ABSTRACT

TANEJA, KUNAL. Quality Assurance of Database Centric Applications. (Under the direction of Tao Xie.)

Software faults cost USA's economy over 59 billion dollars. The cost can be brought down by making software testing more effective in finding and fixing faults quickly. Software developers test an application by writing tests with inputs that achieve high structural code coverage. However, manual testing is labor intensive and time consuming. To reduce the efforts of manual software testing, there exist tools for automated test generation. These tools can generate tests that achieve high code coverage. The generated tests can be used to find faults in the software application under test. In addition, these tests can be used as regression tests, i.e., when changes are made to the software application under test, these tests can be executed on the modified application to find regression faults.

Database Centric Applications (DCAs) are getting more and more popular in enterprise computing. DCAs consist of a front-end application (FA) that interacts with a back-end database (DB). Testing of DCAs not only requires testing the FA with inputs that achieve high code coverage but also preparing necessary states in its DB to cover various branches in the FA, in short as FA branches, that are dependent on the DB. When existing test generation tools are used to test the DCAs, the tools can generate tests for the FA but cannot generate inputs for the DB. As a result, various FA branches that are dependent on the DB cannot be covered. In addition, the tools are not efficient and effective for regression testing of the FA of the DCAs.

This dissertation addresses various problems in test generation for DCAs. First, often testing of DCAs is outsourced to testing centers and is conducted by test engineers there. When proprietary DCAs are released, their DB should also be made available to test engineers. However, different data privacy laws prevent organizations from sharing the records in the DB with test centers because the DB can contain sensitive information. As a result, various FA branches that depend on the DB cannot be covered leading to more regression faults undetected. Second, even if the DB can be released to the test engineers, the DB has insufficient (or no) records in it for effective regression testing of the DCA. As a result, for effective regression testing of a DCA, a test generation tool not only needs to generate inputs for the FA but also generate inputs for the DB to cover various FA branches that are dependent on the DB. Hence, a test generation tool needs to bridge the gap between the FA and the DB. Third, existing test generation tools generate tests to achieve high code coverage of the FA but not specifically to find behavioral differences between the two versions of the FA. As a result, these approaches are ineffective and inefficient for regression test generation.

In this dissertation, we propose a framework that addresses the preceding problems in test generation for DCAs. First, we present an approach, called PRIEST, for anonymizing the records in the DB of a DCA such that the anonymized DB can be released to the test engineers (conducting the regression

testing) without leaking sensitive information in the DB. With PRIEST, organizations can balance the level of privacy with needs of regression testing. Second, we present an approach, called MODA, that facilitates the generation of inputs for the DB of a DCA to cover various FA branches that are dependent on the DB. To generate tests efficiently, MODA can use the existing records in the DB or the anonymized DB generated by PRIEST. Third, we present approaches, called DiffGen and eXpress, for effective and efficient regression test generation of the FA of a DCA.

Quality Assurance of Database Centric Applications

by
Kunal Taneja

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2013

APPROVED BY:

Laurie Williams	James Tuck
Emerson Murphy-Hill	Tao Xie Chair of Advisory Committee

DEDICATION

To my parents.

BIOGRAPHY

Kunal Taneja was born in Agra, India. He completed his Bachelor of Technology from Indian Institute of Technology, Guwahati in 2004 and his Master of Science from North Carolina State University in 2009, both in Computer Science. His primary research interest is in software engineering with a primary goal to develop techniques and tools that can help in developing high-quality software. During his PhD program, his internship experiences included Avaya Labs, Basking Ridge, NJ, USA (Summer 2007), IBM Research, Hawthorne, NY, USA (Summer 2008), the FDA, Silver Spring, MD, USA (Summer 2009), and Accenture Technology Labs, Chicago, IL, USA (Summer 2010 and 2011). He is a recipient of a University Outstanding Teaching Assistant award in 2006. He is a student member of ACM.

ACKNOWLEDGEMENTS

First of all, I would like to thank my adviser, Dr Tao Xie, for his invaluable guidance during the course of my PhD. His guidance made it possible for me to publish parts of this dissertation in top conferences such as ISSTA 2011 and FSE 2011. His guidance has helped me in improving my all-round research skills including technical writing, presentation, and independent thinking. These skills will be useful throughout my career. Second, I would like to thank my collaborator, Dr Mark Grechanik, for his help in improving parts of the dissertation and making it possible to publish in a top conference (FSE 2011). Third, I would like to thank my dissertation committee including Dr Laurie Williams, Dr Emerson Murphy-Hill, and Dr James Tuck for their comments and suggestions in improving this dissertation.

I thank my internship mentors in the industry, who have immensely helped me expand my research to a broader scope: Dr Mark Grechanik (Accenture Technology Labs), Dr Yi Zhang (FDA), Dr Amit Pradkar (IBM Research), and Dr Joann Ordille (Avaya Labs Research).

I would like to deeply thank all my dear friends and colleagues at NC State for their support when the stress of PhD took the better of me and for making Raleigh a home. Special thanks to DaYoung Lee, Suresh Thummalapena, Padmashree Ravindran, Abhik Sarkar, Ajeet, Pawandeep Singh, Nikhil Deshpandey, Shailen Mishra, Sundar Srinivas, Mithun Acharya, Madhuri Marri, Jeehyun Hwang, Xusheng Xiao, Yoonki Song, Prasanth Anbalagan, Laxmi Ramachandran, John Majikes, and Rahul Pandita.

I thank my parents for having faith in me and for having patience for all these years. Last but not the least I would like to thank my wife, Peehoo Dewan, for believing in me and for providing me the kind of support that I needed while writing this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Testing of Software Applications

Software testing is an important part of software development lifecycle as it helps in finding and fixing faults in a software application before it is released. According to a study [80] by National Institute of Standards and Technology, software faults cost the United States of America's economy an estimated 59.5 billion dollars annually (about 0.6% of the gross domestic product). The study concludes that, although the faults cannot be completely removed, more than a third of these costs could be eliminated by an improved testing infrastructure. Hence, software testing is the largest growing IT service [56], growing at the rate of 9.5% and is projected to reach 56 billion dollars by 2013.

Software testing is labor intensive and time consuming. The total cost of quality assurance (including testing, debugging, and verification) is estimated to range from 50 to 75 percent of the total development cost [49]. Software testers (or developers) test a software application by executing the application with various tests and checking whether the application produces correct outputs for those tests. Since the number of inputs of an application can be infinite, various coverage criteria [126] have been proposed to select a finite number of tests for effective testing.

To reduce the cost of software testing, there exist various coverage-based approaches for automated test generation. These approaches systematically (or randomly) generate tests for a software application to achieve high code coverage. These tests can be used to detect faults in the existing version of the software application. In addition, after the developers modify the software application, developers can conduct regression testing by rerunning the generated tests (for the original software application) to assure that no regression faults are introduced. The outputs produced by the modified version can be automatically compared with the outputs produced by the original version to detect behavioral differences [1]. Software developers can inspect these behavior differences to find out whether they are intended or

[1] A behavioral difference between two versions of a software application can be reflected by the difference between the observable outputs produced the execution of the same test on the two versions.

```
void testDCA(int k ){
        ...
1       for (int i=0; i<k; i++)
2             n++;
3       if (n==15){
4             SQLStatement stmt = new SQLStatement(
                  "SELECT name FROM Patients WHERE disease == Diabetes")
5             ResultSet rs = stmt.executeReader();
6             while(rs.hasNext()){
7                   if(rs.getInt("age") > 65){ // rs.getInt("age") >= 65
8                         ...
9                   }
10            }
11      }
}
```

Figure 1.1: Code of an example DCA.

due to regression faults introduced while modifying the software application.

1.1.1 Testing for Database Centric Applications

Database Centric Applications (DCAs) are getting more and more popular in enterprise computing. DCAs consist of a front-end application (FA) that interacts with a back-end database (DB). Test generation of DCAs not only requires generating tests for the FA that achieve high code coverage of the FA but also generating tests (records) for the DB to cover various branches in the FA, in short as FA branches, that are dependent on the DB. Figure 1.1 shows an example DCA. The DCA selects records from a Table `Patients` (at Line 4) in the DB for all patients with `diabetes`. The DCA then processes the records that have patients with age more than 65 (the `if` statement at Line 7). However, the DCA contains a fault in which a developer by mistake puts > instead of >= in the `if` condition at Line 7. The execution of the `true` branch of the `if` statement is dependent on the state of the DB. Hence, to find the fault at Line 7, a test generation approach not only needs to effectively execute tests that have the input $k = 15$ (so that the true branch of the `if` statement at Line 3 is taken) for the FA but also need to insert records in the table `Patient` (in the DB) with $age = 65$ and $disease = diabetes$, to cover the true branch of the `if` statement at Line 7 to find the fault at Line 7.

1.2 Problems

There are existing state-of-the-art approaches for test generation of software applications that can generate inputs for the FA (or other non-DCAs) of a DCA. However, these approaches cannot be used to generate tests for the DB. In addition, these approaches are not effective or efficient for regression testing of the FA (or a non-DCA) of the DCA. In this dissertation, we address various problems of automated

test generation for DCAs. In this section, we present the problems that we address in this dissertation and how we address the problems.

1.2.1 P1: Data Anonymization in Presence of Privacy Laws

DCAs are common in enterprise computing, and they use nontrivial databases. Testing these enterprise applications is increasingly outsourced to test centers [79, 96] to achieve higher quality and lower software maintenance costs. When releasing these proprietary DCAs to external test centers, it is desirable for DCA owners to make their DBs available to test engineers in these test centers, so that they can perform testing using original data in the DBs. However, since sensitive information cannot be disclosed to external organizations, testing is often performed with synthetic input data. For instance, if values of the field `Nationality` are replaced with the generic value `Human`, DCAs may execute some paths that result in exceptions or miss certain paths [47]. As a result, test centers report worse test coverage (such as code coverage) and regression-fault detection capability, thereby reducing the quality of the applications and obliterating the benefits of test outsourcing [74]. To address the preceding problem, we propose an approachfor application-aware DB anonymization, using which organizations can balance the level of privacy with the needs of testing.

1.2.2 P2: Input Generation for DB

Often at the time of testing of a DCA, the DB does not have specific records that are required to cover certain FA branches. As a result, tests generated for the FA may not be able to cover certain FA branches that are dependent of the DB (as described in Section 1.1.1). To address the preceding issue, we present an approach that can effectively generate tests for DB such that various FA branches that depend on the DB can be covered.

1.2.3 P3: Effectiveness of Regression Testing of the FA

Regression testing of the FA of a DCA is carried out by executing an existing regression test suite (that was developed for a previous version of the FA) against the current version of the FA. The failing tests (that pass for the previous version) indicate behavioral differences[2] between the two versions of the FA. Software developers can then inspect the failing tests to find whether the tests fail due to intended changes or regression faults. As a result, effectiveness of regression testing depends on the effectiveness of the regression test suite in finding behavioral differences. However, the existing regression test suite may not be effective in finding behavioral differences making software maintenance error prone. Hence, there is a strong need of regression tests that are effective in finding behavioral differences. New tests can be generated using existing test generation tools [21, 26, 45, 46, 55, 63, 95, 112, 115]. However, these

[2] A behavioral difference between two versions of an FA can be reflected by the difference between the observable outputs produced the execution of the same test on the two versions.

tools aim at generating tests that can achieve high structural coverage and achieving high structural coverage is not sufficient for finding behavioral differences. To address the preceding issue, we propose an approach that bridges the gap between effectively finding behavioral differences and coverage-based test generation tools.

1.2.4 P4: Efficiency of Regression Testing

Previous studies [14, 15] have found that faults found in the field cost more than five times as much to correct as those faults corrected earlier. Hence, it is desirable to detect regression faults as quickly as possible to reduce the cost involved in fixing them. One existing solution is continuous testing, which runs an existing test suite to quickly find regression faults as soon as code changes are saved. However, the effectiveness of continuous testing depends on the capability of the existing test suite for finding behavioral differences across versions. The existing test suite might not be able to detect behavioral differences as it is usually created (or generated) without taking into consideration the changes to be made in the future. Then the existing test suite can be augmented using existing test generation techniques to improve the capability of the test suite in terms of detecting behavioral differences. Existing test generation techniques such as path-exploration-based test generation (PBTG) [21, 26, 45, 46, 63, 95, 115] and search-based test generation [55, 112] focus their efforts on increasing structural coverage and do not specifically focus on detecting behavioral differences between two versions of an FA. As a result, these techniques are inefficient for regression test generation, even with increasing computing power thanks to multi-core architectures and cloud computing. To address the preceding issue, we propose an approach that targets at reducing the cost of test generation so that it focuses specifically on detecting behavioral differences. As a result, behavioral differences are likely to be detected more efficiently.

