need to have an entry in the `coffees` table which satisfies $[(packets = 1) \wedge (packets \nleq 1)] \vee [(packets \neq 1) \wedge (packets \leq 1)]$ (again using mutant killing constraint rules (this case *ROR*) from Table 3.2). Thus, the constraint expression ψ will look like,

$$\begin{aligned}\psi: \quad \Re \wedge \ & [(packets = 1 \wedge packets \nleq 1) \\ & \vee(packets \neq 1 \wedge packets \leq 1)].\end{aligned}$$

Here, $\Re$ denotes the database schema constraint expression of `coffees` table obtained from Table 4.1.

Step 2.6: Find Satisfiable Assignment for ψ. The constraint ψ is checked for satisfiability to generate a new synthetic data. If ψ is satisfied, then the database state will be updated using the newly generated data. The updated database state will guarantee the previously generated test case to achieve high mutation score by killing the live mutants.

For instance, after solving ψ, the updated `coffees` table with newly generated synthetic data (second row) is shown in Table 4.3. With this new entry, the previously generated test case $x = 0$ now kills the live mutant q_m as shown in Table 4.4.

Table 4.3 **coffees** Table with new synthetic data

cof_id	cof_name	price	packets
1	abc	1	1
2	def	1	0

Table 4.4 Mutants and new Results for test case ($x = 0$)

Query	**Concrete Query**	**Result**
Actual q_c	SELECT * FROM coffees WHERE packets = 1	$\langle 1, abc, 1, 1\rangle$
Mutant q_m	SELECT * FROM coffees WHERE packets $\leq$ 1	$\langle 1, abc, 1, 1\rangle$, $\langle 2, def, 1, 0\rangle$

Step 3: Explore a New Execution Path. Finally, the whole process is iterated starting from **Step 1** to generate new test cases and new data (if required) that explore new execution paths of the program. This iteration continues until all possible branches are covered.

4.2.1 Discussion: Dealing with Nested Queries

SQL queries embedded in the program code could be very complex. One example is the involvement of nested sub-queries. The syntax of SQL queries is dened in the ISO standardization[2]. The basic structure of a SQL query consists of SELECT, FROM, WHERE, GROUP BY, and HAVING clauses. In case of *nested query* the predicate in WHERE or HAVING clause will look like ($C_i opQ$) where Q is an another query block. A large number of works [66, 67] on query transformation in databases have been explored to unnest complex queries into equivalent single level canonical queries. Researchers showed that almost all types of sub-queries can be unnested except those that are correlated to non-parents, whose correlations appear in disjunction, or some ALL sub-queries with multi-item connecting condition containing null-valued columns. In our work scope, we handle canonical queries in DPNF or CPNF form while generating test cases.

Generally, canonical queries can be categorized into two types, DPNF with the WHERE clause consisting of a disjunction of conjunctions like (`(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)`). DPNF and CPNF can be transformed mutually using DeMorgans rules[3].

[2]American National Standard Database Language SQL. ISO/IEC 9075:2008
[3]http://en.wikipedia.org/wiki/DeMorgan'slaws

4.3 Experimental Results

4.3.1 Evaluation Criteria

We evaluate the benefits of our approach from the following two perspectives:

1. What is the percentage increase in code coverage by the test cases generated by existing approaches like SynDB [17] and Emmi et al. [16] compared to the ones generated by SynConSMutate in testing database applications?

2. What is the percentage increase in mutation score of test cases generated by existing approaches like SynDB [17] and Emmi et al. [16] compared to the ones generated by SynConSMutate in testing database applications?

To set up the evaluation, we choose methods (denoted as program identifiers) from two database applications that have parameterized embedded SQL queries and program inputs are directly or indirectly used in those queries. First, we run SynDB [17] to generate test cases and synthetic data for those program identifiers. SynDB does not directly populate the real database schema, therefore in order to measure code coverage and mutation score of the original program, we separately populate the real empty database with those synthetic data and apply our previous framework ConSMutate to measure the code coverage and mutation score for the generated test cases. Second, we make use of SynDB to simulate Emmi et al.'s approach [16]. In this case, SynDB only generates synthetic data based on query conditions only, no database schema constraints were involved during data generation. Next we insert those entries to the real empty database and use ConSMutate to measure the same metrics as in first. Third, we apply SynConSMutate and record the code coverage and mutation score statistics for the same program identifiers. The experiments are conducted on a PC with 2GHz Intel Pentium CPU and 2GB memory running the Windows XP operating system.

4.3.2 Evaluation Test-Bed

Our empirical evaluations are performed on two open source database applications. They are *UnixUsage*[4] and *RiskIt*[5]. UnixUsage is an application which interacts with a database and the queries are written against the database to display information about how users (students) who are registered in different courses, interact with the Unix systems using different commands. The UnixUsage database contains 8 tables, 31 attributes, and over a quarter million records. RiskIt is an insurance quote application which makes estimates based on users' personal information, such as zipcode. The RiskIt database contains 13 tables, 57 attributes, and over 1.2 million records. In our evaluation, we assume an empty database at the beginning of each test case generation and we allow all the techniques to generate synthetic data to cover the branch conditions and to improve mutation score. Both applications were written in Java with backend Derby. To test them in our environment, we convert the Java source code into C# code using Java2CSharpTranslator[6] and the backend database into Microsoft Access 2010.

4.3.3 Summary of Evaluation

Figure 4.5 and 4.6 show the results of our evaluation. The graphs compare the performances of SynDB, Emmi et al.'s approach and SynConSMutate in terms of achieving quality. The x-coordinates in the graph represent different Program Identifiers (methods) for Unix-Usage and RiskIt. The y-axis represents the Quality(%) in terms of Block Coverage and Mutation Score achieved by SynDB, Emmi et al. and SynConSMutate for various program identifiers.

Evaluation Criteria 1: Coverage Benefit. The left hand sides of Figure 4.5 and 4.6 demonstrate the block coverage achieved by SynDB, Emmi et al.'s approach and SynConSMutate for both of the applications. The improvement in block coverage that SynConSMutate and SynDB achieve as compared to Emmi et al.'s approach ranges from almost *ten to seventy percent*. Close observation reveals the fact that Emmi et al.'s approach generates synthetic data without considering the database schema constraints. Therefore all the generated records cannot be inserted to the actual database, which leads to low block coverage. There is no

[4]http://sourceforge.net/projects/se549unixusage
[5]https://riskitinsurance.svn.sourceforge.net
[6]http://sourceforge.net/projects/j2cstranslator/

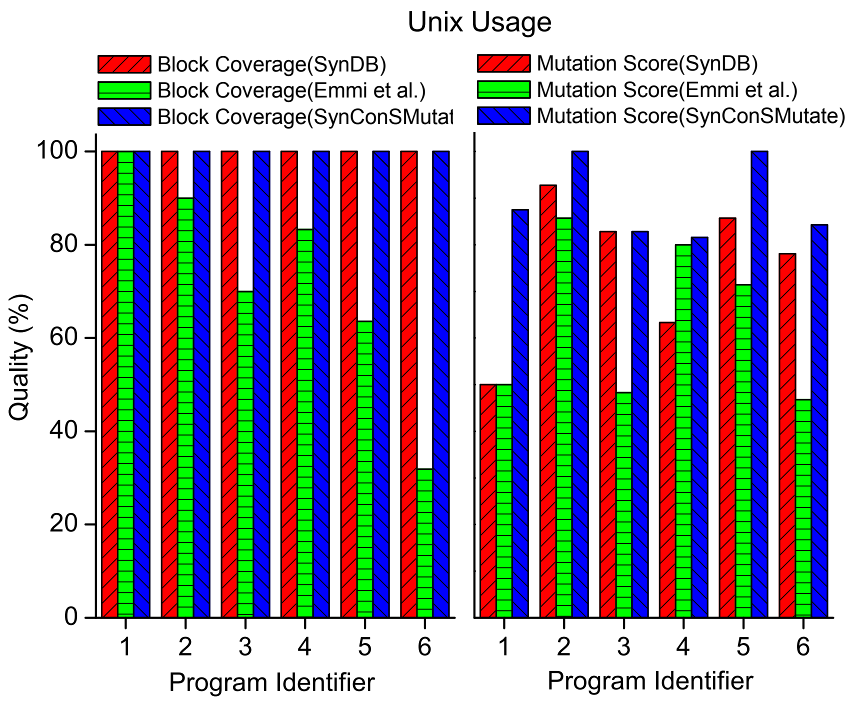

Figure 4.5 Comparison among SynDB, Emmi et. al.'s approach and SynConSMutate for UnixUsage