1.3 Framework

In this dissertation, we present a framework that addresses the preceding problems in testing DCAs. Our framework helps in testing of DCAs in the following ways:

- **Testing in presence of privacy laws.** The state-of-the-art techniques for test generation can be applied to a DCA if the DB of the DCA has enough records to cover various FA branches. However, even if there are sufficient records in the DB, the records often cannot be released to the testing team due to privacy concerns. To address the preceding issue, our framework consists of an approach for data anonymization called PRIEST.

- **Bridging the gap between the FA and the DB of a DCA.** If the records in the DB are not sufficient for covering certain FA branches, a test generation tool needs to generate tests containing inputs for both FA and DB. To generate tests for both the FA and the DB, a test generation tool

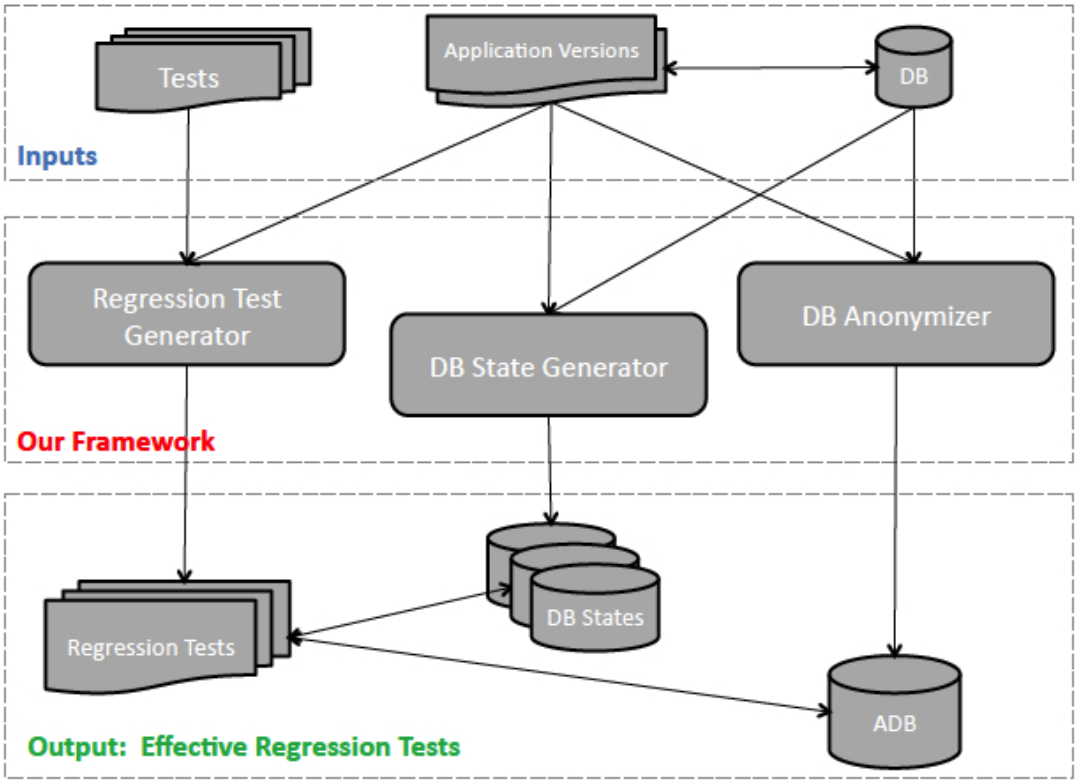


Figure 1.2: Our framework for regression testing of DCAs.

needs to bridge the gap between the FA and the DB. To address the problem, our framework consists of an approach, called MODA, that mocks the DB such that a test generation tool is able to generate tests for the FA and DB. As a result, any test generation approach for a non-DCA can be applied to a DCA.

- **Testing of changes[3] made to the FA of a DCA.** Our framework consists of two approaches, called DiffGen and eXpress, for effective and efficient testing of changes made to the FA of a DCA. These approaches are applicable to non-DCA applications or the FA of a DCA. However, these approaches can be used for regression testing of the whole DCA when used with MODA or PRIEST.

Figure 1.2 show the overview of our framework. The framework takes two versions of the FA, the existing test suite, and the DB as input and generates effective regression tests for the DCA. These tests include tests for both the FA and DB of the DCA. In addition, our framework can anonymize the existing records in the DB if it is not desirable to release sensitive records in the DB to the testing team. Our framework consists of three major components: Regression Test Generator (RTG), Database State Generator (DSG), and Database Anonymizer (DA). RTG consists of two approaches, DiffGen and eXpress, for generating regression tests for the FA. DiffGen, and eXpress address the problems P3 and P4, respectively. RTG takes two versions of the FA and generates a regression test suite that finds behavioral differences between two versions of the FA. If an existing test suite is present, RTG can use it to efficiently generate tests for finding behavioral differences between two versions of the FA. The DSG component uses the schema of the DB and the two versions of the FA to generate inputs for the DB so that various FA branches that are dependent on the DB can be covered, addressing Problem P2. The component DA anonymizes the data to generate an anonymized database (ADB), using which organizations can balance the level of privacy with the needs of regression testing, addressing Problem P1. To efficiently generate inputs for DB, our DSG component can use either the existing records in the DB or the records in the ADB generated by the DA component.

1.4 Contributions

This dissertation makes the following contributions:

- **An approach for effective testing in presence of privacy laws.** We design and implement an approach, called PRIEST, for effective regression testing of the FA of the DCA under test in presence of privacy laws. PRIEST enables business analysts to balance data privacy with test coverage. We

[3]In this dissertation, we assume that the schema of the DB does not change between the two versions. If the schema changes, our approach can use the new schema version. Testing of changes made to the schema of a DB is a limitation of this the work in this dissertation.

evaluate PRIEST using three open-source Java DCAs and one large Java DCA that handles logistics of one of the largest supermarket chains in Spain. We show that with PRIEST, test coverage of regression tests can be preserved at a higher level by pinpointing database attributes that should be anonymized based on their effect on the corresponding DCAs.

- **An approach for generation of DB states.** We design and implement an approach, called MODA [107], for generation of DB inputs such that various FA branches, that are dependent on the DB can be covered. We have conducted two empirical evaluations to assess the effectiveness of MODA: one on a large real-world medical device and the other on an open source DCA. Results of the evaluations demonstrate that our approach can achieve higher code coverage on the FAs of the DCAs than existing test generation tools.

- **An approach for effective regression test generation.** We propose an approach, called DiffGen [105], for generating regression tests that help in detecting behavioral differences between two versions of a given software application by checking observable outputs and receiver object states. We evaluate our approach on detecting behavioral differences between eight subjects (taken from a variety of sources) and their versions. The experimental results show that our approach can effectively detect seeded faults that cannot be detected by state-of-the-art techniques [41] based on achieving structural coverage on either version separately.

- **An approach for efficient regression test generation.** We propose an approach called eXpress [106] for efficient generation of regression tests. To optimize the search strategy of a PBTG technique, eXpress prunes various program paths whose execution guarantees not to find regression faults. As a result, behavioral differences are found efficiently by the PBTG technique with eXpress than without eXpress. We have implemented our eXpress approach in a tool as an extension for Pex [110], an automated structural testing tool for .NET developed at Microsoft Research. We have conducted experiments on 67 versions (in total) of four programs with two from the Subject Infrastructure Repository (SIR) [35] and two from real-world open source projects. Experimental results show that Pex using eXpress requires about 36% less amount of time (on average) to detect behavioral differences than without using eXpress. In addition, Pex using eXpress detects four behavioral difference that could not be detected without using eXpress (within a time bound).

1.5 Outline

The remainder of this dissertation is organized as follows. Chapter 2 describes our PRIEST approach, Chapter 3 describes our MODA approach. Chapter 4 describes our eXpress approach. Chapter 5 describes our DiffGen approach. Chapter 6 describes suggestions for future work. Chapter 6 concludes

with a summary of our contributions and lessons learned.

Chapter 2

Testing Database-Centric Applications In Age of Data Privacy

2.1 Introduction

Large organizations today face many challenges when engineering software applications. Particularly challenging is the fact that many applications work with existing databases that contain confidential data. A large organization, such as a bank, insurance company, or government agency, typically hires an external company to develop or test a new custom software application [96]. However, recent data protection laws and regulations [73] around the world prohibit data owners to easily share confidential data with external software service providers.

Database-centric applications (DCAs) are common in enterprise computing, and they use nontrivial databases [62]. When releasing these proprietary DCAs to external test centers, it is desirable for DCA owners to make their databases available to test engineers, so that they can perform testing using *original data*. However, since sensitive information cannot be disclosed to external organizations, testing is often performed with *synthetic input data*. For instance, if values of the field *Nationality* are replaced with the generic value "`Human`," DCAs may execute some paths that result in exceptions or miss certain paths [47]. As a result, test centers report worse test coverage (such as code coverage) and fewer uncovered faults, thereby reducing the quality of applications and obliterating benefits of test outsourcing [74].

Automatic approaches for test data generation [26, 32, 44, 71, 107] partially address this problem by generating synthetic input data that lead program execution toward untested statements. However, one of the main issues for these approaches is how to generate synthetic input data with which test engineers can achieve good code coverage. Using original data enables different approaches in testing and privacy to produce higher-quality synthetic input data [3] [42, page 42], thus making original data important for test outsourcing.

A fundamental problem in test outsourcing is how to allow a DCA owner to release its private data

with guarantees that the entities in this data (e.g., people, organizations) are protected at a certain level while retaining testing efficacy. Ideally, sanitized data (sanitized data or anonymized data is the original data after anonymization. We use the two terms interchangeably throughout this chapter) should induce execution paths that are similar to the ones that are induced by the original data. In other words, when data is sanitized, information about how DCAs use this data should be taken into consideration. In practice, this consideration rarely happens; previous work [47] showed that a popular data anonymization algorithm, called *k*–anonymity, seriously degrades test coverage of DCAs.

Naturally, different DCAs have different privacy goals and levels of data sensitivity – privacy goals are more relaxed for a DCA that manages a movie ticket database than for a DCA that is used within banks or government security agencies. Applying more relaxed protection to databases is likely to result in greater test coverage since a small part of the databases is anonymized; conversely, stricter protection makes it more difficult to outsource testing. The latter is the result of two conflicting goals: making testing as realistic as possible and hiding the original data from testers who need this data to make testing effective. Balancing these goals, i.e., to anonymize data while preserving test coverage of DCAs that use this data is emerging to be an important problem.

Given the importance of this problem, it may be surprising that there exists little prior research on this topic. There may be two main reasons for the lack of prior research. First, elaborate data privacy laws are a new phenomenon, and many of these laws [73, 108] have been introduced after the year 2000. Second, it is only in the past decade that applications are increasingly being tested by third-party specialized software service providers, which are also called test centers. Numerous test centers have emerged and often offer lower cost and higher quality when compared to in-house testing. In 2007, the test outsourcing market was worth more than USD 25 billion and growing at 20% annually, making test outsourcing the fastest growing segment of the application services market [9, 30].

To address this issue, we offer a novel approach, *PRIvacy Equalizer for Software Testing (PRIEST)* that combines a new data privacy framework with program analysis enabling business analysts to determine how anonymizing values of database attributes affects test coverage. With PRIEST, organizations can balance the goals of preserving test coverage, while releasing DCAs to external test centers with a controlled disclosure of sensitive information. The source code for PRIEST as well as its illustration video are publicly available[1]. To the best of our knowledge, there exists no prior approach that addressed the problem that we pose in this chapter. In summary, PRIEST makes the following main contributions:

- We create a new privacy framework (described in Section 2.4) that includes a novel combination of guessing anonymity-based privacy metrics and a technique of data swapping anonymization to enable organizations to keep original values in sanitized data; keeping such values is important for improving the effectiveness of testing.

- We design and implement a technique using program analysis for determining how values of

[1] http://www.privacytesting.org

database attributes affect test coverage of DCAs that use this data (see Section 2.3.3).

- We combine our privacy framework with this technique in PRIEST to enable business analysts to balance data privacy with test coverage.
- We evaluate PRIEST using three open-source Java DCAs and one large Java DCA that handles logistics of one of the largest supermarket chains in Spain. We show that with PRIEST, test coverage can be preserved at a higher level by pinpointing database attributes that should be anonymized based on their effect on corresponding DCAs.

2.2 The Problem

In this section, we provide the necessary background on how DCAs interact with databases, show how sanitizing data affects test coverage of DCAs, describe the state of the art and practice, and formulate the problem statement.

2.2.1 Background

Majority of enterprise-level DCAs use programs that are written in general-purpose programming languages and relational databases to maintain large amounts of data. A primary way for these programs to communicate with databases is to use *call-level interfaces*, which allow DCAs to access database engines through standardized *application programming interfaces (APIs)*, e.g., Java DataBase Connectivity (JDBC). Using JDBC, programs pass SQL queries as strings to corresponding API calls to be sent to databases for execution.

Once these queries are executed, values of attributes of database tables are returned to DCAs using JDBC `RecordSet` objects and these values are stored in variables of the DCAs, which in turn use these variables as part of their application logic. In this way, values from a database may be used in branch decisions, and these values may therefore affect the subsequent execution of the DCAs. Depending on returned values, different paths can be taken in DCAs, and subsequently these values affect test coverage. Hence removing certain classes of values in database attributes may make some branches and statements in DCAs unreachable.

To see how anonymization affects test coverage of DCAs, consider a fragment of code shown in Figure 2.1. After executing JDBC API calls that submit an SQL query to a database and obtain the object `recordset`, the values of the attributes `Age`, `Nationality`, and `Disease` are put in the corresponding variables of the DCA `nationality`, `age`, and `disease`, respectively. If the values of the attribute nationality are replaced with the generic value "`Human`,", the function call `f()` becomes unreachable.

```
recordset = db.execSql("SELECT * FROM TblPatient");
nationality = recordset.GetAttribute( i );
age = recordset.GetAttribute( j );
disease = = recordset.GetAttribute( "Disease" );
if (nationality=="Palauan" && age>60) f(disease);
```

Figure 2.1: An illustrative example that shows how replacing values of database attributes `Nationality` and `Age` of patients can make the function `f` unreachable.

Certain classes of values of database attributes that contain non-sensitive information should be anonymized. These attributes, also called *quasi-identifiers (QI)* often contain information that can be linked with other data to infer sensitive information about entities (i.e., people, objects). For example, given the values of the QIs `Race`, `Sex`, `Height`, `ZipCode` and the attribute that contains sensitive data about diseases, it is possible to link a person to specific diseases, provided that the values of these QIs uniquely identify this person.

Existing data anonymization approaches are centered on creating models for privacy goals and developing algorithms for achieving these goals using particular anonymization techniques [29]. One of the most popular privacy goals is $k-$anonymity [90], where each entity in the database must be indistinguishable from $k-1$ others. Anonymization approaches use different anonymization techniques including suppression, where information (e.g., nationality) is removed from the data, and generalization, where information (e.g., age) is coarsened into sets (e.g., into age ranges) [29]. These and other techniques modify or suppress values of attributes, and a common side-effect of these modifications is non-covered statements in DCAs that are otherwise covered with the original data.

2.2.2 State of the Art and Practice

After interviewing professionals at IBM, Accenture, two large health insurance companies, a biopharmaceutical company, two large supermarket chains, and three major banks, we found that current test data anonymization is manual, laborious, and error-prone. In a few cases, client companies outsource testing using an especially expensive and cumbersome testing procedure called *cleanroom testing*, where DCAs and databases are kept on company premises in a physically secured environment [47]. Typically, business analysts and test managers from outsourcing companies come to the cleanrooms of their client companies to evaluate their clients' DCAs and to plan work. However, when better options to access data are not available, testers from outsourcing companies are also allowed in these cleanrooms to test software on their clients' premises. Actions of these test engineers are tightly monitored; network connections, phone calls, cameras, and USB keys are forbidden. Cleanroom testing requires significant resources and physical proximity of test outsourcing companies to their clients.

A more commonly used approach is to use tools that anonymize databases indiscriminately, by gen-

eralizing or suppressing all data. Even though this procedure is computationally intensive, it is appealing since it does not require sophisticated reasoning about privacy goals and protects all data.

But in many real-world settings, protecting all data blindly makes testing very difficult. When large databases are repopulated with fake data, it is likely that many implicit dependencies and patterns among data elements are missing, thereby reducing testing efficacy. Moreover, fake data is likely to trigger exceptions in DCAs, leading test engineers to flood bug-tracking systems with false bug reports. In addition, testers often cannot use such anonymized data because DCAs may throw exceptions that would not occur when the DCAs are tested with original data or other real data in field.

A more sophisticated approach is *selective anonymization*, where a team is assembled comprising of business analysts and database and security experts [58, page 134]. After the team sets privacy goals, identifies sensitive data, and marks database attributes that may help attackers to reveal this sensitive data (i.e., QIs), anonymization techniques are applied to these QIs to protect sensitive data, resulting in a sanitized database. A goal of all anonymization approaches is to make it impossible to deduce certain facts about entities with high confidence from the anonymized data [4, pages 137-156]. Unfortunately, this approach is subjective, manual, and laborious. In addition, it involves highly trained professionals and therefore this approach is very expensive. Currently, there exists no solution that enables these professionals to accomplish this task efficiently and effectively with metrics that clearly explain the cost and benefits of selective anonymization decisions.

2.2.3 Balancing Utility And Privacy

Utility of anonymized data is measured in terms of usefulness of this data for computational tasks when compared with how useful the original data is for the same tasks. Consider an example of the utility of calculating average salaries of employees. Adding random noise to protect salary information will most likely result in computing incorrect values of average salaries, thereby destroying utility of this computation. Recent cases with U.S. Census show that applying data privacy leads to incorrect results [5, 12]. Multiple studies demonstrate that even modest privacy gains require almost complete destruction of the data-mining utility [1, 18, 34, 36, 68].

Data swapping is an anonymization technique that is based on exchanging values of attributes among individual records while maintaining certain distribution properties [84, 85]. Data swapping is more effective in preserving utility than data suppression and generalization privacy algorithms since data swapping allows users to better preserve statistical information [42, page 42]. In this chapter, we develop a data swapping privacy algorithm that allows analysts to balance privacy and utility goals, i.e., to choose appropriate levels of privacy that will guarantee certain basic test coverage.

2.2.4 The Problem Statement

The problem statement that we address in this chapter is how to enable stakeholders to balance a level of test coverage for DCAs (i.e., utility) with privacy for databases that these DCAs use. The problem space is restricted by three fundamental constraints of software development and privacy-preserving data publishing as follows.

First, a chosen anonymization approach should preserve original data as much as possible since it is important for preserving the testing utility of DCAs. Anonymizing databases using data suppression techniques may result in a different behavior of the DCAs. Suppose that $N_o \subseteq N$ is the set of all covered nodes in the control flow graph of a DCA when the DCA is tested with the original data, and $N_a \subseteq N$ is the set of all covered nodes when the same DCA is tested with the anonymized data. In general, testing with anonymized data makes DCAs behave in ways that are different from specifications, and as a result new execution paths in DCAs with anonymized data may lead to exceptions or not-covered branches being covered originally. For example, the DCA logic handles suppressed values of `Nationality` by not covering the body of the `if` statement shown in Figure 2.1. Therefore, in the worst case $N_T = N_o \cap N_a = \oslash$, where N_T is the set of nodes of the preserved statement coverage. A problem is how to keep a good extent of the original data in databases for testing DCAs while guaranteeing certain levels of privacy.

Our goal is to ensure that all statements (i.e., nodes in the control flow graph) that are executed with original data will also be executed with anonymized data. Our goal is difficult to achieve since it is an undecidable problem to determine precisely how values of QIs are used in DCAs [65]. In addition, anonymization algorithms present a dilemma: suppressing attribute data with different values results in loss of test coverage, and keeping original data in the database results in loss of privacy. Balancing these conflicting outcomes is the problem that we address in this chapter.

Second, as a result of software evolution [40, page 163], the code of the DCA is modified. In consecutive releases, the DCA may use different database attributes in different ways, thereby requiring the DCA owner to reanonymize data to ensure that the balance between privacy and utility is maintained. However, re-anonymization introduces a problem – if an attacker keeps the previous version of the anonymized data where values of some attributes are not sanitized, then this attacker can link original values in different releases, thus inferring sensitive information. A solution should enable stakeholders to repeatedly release anonymized data in such a way that both privacy and utility are guaranteed at certain levels.

Finally, a privacy metric should be linked directly to test coverage and vice versa, i.e., guaranteeing a certain level of test coverage should allow stakeholders to calculate bounds of the privacy level. For example, if some path is controlled by branch conditions that use values of database attributes, then it is possible to predict the effect of anonymization of these values on this path. In other words, the problem is to present business analysts with choices of predicted coverage levels for different anonymization

goals.

2.3 Our Solution

In this section, we present core ideas behind our approach that we call *PRIvacy Equalizer for Software Testing (PRIEST)* and we describe the PRIEST architecture and its workflow. In this section, we concentrate on the overall architecture of PRIEST that uses our privacy framework that we describe in Section 2.4.

2.3.1 Core Ideas

At the core of our work are three major ideas. The first one is a privacy framework that enables organizations to keep original values in sanitized data. The framework is based on a data swapping anonymization technique to preserve original values of database attributes. As a result, test coverage is not affected so negatively as it happens when data suppression and generalization techniques are used. The technique swaps data based on a probability value provided by a user. Hence different levels of privacy can be achieved.

The second idea is our guessing anonymity privacy metric that allows stakeholders to quantify the level of privacy achieved in an anonymized database. In particular, the metric provides measurement of difficulty for an attacker to relate a sanitized record to the original record.

The third idea is an idea to statically determine how different database values affect the behavior of a DCA. This idea unifies DCAs and their databases in a novel way – database attributes are tied to the source code of the DCAs, and depending on how the DCAs use values of these attributes, business analysts and security experts can determine what anonymization strategy should be used to protect data without sacrificing much of test coverage. A key point is that often not all of the database attributes have to be anonymized to achieve a given level of data protection. Therefore, it is important to extend data protection strategies with information about how DCAs use their databases to which these strategies are applied.

As it often happens, control-flow decisions in DCAs are affected by a smaller subset of database attributes. In an extreme case, if some data in the database is not used in any control-flow decision of any DCAs, then anonymizing this data will have no effect on these DCAs. A more subtle point is that values of some attribute may not affect most branch conditions in DCAs, and therefore test coverage will not be affected much if this attribute is anonymized. Thus it is beneficial to focus anonymization on those aspects of the data that have minimal influence on deeply nested control-flow decisions.

2.3.2 PRIEST Architecture and Process

We propose a novel process for using PRIEST that partially involves the cleanroom testing that we described in Section 2.2.2. Recall that business analysts and test managers from outsourcing companies come to the cleanroom of their client company to evaluate their client's DCAs and to plan work. As part of their evaluation, these analysts and managers determine different sets of attributes of the database (i.e., QIs) that can be used by attackers to re-identify entities. In general, only a few subsets of these QIs should be anonymized, thus creating favorable conditions for preserving test coverage. With PRIEST, this QI selection procedure can be improved by pointing out the QIs that affect test coverage the least. These analysts and managers can use PRIEST as part of their evaluation and planning in order to determine how to maximize test coverage for DCAs while achieving desired privacy goals for databases that these DCAs use.

As the first step of the PRIEST process, programmers link program variables to database attributes using annotations, so that these annotations can be traced statically using control- and data-flow analyses. Tracing these attribute annotations is required to determine how the values of these attributes are used in conditional expressions to make branching decisions, thereby influencing the execution flow of the DCAs. Our goal is to quantify the effect of replacing values of database attributes on reachability of program statements.

At first glance, it appears to be tedious and laborious work for programmers to annotate program variables with the names of database attributes from which these variables obtain values. In reality, it is a practical and modest exercise that takes little time. Programmers annotate selected program variables only once, where these variables are first assigned values that are retrieved from recordset objects using specific database-related API calls. We observe that in many projects it is a small fraction of code that deals with obtaining values from databases, and most code is written to implement application logic that processes these values. This observation is confirmed by a previous study [48] that shows that out of 2,080 randomly chosen Java programs in Sourceforge, there is approximately one JDBC-related API call per 2,200 LOC on average that retrieves a value from a recordset object and assigns it to a program variable. Extrapolating this result means that programmers may have to annotate approximately 450 variables for a project with 1 Million LOC and such expense is acceptable.

In addition, a variety of *object-relational mapper (O/R mapping)* tools and frameworks bridge the gap between an application's object model and the relational schema of the underlying database by generating classes that represent objects in relational databases [6]. Since O/R mapping is done automatically, links between program variables and database attributes are recovered as a by-product of using O/R mapping tools. For example, one of our subject applications, a logistics application for handling one of the largest supermarket chains in Spain, is written using iBatis[2], an open-source O/R mapper.