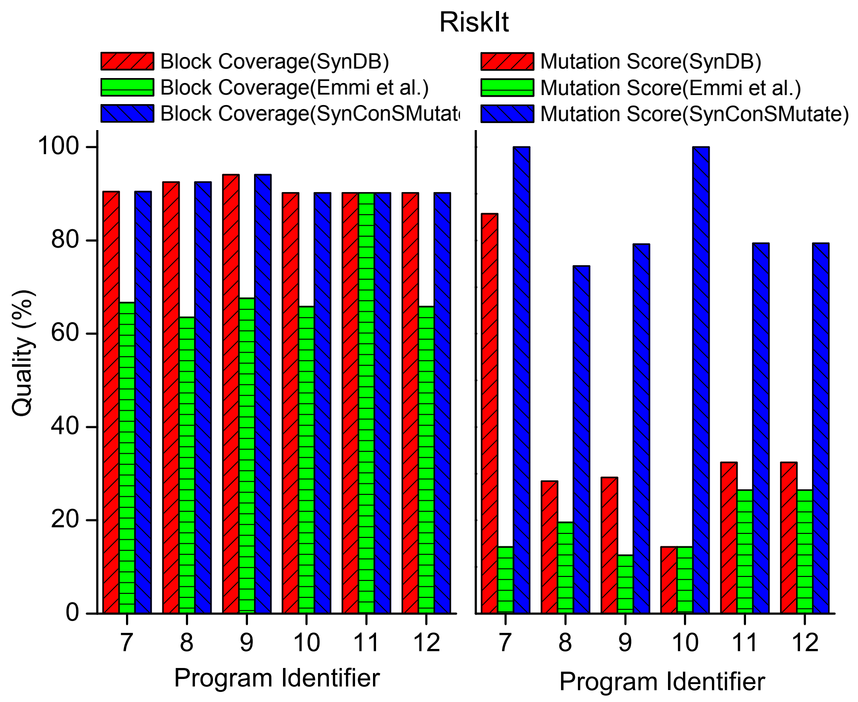

Figure 4.6 Comparison among SynDB, Emmi et. al.'s approach and SynConSMutate for RiskIt

significant improvement in block coverage in SynConSMutate compared to SynDB, as our *Application Branch Analyzer* is loosely coupled with SynDB to explore different branches of a given program.

Evaluation Criteria 2: Mutation Score Benefit. The right hand sides of Figure 4.5 and 4.6 demonstrate the mutation score achieved by SynDB, Emmi et al.'s approach and SynConSMutate for the test applications. The mutation score of the test cases generated by SynConSMutate is always higher than the mutation score of the test cases generated by SynDB and Emmi et al.'s approach. The increase in mutation score ranges from around *ten to eighty percent*. We can see less increase in mutation score in some cases (*e.g.*, program identifiers 2, 4). Further inspection reveals that the improvement in mutation score is less for methods where the numbers of generated mutants are fewer than other methods. Our evaluation demonstrates the fact that our framework SynConSMutate generates test cases which include both program inputs and synthetic data while achieving both high code coverage and high mutation score as compared to the ones generated by existing approaches [17, 16].

In our evaluation we do not eliminate equivalent mutants (semantically same as actual programs). We calculated the mutation score as the number of mutants killed divided by the total number of generated mutants. Manual analysis reveals that there are a number of equivalent mutants for most of the cases under our evaluation; if we exclude these equivalent mutants, SynConSMutate could achieve even higher mutant-killing ratios.

CHAPTER 5. CONCOLIC TESTING OF DATABASE APPLICATIONS WHILE GENERATING MINIMAL SET OF SYNTHETIC DATA

5.1 Introduction

5.1.1 Driving Problem

Typically, automated testing for database applications includes both generation of test cases and database states (synthetic data), if required. In reality, current database state may have data entries which can be used for testing. Using such entries in test case generation is more desirable as those entries represent real constraints that the application might face during execution. Moreover, using existing database state in generating high quality test cases bypasses the delay and overhead of identifying unnecessary synthetic data.

5.1.2 Motivating Example

We present here a simple database application to illustrate the problem scenario. Consider the pseudo-code named CALCULATEDISCOUNT (shown in Algorithm 3). It takes available packets in stock as input (parameter x), finds current discount rate for individual distributors to determine whether the distributor is eligible for more discount or not, and then calculates discount in price for `coffees` which has number of packets equal to x. Assume that the tables `coffees` and `distributor` have entries as shown in Table 5.1 and 5.2.

In database applications both program inputs (x in this case) and current database states are crucial in testing. In this example, we see that (1) the program input determines the result of the embedded SQL statement at *line*2, which in turn determines the condition at *line*4 and *line*10; (2) current database state not only determines the conditions at *line*4 and *line*10, but also determines whether *true* branch at *line*12 can be covered or not. Existing techniques

```
1:  procedure CALCULATEDISCOUNT($x$)
2:      String $q_1$ = "SELECT * FROM coffees c WHERE c.packets =" + x + ";";
3:      $result_1$ = executeQuery($q_1$);
4:      while ($result_1$.next()) do
5:          int i = $result_1$.getInt("price");
6:          int k = $result_1$.getInt("id");
7:          int pack = $result_1$.getInt("packets");
8:          String $q_2$ = "SELECT * FROM distributor d WHERE d.cid =" + k + ";";
9:          String $result_2$ = executeQuery($q_2$);
10:         while ($result_2$.next()) do
11:             int j = result2.getInt("discRate");
12:             if (i - j $\geq$ 5) then
13:                 AddMorediscount(i, pack);
14:             else
15:                 AddNodiscount(i, pack);
16:             end if
17:         end while
18:     end while
19: end procedure
```

Algorithm 3 **Sample Pseudo code for Database Application**

generate synthetic data (if required) along with test cases [16, 17] to improve branch coverage. Our work in chapters 3 and 4 have demonstrated that, the test cases which only satisfy high branch coverage may not be good in terms of identifying SQL related faults. We combine coverage analysis and mutation analysis to generate test cases and synthetic data (if required) so that the generated test cases not only satisfy high branch coverage but also high mutation score.

However, it often happens that a given database state with existing records returns no records (or records that do not satisfy subsequent branch conditions) when the database executes a query with arbitrarily chosen program input value. For example, consider the program in algorithm 3, since x is an integer and it's domain is large, existing approaches like [16, 17] which are based on concolic execution (or DSE) can choose any concrete value for x. Therefore it is very likely that the query at *line*2 will return no records with respect to the Table 5.1. Therefore, existing techniques [16, 17] generate synthetic data for Table 5.1 so that conditions at *line*4 gets satisfied with respect to the chosen value for x. In subsequent iterations,

Table 5.1 New Table **coffees**

id	name	price	packets
1	French	5	5
2	Colombian	5	9
3	English	8	8
4	Espresso	5	10

Table 5.2 Table **distributor**

cid	did	name	discRate	indvPack
1	1	Rob	0	5
2	2	Bob	0	9
4	3	Ron	1	10
3	4	John	3	8

existing approaches generate more synthetic data for Table 5.1 and 5.2 to cover subsequent branch conditions at $line10$ and $line12$. Our work [61] focuses on generating synthetic data not only to improve structural coverage but also the mutation score, therefore generated test cases in our approach achieve high quality both in terms of coverage and mutation score. But none of these approaches (including ours) consider existing database states and the relationship among database variables, program inputs and branch conditions to generate test cases for the program.

Our approach in this chapter generates test cases for database applications by maximizing the usage of the existing database state. The generated test cases not only achieve high branch coverage but also ensure high mutation score. For example, by looking into the existing database state as shown in Table 5.1, we can say that the test case $x = 5$ satisfies branch condition at $line4$ without generating a new synthetic data. It results in the execution of the query at $line2$,

$$\text{SELECT * FROM coffees WHERE packets} = 5.$$

The query generates the tuple $(\langle 1, French, 5, 5\rangle)$ using the `coffees` Table 5.1. Even though the test case $x = 5$ improves branch coverage without generating any new synthetic data, it may fail to identify possible fault that might be present in the query at $line2$. As an example a mutant of the abovementioned query is,

$$\text{SELECT * FROM coffees WHERE packets} \leq 5.$$

This is generated by slightly modifying the $WHERE$ condition in the query (mimicking typical programming error). The result of the mutant is also $(\langle 1, French, 5, 5\rangle)$. That is, if the programmer makes a typical error of using $=$ in the $WHERE$ condition instead of the intended $\leq$, then that error will go un-noticed for test case $(x = 5)$ with respect to the `coffees` Table 5.1. This shows the fact that test case $x = 5$ can improve the structural coverage without generating new synthetic data but fails to identify common programming error that may present in the embedded SQL statement. We will show that our framework successfully identifies new test case which will identify such errors. If no new test case can improve the quality (both in terms of coverage and mutation score), our approach generates synthetic data that will help the generated test cases to improve the quality. Thus only minimal set of synthetic data will be generated.

5.1.3 Problem Statement

How to automatically generate test cases for database applications by maximizing the usage of the existing database state? Test cases generated by the new strategy will reduce the redundant generation of synthetic data by maximizing the usage of current database state. Thus only minimal set of synthetic data will be generated to achieve high quality, both in terms of coverage and mutation score.