Figure 2.2 shows the architecture of PRIEST. The inputs to PRIEST are the DCA bytecode and the

[2]http://ibatis.apache.org

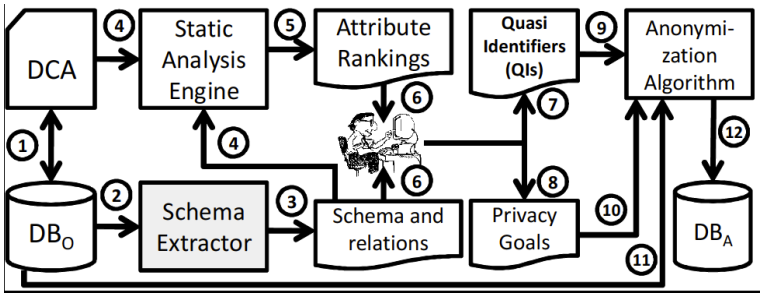


Figure 2.2: PRIEST architecture and workflow. Solid arrows depict the flow of command and data between components, numbers indicate the sequence of operations in the workflow.

original database DB_O that this DCA uses (1). With PRIEST, business analysts, test managers, and security experts connect to DB_O (2) to obtain its schema using the Schema Extractor that uses JDBC metadata services to obtain (3) database schema including all relations among different attributes. Next, PRIEST performs control and dataflow analyses (4) using the Soot toolkit[3] to establish how the DCA uses values of different database attributes. Values of some attributes are used in expressions to compute other values, which in turn are used in other expressions and statements. In some cases, these values are used in conditional statements, and they affect control flows of DCAs using control-flow dependencies. Ideally, attributes whose values affect many other expressions and statements in DCAs (in terms of statement coverage) should not be picked as QIs. The output (5) of this procedure is a list of attribute rankings that show how many statements are approximately encapsulated by branches whose conditions contain program variables that receive their values from database attributes.

At this point, PRIEST displays to the user (6) the list of ranked attributes and their relations with other attributes in the database schema. The user selects (7) a subset of database attributes as QIs whose values should be anonymized to protect sensitive data based on certain privacy goals (8) that are required by the DCA owner. The selected QIs are supplied (9) to the Anonymization Algorithm along with (10) the required privacy level. In addition, the algorithm takes (11) DB_O as its input and outputs (12) the anonymized database DB_A.

2.3.3 Ranking Attributes

To understand which attributes affect DCAs the most, we rank these attributes by counting the numbers of statements that their values affect. To find the preceding information, our approach uses taint analysis to track the annotated variables (as described in Section 2.3.2) corresponding to each attribute. In particular, for each attribute, our approach uses control- and data-flow taint propagation [27] to find out branch conditions that are tainted by an annotated variable corresponding to the attribute. We then count the number of statements that are control-dependent on these branch conditions. We perform virtual-call

[3]http://www.sable.mcgill.ca/soot

resolution using static class hierarchy analysis, and we take a conservative approach by counting all the statements in all the target methods that can potentially be invoked.

Our experiments on subject applications show that this approach can predict the effect of anonymization on coverage within less than 8.3% error (see Section 2.5 and Table 2.6 for further details). Improving the precision to compute how many statements are affected by these attributes is a subject of future work.

2.4 PRIEST Privacy Framework

In this section, we describe the PRIEST privacy framework. We discuss constraints, show how to connect different aspects of data privacy with the testing utility, and present a privacy metric and our anonymization algorithm.

2.4.1 Constraints and Goals

After discussions with a number of experts from different organizations who are involved in testing DCAs, we identified two main constraints for a successful anonymization solution: simplicity and consistency. A simple anonymization solution should not impose significant laborious manual or intellectual effort on stakeholders who are involved in protecting privacy. A case at hand is when using data suppression and generalization algorithms to achieve k-anonymity, a popular data privacy approach, users must specify generalization hierarchies that guide corresponding anonymization algorithms to replace data values with generalized substitutes to protect sensitive information. For example, a branch of the generalization hierarchy for the attribute `Nationality` could be $\{Canadian, Mexican, USA\} \rightarrow NorthAmerican \rightarrow American \rightarrow Human$. In general, identifying and properly using these hierarchies involves deep domain expertise and significant effort. Stakeholders want a high degree of automation where they are presented with choices for privacy levels and corresponding test coverages, making it easy for them to balance privacy and utility.

Consistency of data is the other constraint. Since data used in typical DCAs can span multiple tables in databases, any field that is used as a key to link entities across tables needs to be anonymized *consistently* so that the data can be linked correctly after anonymization. Thus, anonymization techniques should take into consideration different constraints among attributes that are imposed by the database schema for describing this data.

Preserving original data values is a goal of our anonymization framework – no new values for a data field should be introduced and the unique set of values in a field should be preserved after anonymization. This constraint is important because changing the values (such as generalizing a five digit zip code to first three digits, or replacing the city name with the state in the `City` field would result in new values for a field that the DCA may not be able to handle properly. Having new data values often results in

extra effort to modify the DCA and may cause unintended consequences in production with real data; for example, exceptions that are thrown would not be thrown if the original data is used.

Random data generation would typically preserve only univariate properties (marginal distributions) of each attribute. Our approach can preserve distributions over conjunctions of attributes (Gender=Female, Diagnosis=Ovarian Cancer) that are useful for preserving test coverage when conditional statements containing variables linked to multiple attributes are concerned, and such cases are common in real-world DCAs.

Finally, in this chapter, we assume that the developer or data owner cannot be an attacker. While other attack models are possible, we assume that since developers have direct access to all information that is needed to create software, it is reasonable to assume that developers enjoy a high level of trust.

2.4.2 Guessing Anonymity

Guessing anonymity [83] is a privacy metric that enables stakeholders to quantify the level of privacy using a guessing strategy of the attacker as a sequence of questions of the form "Are these the original values of quasi-identifiers that are used to generate a sanitized record?"

Definition: The guessing anonymity of the sanitized record is the number of guesses that the optimal guessing strategy of the attacker requires in order to correctly guess the record used to generate the sanitized record.

To illustrate this definition, consider an attacker with knowledge that Alice and Chris are in the database, and knowledge of their true ages shown in Table 2.1 that contains original data. Sanitized data is shown in Table 2.2 where names are replaced with "***" and the values of the attribute `Age` are perturbed with random noise. The guessing anonymity of each record is shown in the fourth column of Table 2.2. While the record that corresponds to Alice has a guessing anonymity of two, the sanitized record corresponding to Chris has a guessing anonymity of one due to the fact that his age is significantly higher than the ages in the other records. The distribution of guessing anonymity of the different records in a database can be used to define a variety of privacy metrics. *Mean guessing anonymity*, *Minimum guessing anonymity*, and *Fraction of records* with guessing anonymity greater than a certain value m are some metrics that we discuss later in this section. For our toy database shown in Table 2.1, those values would be 1.75, 1, and 0.75 (for m=1), respectively.

To further illustrate our definition of guessing anonymity, we link our definition to the definition of k-anonymity. Recall that in k-anonymity, a database is considered private if every record has at least k other records in the original database with which the released record is indistinguishable. k-anonymity can be achieved by different anonymization operators but typically suppression and generalization operators are the most commonly used ones.

Table 2.1: Original Database

Name	Age	Procedure or Prescription
Alice	19	Antidepressants
Bob	15	Antibiotics
Chris	52	Chemotherapy
Diana	25	Abortion

Table 2.2: Sanitized Database

Name	Age	Procedure or Prescription	Guessing Anonymity
***	23.1	Antidepressants	2
***	19.4	Antibiotics	2
***	49.3	Chemotherapy	1
***	21.1	Abortion	2

Algorithm 1 The `PriestPrivacy` algorithm.

1: **PriestPrivacy**($\{QI\}$, p) $\{\{QI\}$ is the set of QIs, and p is the probability that the original data will remain unchanged in the anonymized set, $T\}$
2: $\|T\| \leftarrow$NULL $\{$Initialize values of the anonymized matrix.$\}$
3: **for** $j \leftarrow 1$ to # of attributes in $\|QI\|$ **do**
4: **DistinctValues**$(\|QI\|_j) \mapsto \{V\}_j$ $\{$Returns the set of distinct values for a given attribute.$\}$
5: **for** $i \leftarrow 1$ to # of rows in $\|QI\|$ **do**
6: **Randomize**$(QI^i_j, p) \mapsto (\exists v \in \{V\}_j$ s.t. $T^i_j \leftarrow v)$
7: **end for**
8: **end for**
9: **return** $\|T\|$

Consider a released record that is anonymized such that there are exactly k records in the original database with which the released record is consistent. Without any further information, the optimal guessing strategy would choose among these k records with uniform probability for its first guess. The probability of the first guess being correct is $\frac{1}{k}$. If the first guess is incorrect, the second guess is chosen with uniform probability from among the remaining $k-1$ records. The probability of the second guess being correct is $\frac{1}{k-1}(1-\frac{1}{k})$. The expected number of guesses simplifies to $\frac{k+1}{2}$, so the expected guessing anonymity of a k-anonymized record is $\frac{k+1}{2}$.

2.4.3 The PRIEST Anonymization Algorithm

The algorithm `PriestPrivacy` is shown in Algorithm 1. We use a data swapping anonymization technique to preserve original values of database attributes. The goal is to preserve test coverage better than when data suppression and generalization techniques are used, while keeping the data usable by the

Table 2.3: A table with original and anonymized (superscript) data. For example, the original value of the attribute Age is 30 and the sanitized value is 40.

Record	Age	Gender	Race
Rec 1	30^{40}	F^{M}	W^{B}
Rec 2	40^{40}	M^{M}	B^{H}
Rec 3	45^{30}	M^{F}	H^{W}
Rec 4	30^{40}	F^{M}	W^{H}

Table 2.4: A similarity matrix for the table shown in Table 2.3. Each cell contains the value that shows the fraction of attributes values that are the same between original and sanitized records.

Original / Anonymized	Rec 1	Rec 2	Rec 3	Rec 4
Rec 1	0	1	0.33	0
Rec 2	0	0.66	0.66	0
Rec 3	1	0	0	1
Rec 4	0	0.66	0.66	0

DCA. To protect privacy, each value for each row for each attribute is swapped with some probability with a different value for the same attribute. For example, for the attribute `Gender` that contains two distinct values `M` and `F`, the user may choose to replace the value of a given cell with the other value with probability 0.5, i.e., an unbiased coin flip. Excluding an attribute from anonymization means that the probability of value replacement for this attribute is zero.

This algorithm takes as its inputs the matrix of QIs and their values, $\|QI\|$, whose columns include QIs and rows include different tuples for these QIs from the original database DB_O, and the value of the probability, p, that the original data will remain unchanged in the anonymized matrix, $\|T\|$. The matrix $\|T\|$ has the same dimensions and semantics as the matrix $\|QI\|$, and it contains anonymized values of QIs. $\|T\|$'s values are initialized to `NULL` in Line `2` and this matrix is returned in Line `9`.

The `for`-loop in Lines `3-8` iterates through attributes that are in $\|QI\|$, and the procedure `DistinctValues` is called to compute the set of distinct values for each QI, $\{V\}$. Next, in Lines `5-7` the nested `for`-loop iterates through all rows for the given QI and the procedure `Randomized` is invoked to replace the original value in a cell with one of the original distinct values of the given QI, and the replaced value is written in the corresponding location in the matrix $\|T\|$. Once the algorithm iterates over all QIs and all rows for each QI, it terminates and returns $\|T\|$ in Line `9`.

An example of application of the algorithm `PriestPrivacy` is shown in Table 2.3 that contains (for illustrative purposes) three attributes: `Age`, `Gender`, and `Race`. The first column of this table holds the record number. Original values are shown in each cell with their sanitized replacements shown as a superscript for each value. For example, record `1` contains original value `30` for the attribute `Age` that is

Algorithm 2 Guessing anonymity metric calculation algorithm.

PrivacyMetric($\|QI\|$, $\|T\|$)
$\|D\| \leftarrow 0$ {Initialize values of the distance matrix, D.}
for $i \leftarrow 1$ to # of rows in $\|T\|$ **do**
  for $k \leftarrow 1$ to # of rows in $\|QI\|$ **do**
    for $j \leftarrow 1$ to # of attributes in $\|QI\|$ **do**
      if $T_j^j = QI_j^i$ **then**
        $D_k^i \leftarrow D_k^i + 1$
      end if
    end for
    $D_k^i \leftarrow \frac{D_k^i}{\#ofattributesinQI}$
  end for
end for
{Compute privacy metrics PM_1 and PM_2.}
$R \leftarrow 0$
$diffRecords \leftarrow 0$
$d \times d \leftarrow \|D\|$ {Dimensions of the similarity matrix.}
for $i \leftarrow 1$ to d **do**
  if $D(i,i) < 1$ **then**
    $diffRecords \leftarrow diffRecords + 1$ {For metric PM_2.}
  end if
  for $j \leftarrow 1$ to d **do**
    if $i \neq j \wedge D(i,j) \geq D(i,i)$ **then**
      $R \leftarrow R + 1$
    end if
  end for
end for
return $PM_1 = \frac{R}{d}$, $PM_2 = \frac{diffRecords}{d}$

anonymized and replaced with the value `40`, which is one of three distinct values for this attribute. The superscripted values populate the matrix $\|T\|$. Interestingly, the sanitized record `1` matches the original record `2`; however, since the attacker sees only the sanitized data, it is not possible for the attacker to know with certainty that the sanitized record matches some original record.

2.4.4 Privacy Metrics

We first calculate the probability distribution of guessing anonymity over all the database records. Specific privacy metrics are then defined as a function of that distribution. In this chapter, we define three of them but the optimal metric may vary based on the task at hand.

A privacy metric measures how identifiable records in the sanitized table are with respect to the original table [42, page 43]. The privacy metrics that we propose in this chapter are all motivated by

the notion of guessing anonymity. We calculate guessing anonymity using the algorithm shown in Algorithm 2. We begin by creating a similarity matrix that shows the similarity between sanitized and original records. For each record R_o in the original table, the similarity of R_o to a record R_s in the sanitized table is measured by the fraction of attributes whose values are the same between R_o and R_s. This computation is performed in Lines `3-12` of the privacy metric algorithm. This similarity matrix, $\|D\|$, shows similarities between rows in the original matrix, $\|QI\|$, and the anonymized matrix, $\|T\|$.

Table 2.4 shows an illustrative example for computing the similarity matrix $\|D\|$ for the data in Table 2.3. The similarity matrix $\|D\|$ has dimensions $d \times d$, where d is the number of records in the matrices $\|QI\|$ and $\|T\|$. Rows correspond to anonymized records and columns correspond to the original records in $\|D\|$. The values of the attributes `Age`, `Gender`, and `Race` are `30`, `F`, and `W` for the original record `1`, respectively, and the values for the sanitized record `1` are `40`, `M`, and `H`, respectively. Therefore, the similarity score is zero for the original and sanitized record `1`. In contrast, the similarity score is one for original record `2` and sanitized record `1` since all attribute values of the original record 2 match the values of their corresponding attributes for the sanitized record `1`.

We use the similarity matrix $\|D\|$ to compute how difficult it is for attackers to guess original records given sanitized records. Consider an extreme case where $\|D\|$ is a diagonal matrix, i.e., all entries outside the main diagonal are zero. In this case, each record is similar to itself only, that is, all records are unique and easily identifiable by attackers. The privacy level for this sanitized table is zero. The other extreme case is when all sanitized records are similar to all other records. In this case, it is very difficult for attackers to guess original data since all sanitized records are highly similar to one another. Correspondingly, the privacy level for this sanitized table is close to one. In practice, the privacy level is between zero and one, and our goal is to help analysts find the right balance between a privacy level and the utility of the sanitized database.

The distribution of guessing anonymity of the different records in a database can be used to define a variety of privacy metrics. In this chapter, we propose three different privacy metrics that are derived from the guessing anonymity distribution, but do not focus on providing the *best* metric. We believe that different applications and risk tolerances of users would require the use of different derived metrics and leave the optimal metric determination as future work.

In this chapter, we propose three metrics based on guessing anonymity: PM_1: *Mean guessing anonymity*, PM_2: *Fraction of records with guessing anonymity greater than a certain value m (where we set m to one in this chapter)*, and *Unique Records*. The computation of these privacy metrics PM_1 and PM_2 is shown in Lines `14-27` of Algorithm 2, and their values are returned in Line `27`.

PM_1: The *Mean guessing anonymity* of a database is the arithmetic mean of the individual guessing anonymities for each record in the database. This metric gives us measurement of the overall privacy of the sanitized database and helps us compare different versions of sanitized databases. Naturally, a database with higher *Mean guessing anonymity* would be more difficult to attack and hence have higher privacy. Suppose that there is a record $r_o \in T_o$, where T_o is the table with original records. After T_o is

sanitized, the table T_s with sanitized records is obtained, $r_s \in T_s$, and we compute the similarity matrix, $\|D\|$, for records in these tables. Then, for each $r_o \in T_o$, we increase the counter, R, by one if we find a record in the table T_s that has the similarity score in $\|D\|$ equal to or higher than the similarity score between r_o and its sanitized version, r_s, not counting the similarity score between r_o and r_s. The formula for the privacy metric is $PM_1 = \frac{R}{d}$, where d is the number of records.

PM_2: The fraction of records with guessing anonymity greater than m is our second metric. We set m to one for the work in this chapter; such setting corresponds to measuring the fraction of original records that have been modified at all by the anonymization algorithm. PM_2 is calculated as the ratio of records that were sanitized and that differ from their original record in at least one attribute value to the total number of records. This metric is useful for a variety of reasons. If the fraction of records that have guessing anonymity greater than m is too low and the users are unsatisfied, they have the option to increase the level of privacy or remove the records that fall below the threshold from the sanitized database, thus making the database more private.

UniqueRecords is measured as part of PM_2 and it is defined as follows. Suppose that there is a record $r_o \in T_o$, where T_o is the table with original records. After T_o is sanitized, the table T_s with sanitized records is obtained. Let $r_s \in T_s$ be the sanitized record for $r_o \in T_o$. r_s is a unique record if there exists $r'_o \in T_o$, such that all attributes of r'_o have the same value as the attributes of r_s. A key idea of guessing anonymity is that although some records after applying data swapping may match some original records, the attacker still cannot know with certainty which ones do and whether sensitive information (that these records identify) was not changed. Ideally, the percentage of unique records in the anonymized database will be zero - that is what we would like to achieve. If the number of unique records is greater than zero and the desired goal is to have complete anonymity, the users will have to delete these unique records before releasing the database.

2.5 Experimental Evaluation

In this section, we describe the results of the experimental evaluation of PRIEST on three open-source Java programs and one large commercial application that is used to manage logistics of one of the largest supermarket chains in Spain.

2.5.1 Research Questions

In this chapter, we make one meta-claim – our privacy framework for managing the tradeoff between data privacy and test coverage for DCAs is "better" than other frameworks. We define "better" in two ways: *coverage* and *flexibility*. Achieving higher coverage with PRIEST means that for a given level of privacy, we can provide higher test coverage. We measure it using the area under the privacy-coverage curve. Flexibility means that we can give more choices to the business analyst to make informed deci-

sions. If more data points in that curve can be created using PRIEST for a given range of privacy, we have a better framework for managing the tradeoff.

PRIEST does not compete with other anonymization techniques, such as *k*-anonymity since the latter is a metric, not a framework. Our claim is that our framework (that includes guessing anonymity) is a better framework to achieve coverage and flexibility. We seek to answer the following research questions.

RQ1 How much test coverage does PRIEST help achieve at given levels of privacy for subject applications?

RQ2 How effective is PRIEST in achieving different levels of data privacy while preserving original data?

RQ3 How effective is PRIEST in releasing different versions of anonymized databases for the same level of privacy and test coverage without enabling the attacker to link sensitive data?

With RQ1, we address our claim that we designed and implemented a technique using program analysis for determining how values of database attributes affect test coverage of DCAs that use this data. Our goal is to show that with PRIEST, the privacy metric is linked directly to test coverage and vice versa; in other words, guaranteeing a certain level of test coverage should allow stakeholders to calculate bounds of the privacy level.

With RQ2, we address our claim that using different levels of privacy enables business analysts to make trade-off decisions about privacy and utility.

With RQ3, we address our claim that PRIEST enables stakeholders to repeatedly release anonymized data in such a way that both privacy and utility are guaranteed at certain levels. Suppose that a technique of data suppression anonymization is applied to protect data and release the anonymized data with the DCA to testers. After some time, programmers produce the next release of this DCA that uses different attributes differently in its database. Suppose that the DCA does not use the attribute `Nationality` any more, and it uses the attribute `Race` instead. At this point, the database should be reanonymized, since the original values of the attribute `Race` should be left intact to preserve test coverage. However, in the previous anonymization, the values of the attribute `Nationality` are left unprotected. It means that the testers (the attackers) know original values of the attribute `Nationality` and now the testers will know the original values of the attribute `Race`. Together, these attributes enable the attackers to infer sensitive information from the released database.

2.5.2 Subject Programs

We evaluate PRIEST with three open-source and one large commercial Java programs that belong to different domains. Our selection of subject programs is influenced by several factors: sizes of the databases,

Table 2.5: Characteristics of the subject DCAs. App = application code, Test = test cases, DB = database, Tbl = tables, Att = attributes in all tables, IC = initial test coverage with the original database, ETC = estimated worst test coverage with sanitized data using the approach described in Section 2.3.

DCA	App [kLOC]	Test [kLOC]	DB [MB]	Tbl	Att	IC %	ETC %
N2A	77.8	19.3	20	506	2514	53	50
DurboDax	2.8	2.0	49	27	114	77	59
UnixUsage	2.8	.9	21	8	31	61	53
RiskIt	4.3	2.6	628	13	57	62	48

size of the source code, presence of unit, system, and integration tests, and the presence of embedded SQL queries that these programs use.

We selected four subject programs that come with test cases. `N2A` is a program for handling logistics of one of the largest supermarket chains in Spain. `N2A` has a total of 77,828 LOC. `RiskIt` is an insurance quote program[4]. `DurboDax` enables customer support centers to manage customer data[5]. Finally, `UnixUsage` is a program for obtaining statistics on how users interact with Unix systems using their commands[6].

Table 2.5 contains characteristics of the subject programs, their databases, and test cases. The first column shows the names of the subject programs, followed by the number of lines of code, LOC for the program code and accompanying test cases. The source code of the project ranges from 2.8 to 77.8 kLOC. The test cases range from 0.9 to 19.3 kLOC. Other columns show the size of the database, number of tables and attributes in the database, test coverage (statement coverage) that is obtained with the original database, and the estimated worst test coverage (statement coverage) with sanitized data using our approach described in Section 2.3.

2.5.3 Methodology

To evaluate PRIEST, we carry out experiments to explore its effectiveness in enabling users to determine how to balance test coverage while achieving different levels of data privacy (RQ1 and RQ2), and to show that it is possible to apply PRIEST to get privacy guarantees while releasing different sanitized versions of the database for the same privacy levels thereby supporting software evolution (RQ3).

[4]https://riskitinsurance.svn.sourceforge.net as of March 10, 2011.
[5]http://se547-durbodax.svn.sourceforge.net as of March 10, 2011.
[6]http://sourceforge.net/projects/se549unixusage as of March 10, 2011.

Variables

The main independent variable is the value of p. Two main response variables are the time that it takes to anonymize data in the database to achieve the desired level of k to answer RQ2 and the test coverage in percentage of program statements to answer RQ1.

The Structure of the Experiments

For the experiments, we select as QIs all attributes whose values affect execution paths in the corresponding DCAs. It is physically not possible to carry out an experiment using all subsets of the powerset of attributes as QIs where we measure test coverage for subject DCAs while achieving anonymity, since it would require us to consider the powerset of all attributes. Given that databases of the subject DCAs contain between 31 and 2,514 attributes, it is challenging to select a subset of them as QIs to enable achieve the higher possible level of test coverage.

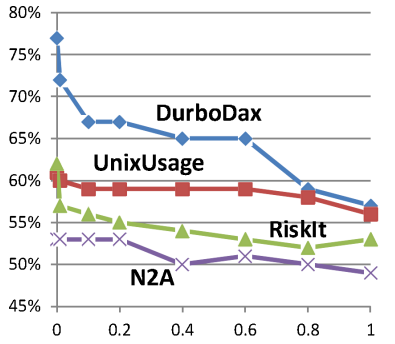
Our goal is to run experiments for different values of the independent variable p and report the effect of varying the values on p on dependent variables. To address RQ3, we anonymize databases for the subject DCAs for a given level of p, and we report and analyze privacy metrics between different sanitized versions of the same original database.

2.5.4 Threats to Validity

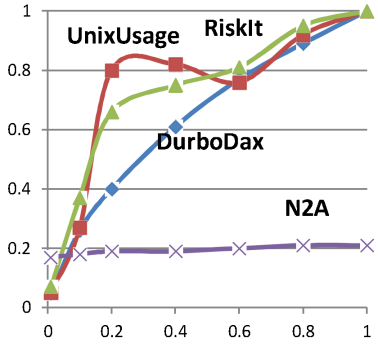
A threat to the validity of this experimental evaluation is that our subject programs are of small to moderate size because it is difficult to find a large number of open-source programs that use nontrivial databases. Large DCAs that have millions of lines of code and use databases whose sizes are measured in thousands of tables and attributes may have different characteristics compared to our small to medium size subject programs. Increasing the size of applications to millions of lines of code may lead to a nonlinear increase in the analysis time and space demand for PRIEST. Future work could focus on making PRIEST more scalable.

2.5.5 Results

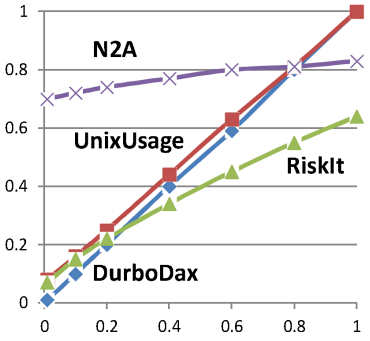
The results of the experiments conducted to address $RQ1$ and $RQ2$ are shown in Figure 2.3 and Table 2.6. Dependency of test coverage on the probability p of replacing original values is shown in Figure 2.3a. The maximum reduction in test coverage is close to 26% from the initial test coverage with $p = 1$ for application DurboDax. Previous work [47] showed that the maximum reduction in test coverage for the same applications reaches 80% from the initial test coverage when using a popular algorithm Datafly that is based on data suppression and generalization techniques [103]. We observed the biggest drop in test coverage with Datafly when $k = 7$, while much smaller maximum drop is observed with PRIEST for $p = 1$, which is the maximum value for the anonymization parameter.