5.1.4 Individual Contributions

The contributions of this portion of our work are summarized as follows:

- We propose a new test case generation technique to reuse the current database state.
- Our new strategy reduces the overhead of unnecessary synthetic data generation while generating high quality test cases and only generates minimal set of synthetic data to improve such quality metrics.

5.2 Approach

Testing database applications has two important challenges:

- Generate test cases to validate correctness or find bugs by improving structural coverage (statement, block or branch coverage) of the program, and
- Identify minimal set of synthetic data which help test cases to improve coverage metrics.

We propose and develop a framework which comprehensively addresses these challenges by incorporating mutation analysis in coverage-based automatic test case generation. We show that the test cases generated in our framework are superior both in terms of coverage and in terms of mutation score.

Solution Overview. We present an approach that is capable of automatically generating high quality test cases for database applications by maximizing the usage of the existing database state. It relies on Concrete and Symbolic execution of the application program written in host language (language in which the database application is coded) and uses mutation analysis of database-queries written in embedded language to guide the generation of high quality test cases.

Our approach addresses an important challenge in the problem context: since concolic execution (or similar technique like Dynamic Symbolic Execution(DSE)) cannot solve constraints (branch conditions) derived from the existing database state, current approaches [16, 17] generate new synthetic data so that the generated test case can satisfy the particular constraint. Our approach combines the relationship among program inputs, database variables and constraints generated by them and formulate a new *intermediate query* to identify a new test case. For each new test case generated for each path, we measure the mutation score of the test case. If the mutation score of the test case is below the pre-specified threshold, our framework analyzes the path constraints (necessary for coverage) and mutation-killing constraints (necessary for high mutation score) in conjunction with the current database state using the intermediate query formulation technique (necessary for identifying range of acceptable test case values) and uses a constraint solver to automatically identify a new test case for the same path with high quality. If, no new test case can improve the branch coverage or mutation score, new synthetic data will be generated. With respect to the updated database state (including newly generated data), the generated test case will guarantee achieving high quality in terms of coverage and mutation score.

5.2.1 Approach Overview

Figure 5.1 shows our framework which has two main parts, *Application Branch Analyzer* and *Mutation Analyzer*. Application Branch Analyzer takes the program under test and the sample database (can be empty) as inputs, and generates test cases to satisfy a branch condition. It uses Pex [20], a dynamic symbolic execution engine (other engines like concolic testing tool [36] can also be used), to generate test cases by carefully comparing the concrete and symbolic execution of the program. Since Pex cannot solve branch conditions derived from existing database state, a new module called *Intermediate Query Construction* is introduced. This module considers the current database state and exploits the relationship among program input, database variables and the branch condition to identify a new test case. If the current database is insufficient or empty to generate such new test case, our framework uses module from *SynDB* [17] (built on top of DSE Engine Pex) to generate synthetic data so that the previously generated test case can satisfy the current branch condition. After exploring each path by Application Branch Analyzer, the Mutation Analyzer performs quality analysis using mutation analysis. If the mutation score is low, the mutation analyzer generates a new test case (by considering the current database state) for the same path, whose quality is likely to be high. For the current database state, if no new test case for the same path can improve the quality, then the mutation analyzer generates new synthetic data to help achieve the test case high quality. The steps followed in our framework are as follows:

Step 1: Generation of Test Cases and Associated Path Constraints Using Application Branch Analyzer. In the first step, the framework uses the *Application Branch Analyzer* module to generate a test case value v and the associated path constraints. It results in a specific execution path constraint (say, PC) of the application, which in turn results in a database query execution (if the path includes some query). The executed query is referred to

```
1: procedure ConstructQuery($Q_c$, $Q_s$, $PI$, $PC$)
2:     Initialization for intermediate query construction
   Set containing $SELECT$ clause attributes $S$,
   Set containig $FROM$ clause attributes $F$,
   Set containing $WHERE$ clause attributes $W$,
   A hashset to store the relationship between program input and database variable
3:     Find variables $V_{db}$ =$\{v_{db1}, v_{db2},..\}$ dependent on database variables and
   the corresponding relationship set with database variables $R_{db} = \{r_{db1}, r_{db2}, ..\}$
4:     Extract each concrete query $q_c$ and coresponding symbolic one $q_s$ from $Q_c$ and $Q_s$
5:     for Each pair of $q_c$ and $q_s$ do
6:         Call QuerySetCreation1($q_c$, $q_s$, $S$, $F$, $W$)
7:     end for
8:     Initialize $PC'$, a new set to store branch condition predicates
9:     Call QuerySetCreation2($PC'$, $PC$, $V_{db}$, $R_{db}$)
10:    Call CreateQuery($PC'$, $S$, $F$, $W$)
11: end procedure
```

Algorithm 4 **Intermediate Query Construction**

```
1: procedure QuerySetCreation1($q_c$, $q_s$, $S$, $F$, $W$)
2:     Copy $FROM$ clause from $q_s$ to $F$
3:     for each condition $C_i$ in $WHERE$ clause from $q_s$ do
4:         if $C_i$ contains program input $pi_i \in PI$ then
5:             Copy the associated database variable to $S$ and store $\langle pi_i, C_i\rangle$ in $R_i$
6:         else
7:             if $C_i$ contains database variables then
8:                 By comparing corresponding $v_{dbi}$ and $r_{dbi}$, replace corresponding variable with
                                              database variable in $C_i$ and copy to $W$
9:             else
10:                Copy concrete valuation of $C_i$ from $q_c$ to $W$
11:            end if
12:        end if
13:    end for
14: end procedure
```

Algorithm 5 **Part 1: Intermediate Query's SELECT, FROM, WHERE clause creation**

```
procedure QuerySetCreation2($PC'$, $PC$, $V_{db}$, $R_{db}$)
    for each $pc_i \in PC$ after the query execution do
        if $pc_i$ contains any variables($v_{dbi}$) from $V_{db}$ then
            Find corrsponding relationship expression $r_{dbi}$ from $R_{db}$
            Replace variables in $pc_i$ with corresponding database variables
                                                by comparing $v_{dbi}$ and $r_{dbi}$
            Copy it to $PC'$
        end if
    end for
end procedure
```

Algorithm 6 **Part 2: Intermediate Query's WHERE clause creation**

```
procedure CreateQuery($PC'$, $S$, $F$, $W$)
    if ConStructQuery procedure executes to traverse a new branch condition then
        Flip last branch condition in $PC'$
    else if ConStructQuery procedure executes to improve the mutation score then
        Keep as it is
    end if
    Copy $PC'$ to $W$
    Append all $s_i \in S$ to intermediate query's $SELECT$ clause
    Append all $f_i \in F$ to intermediate query's $FROM$ clause
    Append all $w_i \in W$ to intermediate query's $WHERE$ clause
                                                as conjunctive normal form
end procedure
```

Algorithm 7 **Creation of the Intermediate Query**

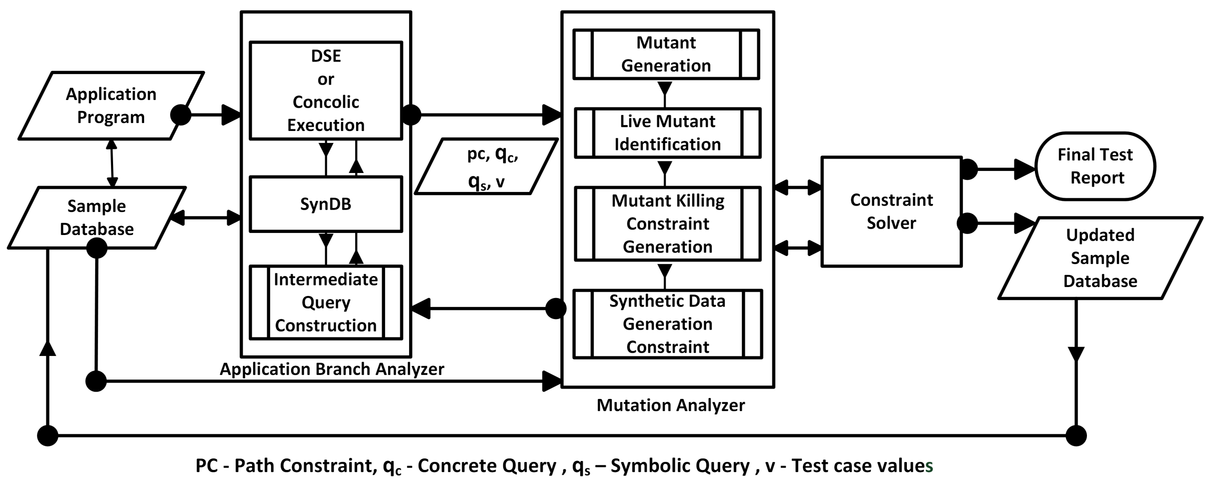
Application Program
Sample Database
DSE or Concolic Execution
SynDB
Intermediate Query Construction
Application Branch Analyzer
pc, qc, qs, v
Mutant Generation
Live Mutant Identification
Mutant Killing Constraint Generation
Synthetic Data Generation Constraint
Mutation Analyzer
Constraint Solver
Final Test Report
Updated Sample Database
PC - Path Constraint, qc - Concrete Query , qs – Symbolic Query , v - Test case values

Figure 5.1 New Framework for Testing Database Applications

as the concrete query q_c and the same without the concrete values (with the symbolic state of the input variable) is referred to as the symbolic query q_s. The path constraints refer to the conditions which must be satisfied for exploring the execution path in the application.