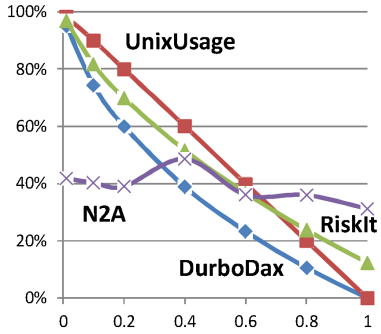
(a) Test coverage.



(b) Privacy metric PM_1.



(c) Privacy metric PM_2.



(d) Unique records.

Figure 2.3: Experimental results for the subject DCAs. All graphs show the dependence on the probability of replacing original data value that is assigned to the horizontal axis. The vertical axis designates dependent variables that are given in the captions to these figures.

Table 2.6: Results of experiments with subject DCAs for different values of the independent variable p that is shown in the last column header that spans seven subcolumns. The second column, QI, shows the number of QI whose values affect control-flow decisions in DCAs, the next two columns show the initial test coverage (statement coverage) with the original database and the estimated worst test coverage with sanitized data using the approach described in Section 2.3. The next column shows the error in the estimated test coverage that varies from 2.0% to 8.3%. Finally, the next column lists four dependent variables, privacy metrics PM_1 and PM_2, test coverage, TC, and the percentage of unique records, UR. Finally, the last seven subcolumns show values of these dependent variables for different values of p.

Subject	QIs	Total Records	Initial Cov	Worst Exp Cov	Max Err Cov	Dependent Variable	The Probability of Value Replacement, p						
							0.01	0.1	0.2	0.4	0.6	0.8	1
N2A	61	586801	53%	50%	2.0%	PM_1	0.17	0.18	0.19	0.19	0.20	0.21	0.21
						PM_2	0.70	0.72	0.74	0.77	0.80	0.81	0.83
						TC,%	49.00	49.00	49.00	50.00	51.00	49.00	49.00
						UR,%	41.80	40.30	39.00	48.60	36.10	36.00	31.30
DurboDax	20	88450	77%	59%	3.6%	PM_1	0.05	0.26	0.40	0.61	0.77	0.89	1.00
						PM_2	0.01	0.10	0.20	0.40	0.59	0.80	1.00
						TC,%	72.00	67.00	67.00	65.00	65.00	59.00	57.00
						UR,%	95.20	74.40	60.00	38.90	23.30	10.60	0.00
UnixUsage	16	250570	61%	53%	5.7%	PM_1	0.05	0.27	0.80	0.82	0.76	0.92	1.00
						PM_2	0.08	0.16	0.25	0.44	0.63	0.81	1.00
						TC,%	60.00	59.00	59.00	59.00	59.00	58.00	56.00
						UR,%	98.30	90.00	80.00	60.20	39.90	20.00	0.01
RiskIt	28	1270792	62%	48%	8.3%	PM_1	0.07	0.37	0.66	0.75	0.81	0.95	1.00
						PM_2	0.07	0.15	0.22	0.34	0.45	0.55	0.64
						TC,%	57.00	56.00	55.00	54.00	53.00	52.00	53.00
						UR,%	96.90	81.70	69.90	51.60	36.70	23.80	12.30

The least drop in test coverage is observed for the application N2A, while the biggest drop is attributed to the application DurboDax. Our explanation is that N2A is least sensitive to values of the database attributes since it uses these values to compute results rather than in path conditions. We explain the sharper drop for DurboDax as a result of branch conditions that use conjunctions of multiple QIs, thereby making the application sensitive to joint distributions of values of different attributes that may be destroyed by anonymization. We also observe that the shape of the test coverage curves show gradual decline in test coverage rather than abrupt changes. The former makes it easier for stakeholders to balance privacy and coverage on a continuous scale.

Dependencies of privacy metrics PM_1 and PM_2 on the probability p of replacing original values are shown in Figure 2.3b and Figure 2.3c. Combined with the test coverage curve shown in Figure 2.3a, these dependencies enable business analysts to select a level of protection that also ensures a certain level of test coverage. While values of PM_1 are monotonically increasing for N2A, RiskIt, and DurboDax with the increase in p, the values of PM_1 for UnixUsage slightly drop when the values of p are increasing from 0.2 to 0.6. This result can be explained as an effect of the variations of random value replacement that may result in small changes in the numbers of similar records. The values for $PM1$ are relatively small for N2A since many of the selected QIs were involved in some primary or unique key. As a result, the sanitized records contain distinct values for these QIs and fewer records (in the original database) are likely to be similar to a sanitized record.

Opposite to the dependencies of privacy metrics PM_1 and PM_2, the graph of the dependency of

Table 2.7: Results for privacy metrics to compare two sanitized databases for $p = 0.6$ for subject DCAs

Subject DCA	PM_1	PM_2	Total Records	Unique Records	$UR, \%$
N2A	0.20	0.78	586801	265400	38.7
DurboDax	0.60	0.77	88450	20687	23.4
UnixUsage	0.61	0.60	250570	100422	40.1
RiskIt	0.47	0.76	1270792	512051	40.3

the percentage of unique records, UR shows corresponding decline in Figure 2.3d. It means that the percentage of unique records between the original and sanitized databases is monotonically declining as the values of the probability p are increasing thereby making it more difficult for attackers to use these unique records to guess the original data. While values of UR are monotonically decreasing for UnixUsage, RiskIt, and DurboDax with the increase in p, the values of UR for N2A slightly increase when the values of p are increasing from 0.2 to 0.6. This result can be explained as an effect of the variations of random value replacement that may result in small changes in the numbers of unique records.

To address RQ3, we generated two sanitized databases for $p = 0.6$ and computed the privacy metrics $PM1$, $PM2$, and number of unique records between the two databases. These metrics quantify the difficulty of an attacker to relate records from the two sanitized databases. Since the same database is anonymized independently and the probabilities of replacing cell values are independent from one another, then resulting databases cannot be deterministically correlated since they are not used to produce each other. Thus, the attacker has to guess original data independently, meaning that the lowest guessing anonymity score can be used to quantify the difficulty of an attacker to guess records from both sanitized databases.

The values for $PM1$ are slightly lower to the values of $PM1$ for $p = 0.6$ in Table 2.6, while the values of $PM2$ are slightly higher to the values of $PM1$ for $p = 0.6$ in Table 2.6. This phenomenon is expected since the variation between the records of the two anonymized databases is expected to be more as compared to variation between the records of original and an anonymized database. The number of unique records is similar to the number of unique records for $p = 0.6$ in Table 2.6. It is also important to note that the overall privacy of multiple sanitized databases is only as good as the privacy of the least private version. If the first released version has PM_2 = 3 and the second version has PM_2 = 5, PM_2 for the two databases combined will be 3. In general, for two released databases D_i and D_j, the overall PM_2 = Min($PM_2(D_i)$, $PM_2(D_j)$).

Result summary. These results strongly suggest that PRIEST helps achieve higher test coverage for given levels of privacy for subject applications when compared with Datafly that is based on data suppression and generalization techniques, thereby addressing RQ1. PRIEST can also achieve different

levels of data privacy while preserving original data as it is seen from Table 2.6, thereby addressing RQ2. Finally, the results shown in Table 2.7 strongly suggest that PRIEST is effective in releasing different versions of anonymized databases for the same level of privacy and test coverage, thereby addressing RQ3.

2.6 Related Work

Our work is related to regression testing [127] since PRIEST is used to assess the impact of data anonymization on testing. Numerous techniques have been proposed to automate regression testing. These techniques usually rely on information obtained from the modifications made to the source code. These techniques are not directly applicable to preserving test coverage while achieving data anonymity for test outsourcing, since regression information is derived from changes made to the source code and not to how this code uses databases.

Closely related to PRIEST is $kb-$anonymity model that enables stakeholders to release private data for testing and debugging by combining the $k-$anonymity with the concept of program behavior preservation [19]. Unlike PRIEST, $kb-$anonymity replaces some information in the original data to ensure privacy preservation so that the replaced data can be released to third-party developers. PRIEST and $kb-$anonymity are complementary in using different privacy mechanisms to preserve original data thereby improving its testing utility.

Recently proposed is an anonymization technique for protecting private information in bug reports that are delivered to vendors when programs crash on computers of customers [23] and the follow-up work on this technique by Clause and Orso [28]. This technique provides software vendors with new input values that satisfy the conditions required to make the software follow the same execution path until it fails, but are otherwise unrelated with the original inputs. This technique uses symbolic execution to create new inputs that allow vendors to reproduce the bug while revealing less private information than existing techniques. The technique requires test cases, which are not present in our situation. In contrast, PRIEST does not require any test case.

There has been a lot of recent work to achieve general purpose (task-independent) data anonymization. We choose the guessing anonymity approach because guessing anonymity can be used to provide privacy guarantees for data swapping algorithms and can also provide an optimal noise parameter when implementing data swapping algorithms for anonymization. In contrast, approaches that aim to achieve k-anonymity do not allow the user to explicitly control how much each record is altered. Empirical results reported by Rachlin et al. [83] show that Guessing anonymity outperforms DataFly, a well-known k-Anonymity algorithm on specific data mining tasks, namely classification and regression, while at the same time providing a higher degree of control over how much the data is distorted.

Recent work on privacy introduced a similar definition of privacy for noise perturbation methods, known as k-randomization [2]. This work defines a record as k-randomized if the number of records that

are a more likely match to the original is *at least k*. Although this notion is similar to the definition of guessing anonymity, the definition differs by not providing a lower limit on the number of records that provide a more likely match, and by explicitly establishing a connection between privacy and guessing functions.

2.7 Conclusion

We offer a novel and effective approach called PRIEST that helps organizations to remove an obstacle to effective DCA test outsourcing. With PRIEST, DCAs can be released to external testing organizations without disclosing sensitive information while retaining testing efficacy. We built a tool and applied it to nontrivial DCAs. The results show that PRIEST is effective in enabling users to determine how to balance test coverage while achieving different levels of data privacy, and that with PRIEST, users can release different sanitized versions of the database for the same privacy levels, thereby supporting software evolution.

Chapter 3

Test Generation for Database-Centric Applications via Mock Objects

Database-centric applications (DCAs) consist of a front-end application (FA) that interacts with a back-end database (DB). DCAs are increasingly used in mission-critical systems, such as aerospace crafts and medical devices, for the purpose of collecting, managing, and distributing mission-relevant information. For example, an infusion pump at a health care unit may have to retrieve drug libraries or prescriptions from the unit's central database to ensure right treatments being delivered to right patients. Since the failure of mission-critical systems may cause disastrous consequences, it therefore becomes critical to ensure the correctness and soundness of DCAs that mission-critical systems integrate to interact with a database. In this chapter, we focus on the problem of providing an effective means for evaluating the quality of DCAs.

Software (unit) testing has been widely used by developers to evaluate the quality of Software applications including DCAs. Given that manual test generation can be labor-intensive and error-prone, developers (or testers) often rely on automatic test generation tools [21, 45, 76, 95, 112] to obtain tests with high code coverage. However, these approaches, when applied to DCAs, could face two significant challenges:

First, automatic test generation usually requires to execute the DCA under test to obtain necessary information. As a result, the test generation process can be costly if the interaction between the front-end application of the DCA and the database is intensive and slow [88]. Moreover, the database associated with the DCA needs to be modified during test generation; such modification is often not desirable to testers because of the concerns of data privacy and preservation. Using a local test database, instead of the original one, may address this issue. However, it is not always an easy task to set up such a local database and clone configurations of the original database onto it.

Second, no matter what database (test or real) is used, a test generation tool needs to bridge the gap between the front end application (FA) of the DCA under test and its DB. More specifically, certain

portions of the FA can be covered only if specific records exist in the DB. The records in the existing DB might not be sufficient to cover those portions of the FA that are dependent on the DB. Hence, to cover such portions, generated tests need to not only provide inputs for the FA, but also prepare a particular set of initial states in the DB.

If our aim is to test only the FA (and not the database), mocking techniques [70] can be used to alleviate the first challenge. With such techniques, a real DB can be simulated by some trivially-implemented mock objects, called mock database, which is capable of providing default return values to database queries, without reflecting the genuine behavior of the real database. However, the default values cannot guarantee high coverage of the FA, leaving the second challenge not addressed. To cover most parts of the application, the mock DB ought to be sophisticated enough to mimic the behavior of the real DB and provide expected return values.