Going back to the example in Section 5.1.2, Application Branch Analyzer (Pex in our case) generates a test case randomly, say $v = (1)$, i.e., $x = 1$. This results in an execution path with path constraints $PC = (result_1.next() \neq true)$. It also results in a symbolic query and a corresponding concrete query:

$$\text{Symbolic } q_s\text{: SELECT * FROM coffees WHERE packets} = x_s,$$
$$\text{Concrete } q_c\text{: SELECT * FROM coffees WHERE packets} = 1.$$

where x_s is related to program input x as $x_s = x$ in this case (see the example program in Section 5.1.2).

Step 1.1: Intermediate Query construction to improve branch coverage. DSE [20] or concolic testing [36] techniques can not solve branch conditions which depend on executed query result. Recent other techniques like [17, 16, 61] analyze previously executed path and generate synthetic data(s) to the existing database which can satisfy such branch conditions and improve the branch coverage.

Table 5.3 Updated Table **coffees**

id	name	price	packets
1	French	5	5
2	Colombian	5	9
3	English	8	8
4	Espresso	5	10
5	abc	1	1

Going back to the example, existing techniques insert a new record to the `coffees` table as shown in Table 5.3. This new entry will help test case $x = 1$ to satisfy branch condition $(result_1.next() = true)$ (as the query at *line 2* does not return empty result, see program 3), thus branch coverage is improved.

Our current approach differs from the existing techniques as we leverage concolic execution technique as a supporting technique and generate test cases by executing newly constructed intermediate queries against the current database state. This results in high code coverage without generating unnecessary synthetic data. Our approach can assist any recent techniques to determine test cases such that the executed query results return records and branch conditions which depend on those results can be covered.

If we look at program in Algorithm 3, we see that the concrete valuation of x is assigned directly to the database variable *packets* in the query at *line 2*. Now if we look at Table 5.1, we can see records with concrete *packets* values. If we choose any of those valuation of packets ($\langle 5, 8, 9, 10\rangle$) as a test case valuation of x, branch condition at *line 4* will be satisfied (as the query at *line 2* will return non-empty result).

Algorithm. Intermediate query construction technique combines the relationship among program inputs, database variables and constraints generated by them and formulate the query to identify a new test case which will help improve the quality. Algorithm 4 illustrates how to construct an intermediate query. The algorithm accepts the set of queries (Q_c) executed during previous execution and their corresponding symbolic ones (Q_s), set of program input values (PI) and the captured branch conditions (PC).It creates the auxiliary query and also stores the relationship between program inputs and database variables to choose the concrete value for the test case (x in this case).

ConstructQuery calls QuerySetCreation1 (see Algorithm 5) which illustrates how to construct the $SELECT, FROM, WHERE$ clauses of the intermediate query from the queries executed in the previous path. All the tables that are present in the $FROM$ clause of the executed queries are copied to the set F (set which holds all $FROM$ clauses of the intermediate query). Next we decompose concrete and symbolic queries using a simplified SQL parser and get their `WHERE` conditions (C_is), which we assume to be in conjunctive normal form. For each C_i, we check whether C_i contains program input from PI. If yes, we copy the associated database variable to S (set which holds intermediate query's $SELECT$ clause) and also store the program input and the corresponding database relationship into the hash set R_i. If not, we insert proper clause to W to identify $WHERE$ clause conditions for the intermediate query.

In our running example, the executed query at *line 2* (see Program 3) has only one $FROM$ clause. Therefore F will contain only one value, `coffees`. Also, the executed query has only one $WHERE$ clause in this case. The clause has x which is a program input. So, the corresponding database variable *packets* will go into the S clause and the program input x and its relationship with database variable $packets = x$ will be stored as $\langle key, value \rangle$ pair in R_i. The $WHERE$ clause set W is empty in this case. So, the intermediate query will look like,

SELECT packets FROM coffees.

After executing this query against the current database table (see Table 5.1), we will get a range of values $\langle 5, 8, 9, 10 \rangle$. Now we can choose any of these values as a new test case which will satisfy the condition $result_1.next() = true$, thus improving the branch coverage without generating a new synthetic data. Let us assume that the framework arbitrarily chooses new value $x = 5$. This clearly improves branch coverage, but does not guarantee that the test case is of high quality in terms of both branch coverage and mutation score (see Section 5.1.2 for the example problem scenario).

Step 2: Deployment of Mutation Analyzer. After exploring a path of the program under test, our framework forwards PC, q_c, q_s and v to *Mutation Analyzer* to evaluate the quality of the generated test case in terms of mutation score.

Step 2.1: Generation of Mutant Queries. In Mutation Analyzer, the obtained concrete query q_c is mutated to generate several mutants q_m(s). The mutations are done using pre-specified mutation functions in the *Mutant Generation* module.

It is generally agreed upon that a large set of mutation operators may generate too many mutants which, in turn, exhaust time or space resources without offering substantial benefits. Offutt et al. [62] proposed a subset of mutation operators which are approximately as effective as all 22 mutation operators of Mothra, a mutation testing tool [38]. They are referred to as *sufficient mutation operators*. In our context, we are specifically focused on SQL mutants. We have identified *six* mutation operators by comparing SQL mutation operators developed in [8] with the sufficient set of mutation operators mentioned in [62]. We refer to these five rules as the *sufficient set of SQL mutation operators*, sufficient to identify logical errors present in the `WHERE` and `HAVING` clauses.

Our approach uses these mutation operators in generating mutants. It should be noted here that new mutation operators can be considered and incorporated in mutation generation module in our framework as and when needed. Table 3.2 (first three columns) presents such mutation generation rules.

Going back to the example in section 5.1.2, after executing the program with test case $x = 5$ we have new concrete query q_c,

$$\text{SELECT * FROM coffees WHERE packets} = 5$$

and its corresponding symbolic version q_s is,

$$\text{SELECT * FROM coffees WHERE packets} = x_s.$$

Now, one of the mutants of the symbolic q_s is

$$q_m\text{: SELECT * FROM coffees WHERE packets} \leq x_s.$$

In the above transformation, α is "=" (equality relational operator) and β is "$\leq$" (less-than-equal-to relational operator) as per the rule in the first row, second and third columns of Table 3.2. Point to be noted here is that we only consider query at *line 2* in our mutation analysis as it is the only query which includes program input as input parameter, thus valuation of the program input (x in this case) determines the query result. Therefore, in our example program, query at *line 8* will not be considered for mutation analysis.

Step 2.2: Identification of Live Mutants. Using the test case under consideration, the live mutants are identified. Live mutants are the ones whose results do not differ from that of the concrete query in the context of the given database table. The above mutant q_m is live under the test case $v = (x = 5)$ as it results in a concrete query

$$\text{SELECT * FROM coffees WHERE packets} \leq 5$$

The above query and the concrete query q_c produce the same result for the given database table (Table 5.1). Therefore, q_m is live under the test case $(x = 5)$.

Step 2.3: Generation of Mutant Killing Constraints. A new set of constraints, θ is generated in *Mutant Killing Constraint Generation* module from

- the symbolic query q_s and its concrete version q_c,
- the live mutants (q_m's),
- the path constraint of the execution, and
- the range of acceptable values of the program input(s) with respect to the current database state.

θ includes conditions on the inputs to the application. Due to the high cost of mutation analysis, we adopt the concept of weak mutation analysis [65]. Therefore, the test cases (if generated) do not guarantee to kill the live mutants, but improve the probability of killing them.

θ is generated as follows. The mutant q_m is live because the WHERE clauses $packets = x_s$ and $packets \leq x_s$ do not generate two different result-sets. We also know that x_s is set to x (x is the test input) and *packets* is set to 5. Therefore, the relationship between the valuations of the database attribute *packets* and the test input x is $5 = x$ in the original query-condition and

$5 \leq x$ in the mutant query-condition. We will use these relationships/conditions to generate the mutant killing constraint. In order to generate a different value of x to likely kill the mutant q_m, we need to choose a value for x such that $[(5 = x) \wedge (5 \nleq x)] \vee [(5 \neq x) \wedge (5 \leq x)]$. The last column of Table 3.2 demonstrates the general rules for generating these mutant killing constraints.

Now, if the above expression is satisfiable, the constraint solver may produce any arbitrary value for x. Since the concrete value of x is assigned to database variable *packets*, any arbitrary value of x might not satisfy the path constraint as well as might not kill the mutant. As an example, the above expression is satisfiable for $x = 6$. Constraint solver clearly gives us a new solution for x, but $x = 6$ will make the query at *line*2 (see program 3) return empty result, thus not satisfying the desired branch condition.

In order to avoid this scenario, mutation analyzer triggers Intermediate Query Construction to get a range/set of acceptable values for program input x. Since x has a relationship with the database variable *packets*, we exploit such relationship, get range/set of values for *packets* using intermediate query and then derive the acceptable range/set of values for the program input x.

As an example, test case $x = 5$ executes both the queries at *line*2 and *line*8 while satisfying *true* conditions at *line*4, *line*10, *line*12. Algorithm (see 4) which calls Algorithm 5, constructs the sets S, F, W as follows. S will only have $\langle packets \rangle$ as the *WHERE* clause $packets = x$ (at *line 2*) has program input x which is assigning value to database variable *packets*. F will have $\langle$`coffees, distributor`$\rangle$ as there are two table names in the two *FROM* clauses (see the queries at *line*2 and *line*8 in program 3). W will have $\langle cid = id \rangle$ after replacing variable k with its corresponding database variable *id* as the variable k in the *WHERE* clause at *line 8* is dependent on database variable *id* (see *line 6* in program 3).

In database applications there may exist branch conditions which are data dependent on returned query results. In our example condition at *line*12, i.e. branch condition $i - j \geq 5$, correspond to the values of attributes *price* and *discRate* of returned records by query at *line*2 and query at *line*8 respectively. QuerySetCreation2, as shown in Algorithm 6, demonstrates how to incorporate such branch conditions in intermediate query's W clause.

Otherwise, the intermediate query will also return concrete values (of *packets*) which are not related to the current path. For each variable $v_{dbi} \in V_{db}$, we store the corresponding relationship with database variable in the set R_{db}. For each branch condition $pc_i \in PC$, we check whether any variable is data-dependent on any V_{db}. If yes, by comparing v_{dbi} with corresponding r_{dbi} we replace the variable with corresponding database variable. The new expression of pc_i is stored into PC'. All branch conditions in PC' are appended to the W set. In our example, PC' will only contain $(price - discRate \geq 5)$ from $(i- \geq 5)$. Therefore the new intermediate query will be,

SELECT packets FROM coffees, distributor
WHERE cid = id AND (price - discRate) ≥ 5.

This query will be executed against the current database state (see Table 5.1 and 5.2). The result set will be $\langle 5, 8, 9 \rangle$. The set R_i holds the relationship between program input (x in this case) and database variable (*packets* in this case), which is $packets = x$. By obtaining such relationship from R_i, we can create the acceptable set of values for program input x. The expression will be,

$$(x = 5) \vee (x = 8) \vee (x = 9).$$

Then we extract sub-path constraint pc_{pi}, which depends on program input (x in this case) from PC. The mutant killing constraint in conjunction with the range expression and pc_{pi} (since the new test case should satisfy the executed path constraint, though no branch condition depends on x in this case, makes pc_{pi} empty) results in θ, the constraint which when satisfied is likely to generate a test case that can kill the mutant q_m.

$$\theta : \quad ((x = 5) \vee (x = 8) \vee (x = 9)) \wedge \; [(5 = x \wedge 5 \not\leq x) \\ \vee (5 \neq x \wedge 5 \leq x)].$$

Step 2.4: Find Satisfiable Assignment for θ. The constraint θ is checked for satisfiability to generate a new test case in the *Constraint Solver* module (Z3[1] is used). If θ is satisfied then a new test case v' is identified by the framework. The mutants that were left "live" by v are

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

Table 5.4 Mutants and results for test case (8)

Query	**Concrete Query**	**Result**
q_c	SELECT * FROM coffees WHERE packets = 8	$\langle 3, English, 8, 8\rangle$
q_m	SELECT * FROM coffees WHERE packets $\leq$ 8	$\langle 1, French, 5, 5\rangle$, $\langle 3, English, 8, 8\rangle$

now *likely to be* "killed" by v'. Therefore, it is necessary to check whether v' indeed kills the live mutants; if not, constraint solver is used again to solve θ and generate corresponding synthetic data (if required for coverage criterion). This iteration is terminated after pre-specified times (e.g., 10) or after all mutants are killed (whichever happens earlier). If the live mutants are killed, the control goes to *Step 3*. But there are situations where (a) θ becomes unsatisfiable or (b) the new test case valuations cannot kill the live mutants. This implies that for the given path PC and the given (or generated) database state, there does not exist any new test case which can have higher mutation score than previous one. In order to improve mutation score, the control goes to *Step 2.5*.

Going back to our running example, when the SMT solver generates a satisfiable assignment $x = 8$ for the mutant killing constraint θ (see above), the new test case $v' = (8)$ successfully kills the live mutant q_m by distinguishing its result from the original query result, as shown in Table 5.4.

Step 2.5: Produce Synthetic Data Generation Constraint to Improve Mutation Score. To improve the mutation score of the generated test case, the *Synthetic Data Generation Constraint* module is triggered and a new set of constraints ψ is generated from

- the concrete query q_c,
- the sample database state,
- the live mutants (q_m's), and
- the path constraint of the execution.

ψ includes the database schema as a constraint expression. Otherwise, the generated synthetic data may become invalid with respect to the given database state, causing low quality test case generation for the database application. More detailed description of this module which has been leveraged from our previous work can be found in [61].

Going back to our running example, after generating test case $x = 8$, the control goes to $step1$ to generate a new test case for the uncovered branch condition, which is the $else$ branch condition $(i - j < 5)$ at $line14$. Since the branch condition depends on the result set returned by the queries at $line2$ and $line8$, Pex or Concolic testing techniques can not generate test cases for such branch conditions. Recent other techniques [16, 17, 61] analyze the queries executed in the previous execution, exploit the relationship between branch condition and query result conditions ($WHERE$ clauses) and generate synthetic data so that the previously generated test case can satisfy such branch condition. Our approach, on the other hand, exploits the relationship among program input, query conditions, and branch conditions and checks the current database state to find any new test case which can satisfy such condition with respect to the current database state. Our algorithm ConstructQuery (see Algorithm 4 which calls Algorithms 5, 6, 7) constructs a new intermediate query to find concrete valuation of packets (as $packets = x$, see Program 3). The intermediate query will be,

$$\text{SELECT packets FROM coffees, distributor}$$
$$\text{WHERE cid} = \text{id AND (price - discRate)} \leq 5.$$

This will result only one value, i.e., $packets = 10$. As we know, $packets = x$, therefore the new test case will be $v = (10)$, $x = 10$. This new test case will surely cover the branch condition at $line14$ without generating any new synthetic data. But it might not be able to kill all the mutants generated by the rules as described in Table 3.2. As an example, a mutant will be,

$$\text{SELECT * FROM coffees WHERE packets} \geq 10.$$

This mutant will be live as the executed query at $line2$ and this mutant return same result set$(\langle 4, Espresso, 5, 10\rangle)$. Next, our mutant killing constraint generation box generates a new θ to kill such mutant and the expression will be,

$$\begin{aligned}\theta: \quad & (x = 10) \wedge [(10 = x \wedge 10 \ngeq x) \\ & \vee(10 \neq x \wedge 10 \geq x)].\end{aligned}$$

In this case θ becomes unsatisfiable, which means, no new test case can be generated (with respect to the existing database state) which can kill the live mutant. Therefore new synthetic data needs to be generated to improve the mutation score of the test case $x = 10$. In this case, the synthetic data generation constraint ψ is generated as follows. The mutant is live because there are not enough entries in the `coffees` table to generate different entries for WHERE clauses $packets = 10$ (from the executed query) and $packets \geq 10$ from the mutant). In order to improve the mutation score of the generated test case $x = 10$, we need to have an entry in the `coffees` table which satisfies $[(packets = 10) \wedge (packets \ngeq 10)] \vee [(packets \neq 10) \wedge (packets \geq 10)]$ (again using mutant killing constraint rules (this case *ROR*) from Table 3.2). Thus, the constraint expression ψ will look like

$$\begin{aligned}\psi: \quad & \Re \wedge \ [(packets = 10 \wedge packets \ngeq 10) \\ & \vee(packets \neq 10 \wedge packets \geq 10)] \wedge pc''.\end{aligned}$$

Here, $\Re$ denotes the database schema constraint expression of `coffees` table. pc'' denotes the sub-branch conditions (extracted from PC) which depend on table attribute values (`coffees` in this case) for the current path. In our running example for the current path, we have such branch condition as $(price - discRate \leq 5)$. After analyzing the current execution path, our framework learns that the vauation of the attribute *price* comes from `coffees` table and the valuation of *discRate* comes from *distributor* table. Since ψ will create synthetic entry for the `coffees` table only, replacing the symbolic value of *discRate* with the current concrete value which is $discRate = 1$, we get our pc'' as $(price < 6)$. We use this constraint expression in ψ to generate a new synthetic data for `coffees` table.