Parameterized mock objects [111] extend conventional mocking techniques to allow mock objects to automatically produce return values (to the executed queries), with the aim of covering as many program paths as possible. However, when applied to test DCAs, such techniques may not be effective, as they cannot assure that all return values reflect valid DB states. In other words, they simply assume that executing a SQL query can result in whatever values needed to cover some program path, even if such a query is not satisfiable. Moreover, parameterized mock objects cannot monitor the state of the DB as it evolves during system execution, resulting in infeasible database states. These infeasible states can result in generation of tests that are not valid in real situations. Hence, these tests can result in false warnings, which can be time-consuming for developers (or testers) to filter out.

In this chapter, we address the preceding challenges by providing an automatic test generation solution for DCAs. As inputs, our approach takes the FA of a DCA and schemas of its DB, and customizes a *mock database* that complies with the input schemas and *mirrors* real states of the DB. Our approach then simulates operations over the DB by performing corresponding ones over the mock DB. In this way, the mock DB tracks changes of the DB state during program executions. Hence, the generated tests do not result in false warnings.

Note that our approach does not require the associated DB as input, making the approach suitable for regulation (e.g., by the US Food and Drug Administration) or third-party reviewing (or testing) practices.

Our approach is implemented upon Pex [93], an automated test generation tool from Microsoft Research. To address the issue that Pex (or any other general white-box testing tool) cannot work closely with databases, our approach extends Pex with two important capabilities: first, a pre-processing component is attached to Pex, such that the input DCA is transformed and all its interactions with the database are replaced by corresponding interactions with the mock database; second, our approach equips Pex with the capability of injecting alternate records into the mock DB, allowing it to prepare initial database states needed to cover specific program paths. Due to code pre-processing, tests generated by our approach can be run only over the transformed code. However, such tests are accompanied with SQL

statements for preparing initial database states when testing the DCA with the real DB.

In summary, this chapter makes the following main contributions:

- We propose a technique to fully capture data-manipulation behavior of databases. Specifically, our technique devises a set of parameterized mock objects, which, when instantiated with schemas of any database, can faithfully mimic its data-manipulation behavior.
- We propose an automatic test generation technique for DCAs, no matter whether or not the DB is in place. By instrumenting a DCA with instantiated mock database objects, our technique allows to generate tests that reflect feasible DB states needed to cover various parts of the FA of th DCA under test.
- We have implemented a prototype tool, called MODA (Mock Object Based Test Generation for Database-centric Applications), for our approach. In its current implementation, MODA incorporates Pex [93], a tool from Microsoft Research, as its DSE engine.
- We have conducted two empirical evaluations to assess the effectiveness of MODA: one on a real-world medical device DCA and the other on an open source DCA. Results of the evaluations demonstrate that our approach can achieve higher code coverage, with no false warnings, than conventional DSE-based techniques.

3.1 Example

In this section, we illustrate major components of the MODA approach with an example. Figure 3.1 shows an FA of a DCA. The FA first sets up a connection to the DB at Lines 2-3, and then issues a query requesting the DB to delete from the table *persons*, any record with firstname *John*, lastname *Smith*, and with *age* more than 25. The FA then constructs and executes a select SQL query at Lines 6-7. In particular, the query selects all records from the table `persons` with age more than 25. The `while` loop at Line 8 is iterated for each record in the result set of the select query. A record should be selected with firstname *John* and with lastname *Smith* to allow the execution of the true branch of `if` statement at Line 10,. To execute the true branch of `else if` statement at Line 11, a record should be selected with firstname *Tom*. To test this method, a test generation technique needs to generate desirable DB states for the `person` table (used in the queries at Lines 4 and 6) in the DB. The generated DB states are crucial in testing this DCA: (1) the DB states determine which branches in Lines 10 and 11 are executed ; (2) the DB states determine how many times the loop body in Lines 8-12 is executed.

3.1.1 Existing Mocking Techniques With Example

Existing systematic test generation techniques for DCAs [39] require the DB to be in place. Mocking techniques can be used to replace the associated DB with a mock DB. A developer can replace the

```
1  public void ExecuteQuery(){
2    SqlConnection myConnection = new SqlConnection(...);
3    myConnection.Open();
4    SqlCommand myCommand = new SqlCommand(
         "DELETE FROM persons WHERE firstname = 'John'
          AND lastname = 'Smith' AND age>25", myConnection);
5    myCommand.ExecuteScalar();
6    myCommand = new SqlCommand(
         "SELECT firstname, lastname FROM persons
         WHERE age > 25", myConnection);
7    SqlDataReader myReader = myCommand.ExecuteReader();
8    while (myReader.Read())
9    {
10     if(myReader[0].Equals("John") &&
       myReader[1].Equals("Smith")){...}
11     else if(myReader[0].Equals("Tom")){...}
12   }
13 }
```

Figure 3.1: An example FA of a DCA.

method `ExecuteReader` with a mock method that captures his assumptions on how the result set of the query executed at Line 7 is used in the DCA. For the FA in Figure 3.1, to execute the true branch of `if` statement at Line 10, the developer has to manually configure the mocked `ExecuteReader` method, forcing it to return a record with firstname *John*, lastname *Smith*, and with age more than 25. To execute the true branch of `else if` statement at Line 11, a record should be selected with firstname *Tom* and with age more than 25. However, for FAs, it can be labor-intensive to achieve high code coverage using manually written mocked methods.

To automate the generation of records, the developer can use parameterized mock objects [111]. Suppose that parameterized mock objects are used to mock method `ExecuteReader` at Line 7 of Figure 3.1. To cover the true branch at Line 10, the mock method returns a *myReader* object with $myReader[0] =$ "*John*" and $myReader[1] =$ "*Smith*". However, since the query at Line 4 deletes all such records (records with firstname *John*, lastname *Smith*, and with age more than 25) from the DB, the true branch at Line 10 cannot be covered in reality. Such kinds of tests are generated because the mock method does not keep track of the records available in the mock DB and does not manipulate the records based on the SQL queries encountered in the FA. These tests can result in false warnings for the developer to filter out. Moreover, such an approach cannot be used to find bugs in the FA due to syntactically incorrect queries. Furthermore, these mocking techniques do not generate any records to be inserted in the DB (or the mock DB). Hence, tests generated by these techniques cannot be replayed on the DCA interacting with a real DB.

```
1 public class MockSqlConnection(string query){
2   public MockDBMS dbms;
3   public  MockSqlConnection(){};
4   public void  open(){
5     dbms = MockDBMS.MockDBMSFactory();
6   }
7 }
```

```
1 public class MockSqlCommand(string query){
2   public string sqlCommand;
3   public  MockSqlCommand(string command){
4    sqlCommand = command;
5   }
6   public MockSqlDataReader ExecuteReader(){
7    //Query Parsing and Simulation on the Mock DB
8   }
9   public int ExecuteScalar(){
10   //Query Parsing and Simulation on the Mock DB
11  }
12}
```

Figure 3.2: The Mock Methods for the DB Interface methods invoked in Figure 3.1.

3.1.2 MODA With Example

Our MODA approach takes as input the FA of a DCA and the schema of the DB. As output, our approach generates a test suite for the application that achieves high code coverage. MODA mocks the real DB and transforms the FA so that the FA interacts with the mock DB instead of the real DB. The mock DB used by our approach is capable of reflecting the effects of SQL queries by manipulating its states accordingly. With such a mock DB, our approach constructs the required mock-DB states to cover the true branches at Lines 10 and 11 (if possible). We next illustrate the components in our approach with the example in Figure 3.1.

Mock Object Framework

The *Mock Object Framework* component implements the mock objects for mimicking the associated DB and the operations that can be performed on the DB. In particular, the component provides the following functionalities:

Mocking DB Interface. The component mocks the APIs that provide an interface to the associated DB. The *Code Transformer* component (described in Section 3.1.2) replaces each such API method invocation with a corresponding mock method. Figure 3.2 shows the corresponding mock methods of the DB interface methods invoked in the FA in Figure 3.1. The `open` method of the mocked class `SqlConnection` inserts records in the mock DB to synthesize an initial mock DB state. These records are generated by the *Test Generation* component (described in Section 3.1.2). The methods `ExecuteReader` and `ExecuteScalar` of the mocked class `MockSqlCommand` parse a query, simulate the query on the mock DB, and return the result set for the query.

Mocking DB. This component takes as input the schema the associated DB and constructs a mock DB

that inherits all tables present in the original DB.

Mocking DB Operations. The component implements operations for selecting, inserting, updating, and deleting records in the mock DB. These methods correspond to $SELECT$, $INSERT$, $UPDATE$, and $DELETE$ SQL queries, respectively. Whenever the component encounters a SQL query (e.g., invocations of `ExecuteScalar` and `ExecuteReader` methods at Lines 5 and 7 of Figure 3.1), the component interprets and simulates the query on the mock DB. For the query at Line 5 of Figure 3.1, the component deletes all the records (present in the mock DB) with $firstname$ *John* , *lastname Smith*, and with *age* more than 25. For the query at Line 7 of Figure 3.1, the component selects all the records (from the mock DB) that have age more than 25.

Code Transformer

The *Code Transformer* component transforms code of the FA such that the FA interacts with the mock objects implemented by the *Mock Code Framework* component instead of the real DB. In particular, the methods for DB interactions in Figure 3.1 are mocked with the corresponding methods in Figure 3.2.

Test Generator

The *Test Generator* component uses a test-generation tool, called Pex [93], to generate tests for the DCA under test. We first explain how Pex generates tests and next explain how the test generator used Pex. Pex uses Dynamic Symbolic Execution (DSE) [21, 45, 95] to explore feasible execution paths of the application under test. Pex initially executes the code under test with default inputs. During execution, Pex collects symbolic constraints on inputs obtained from predicates in branch statements (referred to as branch condition) along the execution path. The conjugation of all the branch conditions is referred to as a path condition. Pex selects a branch condition from the path condition of a previously executed path, negates the branch condition, and conjugates it with the prefix branch conditions in the path condition. Pex then generates an input satisfying the resulting condition. Note that the generated input is guaranteed to follow a new path. This preceding process is performed iteratively until no new path is found or pre-configured number of iterations are reached.

The test generator generates records for the mock DB and inserts the generated records in the mock DB to synthesize mock DB states such that a new path can be covered. In particular, the test generator turns the number of records and the values in each record as symbolic inputs (in addition to the inputs for the FA). The test generator then applies Pex to generate concrete values for the inputs for the FA as well as the mock DB state to cover feasible paths in the FA. For the example in Figure 3.1, test-generator inserts a record in the table *Person* with $firstname =' Tom'$ and $age > 25$. When the query at Line 7 is executed, the result set is not empty. Hence the `true` branch of the `else if` statement at Line 11 of Figure 3.1 is covered. The test generator also inserts a record with $lastname =' smith'$, $firstname =' John'$, and $age > 25$. However, this record is deleted by the query executed at Line 5 due

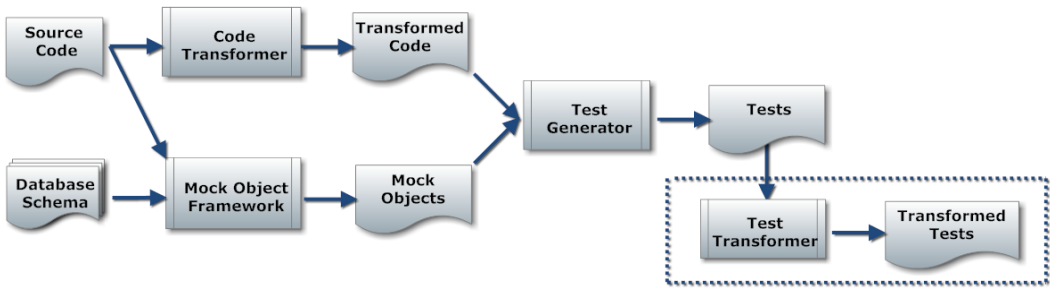


Figure 3.3: Overview of our appraoach.

to which the `true` branch of the `if` statement at Line 10 is not covered, thereby not producing any false warnings.

3.2 Approach

Our MODA approach takes as input the FA of a DCA and the schema of its DB. As output, our approach generates a test suite for the application that achieves high code coverage. Figure 3.3 shows the high-level overview of our MODA approach. MODA consists of four major components: *Mock Object Framework*, *Code Transformer*, *Test Generator*, and *Test Transformer*.

The *Mock Object Framework* component implements the mock objects required to simulate the DB and operations on the DB. The *Code Transformer* component transforms the original FA such that the interactions of the program with the real DB are replaced by interactions with the mock DB. The *Test Generator* component generates tests for a method (in the DCA) selected by a developer. The generated tests are transformed by the *Test Transformer* component to be able to be executed on the original DCA interacting with the DB instead of the mock DB.

3.2.1 Mock Object Framework

The *Mock Object Framework* component implements three major functionalities: (1) parsing input DB schema, (2) mocking the DB, and (3) mocking DB interface APIs.

Parsing DB Schema

The component parses an input schema S and uses the obtained information to initialize the tables in the mock DB. The component parses S to get the tables T and columns C_i in each table $t_i \in T$. In addition, for each table, the component obtains the primary key, foreign key, and the *column constraints* the component obtains from S.