Step 2.6: Find Satisfiable Assignment for ψ. The constraint ψ is checked for satisfiability to generate a new synthetic data. If ψ is satisfied, then the database state will be updated using the newly generated data. The updated database state will guarantee the previously generated test case to achieve high mutation score by killing the live mutants.

Table 5.5 Final Updated Table **coffees**

id	name	price	packets
1	French	5	5
2	Colombian	5	9
3	English	8	8
4	Espresso	5	10
5	abc	5	11

For instance, after solving ψ, the updated `coffees` table with newly generated synthetic data is shown in Table 5.5. With this new entry, the previously generated test case ($x = 10$) now kills the live mutant as shown in Table 5.6.

Table 5.6 Mutants and new Results for test case ($x = 10$)

Query	**Concrete Query**	**Result**
Actual q_c	SELECT * FROM coffees WHERE packets = 10	$\langle 4, Espresso, 5, 10\rangle$
Mutant q_m	SELECT * FROM coffees WHERE packets $\geq$ 10	$\langle 4, Espresso, 5, 10\rangle$, $\langle 5, abc, 5, 11\rangle$

Step 3: Explore a New Execution Path. Finally, the whole process is iterated starting from *Step 1* to generate new test cases and new data (if required) that explore new execution paths of the program. This iteration continues until all possible branches are covered.

5.3 Future Work

We will prove the correctness criterion of our approach, i.e., *Generating Minimal Set of Synthetic Data*, by proving the following theorem,

Theorem. *For any path explored by a test case t_0 with path constraint PC, if the synthetic data set generated for that path is D to achieve mutation score M, the size of the set D will be minimal.*

We also plan to evaluate the benefits of our approach from the following two perspectives:

1. What is the percentage increase in quality (where quality is attributed as both code coverage and mutation score) by the test cases generated by existing approaches like Pex [20] and SynDB [17] compared to the ones generated by our new approach in testing database applications?

2. What is percentage decrease in generating database state by our new approach compared to the ones generated by our previous work [61] while generating high quality test cases for database applications?

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

6.1 Summary

Typically, test case generation for an application relies on ensuring a high degree of (code, block or branch) coverage. Mutation testing is performed separately to assess the quality of those generated test cases. If the mutation score is low, new test cases are generated and mutation analysis is performed again. This results in unnecessary delay and overhead in identifying the high quality test cases, where quality is attributed to both coverage and mutation scores. In this work, we propose and develop a test case generation technique which addresses the above problem.

First we have proposed a framework called ConSMutate that combines coverage analysis and mutation analysis in automatic test case generation for database applications using a given database state. Our experiments show the effectiveness and practical applicability of the approach. Moreover, our framework is generic, and therefore new coverage-based and mutation generations techniques can be easily incorporated and evaluated in this framework.

Killing SQL mutants depends partially on choosing the right test cases and partially on the current database state. Since the framework relies on identifying important control-path constraints of the application and the constraints for killing mutants, the constraint generated so far may result in a satisfiable assignment that will not be able to kill all the mutants with respect to the given database state. Our framework SynConSMutate leverages our basic work [54] as discussed in Chapter 3 and generates test cases which include both program inputs and synthetic data for database applications where the database entries are absent (or insufficient).

Synthetic data sometimes cannot capture all the scenarios that might be present in the real data. Therefore, testing database application with synthetic data might overlook certain faults/scenarios that might occur while running the application using real-life data. So, there is a need of identifying test cases while reusing the real database state to the fullest has come into the picture. We propose an algorithm in chapter 5 which address this problem scenario. Our approach generates program inputs with high quality both in terms of coverage and mutation score while maximizing the usage of current database state. This will eliminate the synthetic data generation for program inputs to improve quality. Thus only minimal set of synthetic data will be generated. The approach is generic enough; therefore it can be used as a helper technique with any automated testing strategy to maximize the database state usage.

6.2 Uniqueness

Several features of this comprehensive testing strategy sets apart our approach and attributes to its uniqueness in solving a very important problem of testing database applications.

Quality. Our approach combines coverage constraints and mutation analysis to automatically generate high quality test cases for database applications.

Applicability. Being based on constraint-satisfaction, our approach does not rely on the usage of any specific application language or query language. In other words, it is applicable to any database applications.

Extensibility. Our approach is implemented in a highly modular fashion which makes it possible to include different (and newly developed) techniques in plug-and-play basis for generating path and mutation killing constraints. This makes our approach and framework relevant and applicable even when new languages and technologies are developed for realizing and testing database applications.

6.3 Discussion

Concolic Testing [37, 36] which is a variant of symbolic execution, has been proven to be an effective strategy for generating test cases automatically. The primary advantage of concolic execution over pure symbolic execution is the presence of concrete values, which can be used both to reason precisely about complex data structures as well as to simplify constraints when they go beyond the capability of the underlying constraint solver. But in practice, it has been seen that for concolic execution, the possible number of paths that must be considered symbolically is so large that the methods end up exploring only small parts of the program, and those that can be reached by short runs from the initial point, in reasonable time. Also, maintaining and solving symbolic constraints along execution paths becomes expensive as the length of the executions grows. That is, although wide, in that different program paths are explored exhaustively, symbolic and concolic techniques are inadequate in exploring the *deep* states reached only after long program executions. To overcome such limitation scenarios, techniques like *hybrid concolic testing* [68] are proposed. Hybrid concolic testing interleaves the application of random tests with concolic testing to achieve deep and wide exploration of the program state space. The interleaving strategy thus uses both the capacity of random testing to inexpensively generate deep program states through long program executions and the capability of concolic testing to exhaustively and symbolically search for new paths with a limited look ahead. In our work we use concolic execution (testing) as our coverage analysis technique. But the strategy is loosely coupled in our framework, therefore to improve efficiency in coverage analysis; we can replace concolic execution with other effective technique like hybrid concolic execution and traverse through deep program states.

Mutation Analysis in our framework plays an important role in identifying high quality test cases. The metric which is used to measure the quality of the generated test cases is called mutation score. The test cases achieve better confidence in identifying maximum programming errors as the mutation score goes higher. Our approach generates test cases and database state so that the test suite can kill all the generated mutants. Ideally, mutation score for generated test cases should achieve 100%. But in reality, we see test cases achieving 100% mutation score for very few programs (typically for simple programs). In most cases, we see some of the generated mutants cannot be killed by any of the generated test cases. This is because these mutants are semantically same as original program. They are called equivalent mutants. Therefore mutation score does achieve to 100% for those cases. *Secondly*, our framework generates a unique expression called mutant killing constraint and solves the expression in conjunction with a particular path constraint. If the expression is solvable, then the new solution will be the new test case for a path with better confidence in killing live mutants. But we have encountered scenarios where the conjunction of mutant killing constraint and path constraint becomes unsatisfiable. Constraint solver then cannot come up with new test case which has higher mutation score. Therefore, 100% mutation score cannot be achieved for those situations.

6.4 Future Directions

In modern software industry, applications are designed in multiple tiers and in multiple languages and are executed on multiple, architecturally different machines. An ideal example is Web-based applications. In such applications web components are software components which interact with each other to provide services as part of web applications. Web components are written in different languages, including Java Servlets, Java Server Pages (JSPs), JavaScripts, Active Server Pages (ASPs), PHP, and AJAX (Asynchronous JavaScript and XML).

There are many good reasons to develop Multilanguage systems. First, most algorithms are easier to implement or run more efficiently when programmed in a specific language. Therefore, a Multilanguage system is easier to program and more efficient because all its components are programmed in the most suitable language. Second, the language that is best for quickly developing an application might not be the most efficient. This forces developers to completely re-implement the final version of a system in a different language. If multiple languages are available, a selective reimplementation of only a few modules solves the efficiency problem with less programming effort. Next, it is substantially more convenient to reuse an existing component written in one language and integrate it with other components written in different languages rather than to reprogram it.

There are also good reasons why concurrent systems can benefit from multiple machine architectures. First, some architectures are optimized for efficient execution of specific languages. Although the latest CPU architectures promise uniformly good performance across many different languages, it is still true that certain languages are only available or run more efficiently on certain machines. The availability of a good implementation of a given language is more often than not the reason for preferring a particular machine. Second, some architectures have been explicitly designed for efficiently programming certain classes of problems, for example array processors. One would like to take advantage of these architectures and embed them in larger applications. Last but not the least, there is a fair amount of large and medium-grain parallelism that can be exploited in Multilanguage applications, because the modules are naturally decoupled and pursue independent subtasks. In some cases, a concurrent implementation can be substantially easier to program than a sequential one because it more naturally models the application. For instance, consider a user interface that controls and coordinates a few independent components which, in turn, interact among themselves. Programming the control flow of this application as a sequential program can be much harder and prone to errors than programming it in a concurrent way.

Several efforts have been made to test individual software components in terms of structural coverage level and fault analysis level. Even efforts have been made to perform structural coverage analysis and fault analysis in interface/integration level. These two approaches have their individual benefits and help testers to achieve confidence on an application based on individual criteria. For example, structural testing gives better confidence in terms covering all expected functional scenarios of an application whereas fault based testing allows to identify faulty behavior, if any, present in the application.

The applicability and future extension of our work are as follows. Several individual efforts have been made to develop strategies in structural testing and fault-based testing. Initially, structural testing and fault-based testing strategies were developed independently for applications with one type of software component, later old techniques are leveraged and new techniques are proposed for applications which have multiple software components written in multiple languages. But none of these techniques combine coverage analysis and fault-based analysis together while generating test cases for Multilanguage programs. Our proposed work in this thesis combines these two testing strategies together and proposes a new strategy which offers best from both the worlds. Our methodology will give higher confidence covering both normal and faulty scenarios for a given application, thus generated test cases will be qualitatively higher compared to other existing test strategies for other multi language applications.

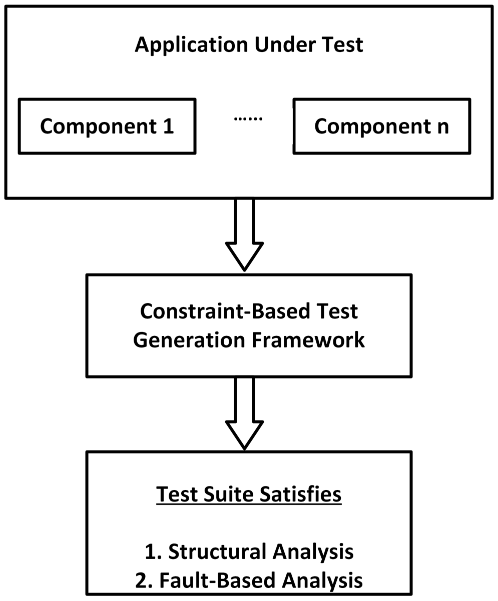

Figure 6.1 Overall Impact of Our Work

The novelties of our approach are

- it is not dependent on any particular programming language or languages, and
- the strategy works for applications which may consist of multiple software components written in multiple languages.

With the arrival of the cloud computing era, large-scale distributed systems are increasingly in use. These systems are built out of hundreds or thousands of commodity machines that are not fully reliable and can exhibit frequent failures [69, 70]. Due to this reason, todays "cloud software" (i.e., software that runs on large-scale deployments) does not assume perfect hardware reliability. Cloud software has a great responsibility to correctly recover from diverse hardware failures such as machine crashes, disk errors, and network failures.

Even if existing cloud software systems are built with reliability and failure tolerance as primary goals [71], their recovery protocols are often buggy. For example, the developers of Hadoop File System [72] have dealt with 91 recovery issues over its four years of development [73]. There are two main reasons for this. Sometimes developers fail to anticipate the kind of failures that a system can face in a real setting (e.g., only anticipate fail-stop failures like crashes, but forget to deal with data corruption), or they incorrectly design/implement the failure recovery code. There have been many serious consequences (e.g., data loss, unavailability) of the presence of recovery bugs in real cloud systems [73].

Our framework injects one-fault at a time while testing, therefore addresses solving single failures during program execution. We want to extend our work for testing cloud-based applications. Cloud software systems face frequent, multiple, and diverse failures. In this regard, we are planning to advance our approach to consider multiple failures in program execution while testing such applications. Therefore, the applicability of this approach is huge and leads to new research avenues involving concolic testing, model checking, and constraint solving for generating high quality test cases.

BIBLIOGRAPHY

[1] A. J. Offutt, Z. Jin, and J. Pan, “The dynamic domain reduction procedure for test data generation,” *Softw., Pract. Exper.*, vol. 29, no. 2, pp. 167–193, 1999.

[2] M. Natu and A. S. Sethi, “Application of adaptive probing for fault diagnosis in computer networks,” in *NOMS*. IEEE, 2008, pp. 1055–1060.

[3] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on test data selection: Help for the practicing programmer,” *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[4] R. G. Hamlet, “Testing programs with the aid of a compiler,” *IEEE Trans. Software Eng.*, vol. 3, no. 4, pp. 279–290, 1977.

[5] T. A. Budd and D. Angluin, “Two notions of correctness and their relation to testing,” *Acta Inf.*, vol. 18, pp. 31–45, 1982.

[6] K. N. King and A. J. Offutt, “A fortran language system for mutation-based software testing,” *Softw., Pract. Exper.*, vol. 21, no. 7, pp. 685–718, 1991.

[7] A. Derezinska, “Advanced mutation operators applicable in c# programs,” in *SET*, ser. IFIP, K. Sacha, Ed., vol. 227. Springer, 2006, pp. 283–288.

[8] J. Tuya, M. J. S. Cabal, and C. de la Riva, “Mutating database queries,” *Information & Software Technology*, vol. 49, no. 4, pp. 398–417, 2007.

[9] M. E. Delamaro and J. C. Maldonado, “Interface mutation: Assessing testing quality at interprocedural level,” in *SCCC*. IEEE Computer Society, 1999, pp. 78–86.

[10] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, “Integration testing using interface mutations,” in *Proceedings of International Symposium on Software Reliability Engineering (ISSRE 96.* Society Press, 1996, pp. 112–121.

[11] ——, “Interface mutation: An approach for integration testing,” *IEEE Trans. Software Eng.*, vol. 27, no. 3, pp. 228–247, 2001.

[12] W. K. Chan, S. C. Cheung, and T. H. Tse, “Fault-based testing of database application programs with conceptual data model,” in *QSIC*. IEEE Computer Society, 2005, pp. 187–196.

[13] J. Tuya, M. J. Surez-cabal, and C. D. L. Riva, “Sqlmutation: A tool to generate mutants of sql database queries,” 2006.

[14] C. Zhou and P. G. Frankl, “Mutation testing for java database applications,” in *ICST*. IEEE Computer Society, 2009, pp. 396–405.

[15] H. Shahriar and M. Zulkernine, “Music: Mutation-based sql injection vulnerability checking,” in *QSIC*, H. Zhu, Ed. IEEE Computer Society, 2008, pp. 77–86.

[16] M. Emmi, R. Majumdar, and K. Sen, “Dynamic test input generation for database applications,” in *ISSTA*, D. S. Rosenblum and S. G. Elbaum, Eds. ACM, 2007, pp. 151–162.

[17] K. Pan, X. Wu, and T. Xie, “Guided test generation for database applications via synthesized database interactions,” UNC Charlotte, Tech. Rep., 2012.

[18] K. Taneja, Y. Zhang, and T. Xie, “MODA: Automated test generation for database applications via mock objects,” in *ASE*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 2010, pp. 289–292.

[19] C. Li and C. Csallner, “Dynamic symbolic database application testing,” in *DBTest*, S. Babu and G. N. Paulley, Eds. ACM, 2010.

[20] N. Tillmann and J. de Halleux, “Pex: White box test generation for .net,” in *TAP*, ser. Lecture Notes in Computer Science, B. Beckert and R. Hähnle, Eds., vol. 4966. Springer, 2008, pp. 134–153.

[21] R. D. Millo, W. McCracken, R. Martin, and J. Passafiume, “Software testing and evaluation,” *Benjamin/Cummins*, 1987.

[22] D. L. Bird and C. U. Munoz, “Automatic generation of random self-checking test cases,” *IBM Systems Journal*, vol. 22, no. 3, pp. 229–245, 1983.

[23] A. J. Offutt and J. H. Hayes, “A semantic model of program faults,” in *ISSTA*, 1996, pp. 195–200.

[24] C. Pacheco and M. D. Ernst, “Eclat: Automatic generation and classification of test inputs,” in *ECOOP*, ser. Lecture Notes in Computer Science, A. P. Black, Ed., vol. 3586. Springer, 2005, pp. 504–527.

[25] J. E. Forrester and B. P. Miller, “An empirical study of the robustness of windows nt applications using random testing,” in *Proceedings of the 4th USENIX Windows System Symposium*, 2000, pp. 59–68.

[26] R. Ferguson and B. Korel, “The chaining approach for software test data generation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 1, pp. 63–86, Jan. 1996. [Online]. Available: http://doi.acm.org/10.1145/226155.226158

[27] B. Korel and A. M. Al-Yami, “Assertion-oriented automated test data generation,” in *ICSE*, H. D. Rombach, T. S. E. Maibaum, and M. V. Zelkowitz, Eds. IEEE Computer Society, 1996, pp. 71–80.

[28] C. Boyapati, S. Khurshid, and D. Marinov, “Korat: automated testing based on java predicates,” in *ISSTA*, 2002, pp. 123–133.

[29] T. Xie, D. Marinov, and D. Notkin, “Rostra: A framework for detecting redundant object-oriented unit tests,” in *ASE*. IEEE Computer Society, 2004, pp. 196–205.

[30] C. Pacheco and M. D. Ernst, “Randoop: feedback-directed random testing for java,” in *OOPSLA Companion*, R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., Eds. ACM, 2007, pp. 815–816.

[31] M. Boshernitsan, R.-K. Doong, and A. Savoia, "From daikon to agitator: lessons and challenges in building a commercial tool for developer testing," in *ISSTA*, L. L. Pollock and M. Pezzè, Eds. ACM, 2006, pp. 169–180.

[32] D. Beyer, A. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar, "Generating tests from counterexamples," in *ICSE*, A. Finkelstein, J. Estublier, and D. S. Rosenblum, Eds. IEEE Computer Society, 2004, pp. 326–335.

[33] W. Visser, C. S. Pasareanu, and S. Khurshid, "Test input generation with java pathfinder," in *ISSTA*, G. S. Avrunin and G. Rothermel, Eds. ACM, 2004, pp. 97–107.

[34] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *TACAS*, ser. Lecture Notes in Computer Science, N. Halbwachs and L. D. Zuck, Eds., vol. 3440. Springer, 2005, pp. 365–381.

[35] C. Csallner and Y. Smaragdakis, "Check 'n' crash: combining static checking and testing," in *ICSE*, G.-C. Roman, W. G. Griswold, and B. Nuseibeh, Eds. ACM, 2005, pp. 422–431.

[36] K. Sen, "Concolic testing," in *ASE*, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds. ACM, 2007, pp. 571–572.

[37] ——, "Dart: Directed automated random testing," in *Haifa Verification Conference*, ser. Lecture Notes in Computer Science, K. S. Namjoshi, A. Zeller, and A. Ziv, Eds., vol. 6405. Springer, 2009, p. 4.

[38] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Software Eng.*, vol. 17, no. 9, pp. 900–910, 1991.

[39] T. A. Budd and A. S. Gopal, "Program testing by specification mutation," *Comput. Lang.*, vol. 10, no. 1, pp. 63–73, 1985.

[40] S.-W. Kim, J. A. Clark, and J. A. McDermid, "Investigating the effectiveness of object-oriented testing strategies using the mutation method," *Softw. Test., Verif. Reliab.*, vol. 11, no. 3, pp. 207–225, 2001.

[41] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Softw. Test., Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, 2005.

[42] A. J. Offutt and K. N. King, "A fortran 77 interpreter for mutation analysis," in *PLDI*, R. L. Wexelblat, Ed. ACM, 1987, pp. 177–188.

[43] H. A. Richard, R. A. Demillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford, "Design of mutant operators for the c programming language," Purdue University, Tech. Rep., 1989.

[44] H. Shahriar and M. Zulkernine, "Mutation-based testing of buffer overflow vulnerabilities," in *COMPSAC*. IEEE Computer Society, 2008, pp. 979–984.

[45] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978. [Online]. Available: http://dx.doi.org/10.1109/C-M.1978.218136

[46] S. Ghosh, P. Covindarajan, and A. P. Mathur, "Tds: a tool for testing distributed component-based applications," in *Mutation testing for the new century*, W. E. Wong, Ed. Norwell, MA, USA: Kluwer Academic Publishers, 2001, pp. 103–112. [Online]. Available: http://dl.acm.org/citation.cfm?id=571305.571328

[47] S. Ghosh and A. P. Mathur, "Interface mutation to assess the adequacy of tests for components and systems," in *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, ser. TOOLS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 37–. [Online]. Available: http://dl.acm.org/citation.cfm?id=832261.833250

[48] M. E. Delamaro, J. C. Maldonado, A. Pasquini, and A. P. Mathur, "Interface mutation test adequacy criterion: An empirical evaluation," *Empirical Softw. Engg.*, vol. 6, no. 2, pp. 111–142, Jun. 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1011429104252

[49] A. M. R. Vincenzi, J. C. Maldonado, E. F. Barbosa, and M. E. Delamaro, "Unit and integration testing strategies for c programs using mutation-based criteria

(abstract only)," in *Mutation testing for the new century*, W. E. Wong, Ed. Norwell, MA, USA: Kluwer Academic Publishers, 2001, pp. 45–. [Online]. Available: http://dl.acm.org/citation.cfm?id=571305.571315

[50] A. Derezińska and A. Szustek, "Cream- a system for object-oriented mutation of c# programs," Warsaw University of Technology, Warszawa, Poland, techreport, 2007.

[51] J. Tuya, M. J. S. Cabal, and C. de la Riva, "Full predicate coverage for testing sql database queries," *Softw. Test., Verif. Reliab.*, vol. 20, no. 3, pp. 237–288, 2010.

[52] D. Chays, J. Shahid, and P. G. Frankl, "Query-based test generation for database applications," in *DBTest*, L. Giakoumakis and D. Kossmann, Eds. ACM, 2008, p. 6.

[53] ——, "Query-based test generation for database applications," in *DBTest*, L. Giakoumakis and D. Kossmann, Eds. ACM, 2008, p. 6.

[54] T. Sarkar, S. Basu, and J. S. Wong, "Consmutate: Sql mutants for guiding concolic testing of database applications," in *ICFEM*, ser. Lecture Notes in Computer Science, T. Aoki and K. Taguchi, Eds., vol. 7635. Springer, 2012, pp. 462–477.

[55] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker, "An AGENDA for testing relational database applications," *Softw. Test., Verif. Reliab.*, vol. 14, no. 1, pp. 17–44, 2004.

[56] K. Pan, X. Wu, and T. Xie, "Database state generation via dynamic symbolic execution for coverage criteria," in *DBTest*, G. Graefe and K. Salem, Eds. ACM, 2011, p. 4.

[57] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid, "Query-aware test generation using a relational constraint solver," in *ASE*. IEEE, 2008, pp. 238–247.

[58] C. Binnig, D. Kossmann, and E. Lo, "Multi-rqp: generating test databases for the functional testing of oltp applications," in *DBTest*, L. Giakoumakis and D. Kossmann, Eds. ACM, 2008, p. 5.

[59] W.-T. Tsai, D. Volovik, and T. F. Keefe, "Automated test case generation for programs specified by relational algebra queries," *IEEE Trans. Software Eng.*, vol. 16, no. 3, pp. 316–324, 1990.

[60] D. Willmor and S. M. Embury, "An intensional approach to the specification of test cases for database applications," in *ICSE*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 2006, pp. 102–111.

[61] T. Sarkar, S. Basu, and J. Wong, "Synconsmutate: Concolic testing of database applications via synthetic data guided by sql mutants," in *10th International Conference on Information Technology : New Generations (ITNG), "to appear"*, 2013.

[62] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *ICSE*, V. R. Basili, R. A. DeMillo, and T. Katayama, Eds. IEEE Computer Society / ACM Press, 1993, pp. 100–107.

[63] H. Andrade, T. Kurc, A. Sussman, and J. Saltz, "Optimizing the execution of multiple data analysis queries on parallel and distributed environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, pp. 520–532, 2004.

[64] L. Weng, Ü. V. Çatalyürek, T. M. Kurç, G. Agrawal, and J. H. Saltz, "Optimizing multiple queries on scientific datasets with partial replicas," in *GRID*. IEEE, 2007, pp. 259–266.

[65] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Software Eng.*, vol. 8, no. 4, pp. 371–379, 1982.

[66] R. Ahmed, A. W. Lee, A. Witkowski, D. Das, H. Su, M. Zaït, and T. Cruanes, "Cost-based query transformation in oracle," in *VLDB*, U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, Eds. ACM, 2006, pp. 1026–1036.

[67] S. Chaudhuri, "Review - of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers," *ACM SIGMOD Digital Review*, vol. 2, 2000.

[68] R. Majumdar and K. Sen, "Hybrid concolic testing," in *ICSE*. IEEE Computer Society, 2007, pp. 416–426.

[69] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," in *FAST*. USENIX, 2007, pp. 17–28.

[70] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you?" 2007.

[71] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, Eds. ACM, 2010, pp. 143–154.

[72] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system."

[73] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "Fate and destini: a framework for cloud recovery testing," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 18–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972482

[74] L. Giakoumakis and D. Kossmann, Eds., *Proceedings of the 1st International Workshop on Testing Database Systems, DBTest 2008, Vancouver, BC, Canada, June 13, 2008*. ACM, 2008.