Mocking DB and Operations

The component implements mock DB DS that has the same schema S as the DB. DS contains the corresponding tables T'. DS also provides corresponding operations for operations $S(FC,T,t_j,J,J_T,W)$ $(M_S(FC,T,t_j,J,J_T,W))$, $I(C,V,T)$ $(M_I(C,V,T))$, $U(C,V,T,W)$ $(M_U(C,V,T,W))$, and $D(T,W)$ $(M_D(T,W))$. The pseudo code of the mock select operation $M_S(F,C,T,W)$ is shown in Algorithm 3. The pseudo code of the mock insert operation $M_I(C,V,T)$ is shown in Algorithm 4. The pseudo code of the mock update operation $M_U(C,V,T,W)$ is shown in Algorithm 5, and the pseudo code of the mock delete operation $M_D(T,W)$ is shown in Algorithm 6. The operations allow the data access/manipulation operations on a real DB (using SQL queries) to be simulated on the mock DB DS. These operations are invoked whenever an API for executing a SQL query is invoked in the FA.

Algorithm 3 Mock Select Operation: $M_S(F,C,T,t_j,J,J_T,W)$

Input: List of functions $F =< f_1, f_2, ..., f_n >$, list of Columns $C =< c_1, c_2, ..., c_n >$, list of table names $T =< t_1, t_2, ..., t_n >$ in schema S, table name t_j to be joined, join condition J, type of join J_T, and condition W. Requires $|F| = |C|$.

Output: Result Set of the Query.

$\tau \leftarrow$ Get the set of tables with names T from mocked DB DS
$R_l \leftarrow$ Select records satisfying W from τ
$R_r \leftarrow$ Select records satisfying W from t_j
$R_W \leftarrow$ Apply join operator J_T using join condition J on records R_l and R_r
for all Record r_i in R_W **do**
    $\rho \leftarrow \phi$
    for all $f_j \in F$ **do**
        if F_j is null **then**
            Append element $r_i[c_j]$ to Record ρ
        else
            Append element $f_j(r_i[c_j])$ to Record ρ
        end if
        Add ρ to *ResultSet*
    end for
end for
return *ResultSet*

Mocking DB Interface APIs

The component mocks the API's for interacting with the DB. The component replaces each API class A with a corresponding mock class M_A, and each API method O with a corresponding mock method M_o. These API methods fall into three major categories.

Connection APIs. These APIs help the FA in connecting/disconnecting with a DB. The component

Algorithm 4 Mock Insert Operation: $M_I(C,V,T)$

Input: List of Columns C, List of values V, and table name T in schema S.

Output: Result Set of the Query.

1: $\tau \leftarrow$ Get table with name T from mocked DB DS
2: $\rho \leftarrow$ Create a new record with values $v_1, v_2, ..., v_n \in V$ for columns $c_1, c_2, ..., c_n \in C$
3: Add ρ to τ

Algorithm 5 Mock Update Operation: $M_U(C,V,T,W)$

Input: List of functions F, list of Columns C, table name T in schema S, and condition W.

Output: Result Set of the Query.

1: $\tau \leftarrow$ Get table with name T from mocked DB DS
2: $\gamma \leftarrow$ Select records from τ that satisfy W
3: Update Columns $c_1, c_2, ..., c_n \in C$ of γ with values $v_1, v_2, ..., v_n \in V$

Algorithm 6 Mock Delete Operation: $M_D(T,W)$

Input: Table name T in schema S, and condition W.

Output: Result Set of the Query.

1: $\tau \leftarrow$ Get table with name T from mocked DB DS
2: $\gamma \leftarrow$ Select records from T that satisfy W
3: Delete records γ from τ

replaces these APIs with an empty stub (i.e., a method with an empty method body).

Query Execution APIs. These APIs involve the execution of SQL queries on the DB. The component mocks these APIs with mock APIs invoking the mock operations (discussed in Section 3.2.1) for the corresponding query of the mock DB DS. In particular, the component parses the query passed as an argument to these API methods to get the list of parameters A needed to invoke a corresponding operation on DS. A is $< F, C, T, t_j, J, J_T, W >$ for a $SELECT$ query, $< C, V, T >$ for an $INSERT$ query, $< C, V, T, W >$ for an $UPDATE$ query, and $< T, W >$ for a $DELETE$ query. The corresponding operation (M_S, M_I, M_U, and M_D for $SELECT$, $INSERT$, $UPDATE$, and $DELETE$ SQL queries, respectively) in DS is invoked with arguments A. The pseudo code for mock query execution is shown in Algorithm 7.

Helper APIs. Helper APIs are APIs for connecting with the DB, or execution of SQL queries. We have implemented the mock APIs corresponding to these APIs based on their available specifications at MSDN [99].

3.2.2 Code Transformer

The *Code Transformer* component transforms the original FA P to P_t such that the invocation of each DB interface API method O (described in Section 3.2.1) in P is replaced by an invocation of corresponding mock method M_O in P_t and each reference to API class A is replaced by the reference to corresponding

Algorithm 7 *MockQueryExecution*(Q)

Input: An SQL query Q.
Output: Result Set of the Query.

```
1:  q ← parse query Q
2:  if q is Select(F,C,T,W) then
3:          invoke operation M_S(F,C,T,W)
4:  end if
5:  if q is INSERT(C,V,T) then
6:          invoke operation M_I(C,V,T)
7:  end if
8:  if q is UPDATE(C,V,T,W) then
9:          invoke operation M_U(C,V,T,W)
10: end if
11: if q is Delete(T,W) then
12:         invoke operation M_D(T,W)
13: end if
```

mock class M_A. The program transformation replaces the interaction of the FA from the DB to the mock DB *DS*.

3.2.3 Test Generator

The *Test Generator* component uses Pex [93] for generating tests for the transformed DCA P_t. Pex generates tests for a C# program using Dynamic Symbolic Execution (DSE). DSE iteratively generates tests that cover feasible paths in a program. The component extends the functionality of Pex so that Pex generates records to be inserted in the mock DB before a program run (in addition to program inputs). In each iteration, the component generates new tests (that include records to be inserted in the mock DB) that follows a new path in the FA. To generate tests, the component records the branch condition at every branching point in the path followed by the input generated in the previous iteration. The component then negates one of the branch condition to get a new path condition. The component then employs a constraint solver to solve the path constraint and generates a new test. Algorithm 8 shows the pseudo code for mock-DB state generation. At Line 1, the component chooses the number of records to be inserted in *DS*, while at Line 5 *Test Generator* chooses the values for each record. The component chooses these values by solving constraints in the DCA for covering a new path. In addition, the component considers the *column constraints* K_i for generating a value for Column C_i.

3.2.4 Test Transformer

It is sometimes desirable to execute the generated tests on the original DCA (which with the original DB). The *Test Transformer* transforms the tests generated by the *Test Generation* component so that they can execute on the DCA having P as the FA that interacts with the original DB. The mock insert operations M_I (from Line 6 of Algorithm 8) to insert records in the mock DB are replaced by execution of a corresponding INSERT query on the real DB. In particular, each operation $M_I(C,V,\tau)$ (in the generated

Algorithm 8 *DatabaseStateGeneration*(S)

Input: Schema S.

Output: A DB state for covering a new path in program P_T.

```
1: n ← Allow Pex to choose a value between 0 and MAX
2: for all Table τ in S do
3:     for i = 0 to n do
4:         C ← Columns in table τ
5:         V ← Allow Pex to choose a value
6:         invoke M_I(C, V, τ)
7:     end for
8: end for
```

tests) is replaced by a query "$INSERT\ INTO\ \tau(c_1, c_2, ...c_n)\ VALUES\ (v_1, v_2, ...v_n)$".

3.3 Empirical Studies

In our empirical studies, we generated tests for two software systems: a real-world medical device DCA and an open source DCA. Through our empirical studies, we intend to address the following research questions

RQ1: How effectively can MODA help in improving code coverage of the FA of DCAs as compared to an existing state-of-the-art test generation tool?

RQ2: How effectively can MODA help in reducing the number of false warnings that are generated by an existing mocking technique?

To address RQ1, we compare the branch coverage achieved by tests generated using MODA with the branch coverage achieved using Pex in the presence of an empty DB. We refer to the latter approach as "Pex approach" in the rest of this paper. To address RQ2, we compare our approach with the parameterized-mocking approach [111]. To simulate the parametrized-mocking approach (referred to as PMock), we transform each result set of a query execution statement present in the original FA of the DCA under test into a `PexChoose` statement. Pex generates the values of the result set so that various parts in the program are covered. For example, the program in Figure 3.1 is translated to the one shown in Figure 3.4. Figure 3.4 shows only the changes made to the program. Line 7 of Figure 3.1 is transformed to the two lines shows in Figure 3.4. Since Pex cannot effectively generate values for non-primitive-type arguments, we synthesize a factory method for our mock `MockSqlDataReader` class. To compare MODA and PMock, we manually inspect the generated tests to find false warnings, i.e., failing tests that should not fail in real scenarios (when the FA interacts with a real DB).

We next describe our experience of applying MODA, PMock, and the Pex approach on the two software systems.

3.3.1 Medical Device Software System

We applied our approach on a point-of-care handheld medical assistant. The software system in the assistant is intended to be installed at smart hand-held devices (such as PDAs) and to communicate injury and treatment data with a remote DB server. Due to confidentiality concerns, we cannot disclose the identify of this device, as well as its details. In the rest of the paper, we refer to it as "the device".

To focus our study on DB interactions, we choose a component of the device for test generation. This component consists of a sequence of fixed GUI screens that can be categorized into two types: some screens retrieve information from the DB and populate it for a user; the remaining screens assist the user in editing clinical data and uploading the resultant data to the DB. The size of the chosen component is about 100,000 lines of C# code (LOC), in which 62 SQL queries can be found. These 62 queries spread over 12 classes in the component, which totally consist of 2,994 LOC. Among these 62 queries, 54 of them are `SELECT` queries, while the rest 8 are either `INSERT`, `UPDATE`, or `DELETE` queries.

MODA could generate non-empty result sets for only 39 `SELECT` queries, leaving the rest 15 `SELECT` queries not covered. The reason of not all `SELECT` queries being covered can be ascribed to a fault that our approach detects in the component. While interpreting a SQL query in the component, our approach found illegal characters in the syntax of this query and threw a parsing exception. Since the execution of this problematic query lies on all execution paths leading to the 15 `SELECT` queries, our approach did not cover these queries.

Table 3.1 shows the results of our studies. Column *Subject* shows the subject, Column *C* shows the number of classes tested. Column *LOC* shows the total number of LOC in these classes. Column *M* shows the total number of methods that we tested. Columns C_M and C_P show the branch coverage achieved by tests generated using MODA. and the Pex approach, respectively. Tests generated using MODA were able to achieve 91.9% branch coverage of the code under test in contrast to 64.4% achieved by the Pex approach. This coverage measurement excludes parts of the component that are not executable due to the above-mentioned fault.

Although PMock approach did not result in any false warnings, one advantage that distinguishes our approach from PMock is the capability of detecting faults associated with SQL queries. During our empirical study, our approach detected 3 faults including the one previously mentioned in this section. Another fault that our approach detected indicates that a column name was referred to in lowercase by one of the SQL queries, while the input DB schema defines the column in uppercase. According to the SQL standard, column and table names in a DB can be configured to either case sensitive or insensitive. Hence, a mismatched column name being referred by SQL queries could cause a system failure under certain DB configurations. The last detected fault can be categorized as unsafe compositions of SQL queries. In particular, a method in the component takes as input a list and then appends each element of the list to the WHERE condition of a `SELECT` query as an `AND` clause. Since the input list can be empty, this method may construct `SELECT` queries with invalid WHERE conditions, such as *SELECT*

Table 3.1: Subjects and results of our empirical studies.

Subject	C	LOC	M	C_M	C_P	Inc
Medical Device	12	2994	46	91.9	64.4	27.5
Odyssey	2	1227	53	89.7	77.2	12.5

```
...
7   var chooser = PexChoose.FromCall(this);
7   MockSqlDataReader myReader = chooser.
        ChooseValue(<SqlDataReader>)();...
```

Figure 3.4: Program transformed for using parameterized mock object for the example in Figure 3.1.

* $FROM\ X\ WHERE\ (X.Y = 1)\ AND\ ()$. None of these faults could be detected by PMock, as these techniques do not interpret SQL queries during test generation. Two of these faults were not detected by the Pex approach since the execution of the queries (that expose faults) were dependent on the result set returned by a previously executed query. Hence, the dependent queries could not be executed using the Pex approach.

3.3.2 Open Source Software System

We conducted a study on an open source library *Odyssey*[1]. *Odyssey* provides a collection of windows presentation framework controls. In our study, we considered the *PasswordSafe* module of *Odyssey*. We generated tests for two classed `DAL` and `BizContext` (using MODA and Pex approach) in the *PasswordSafe* module. The two classes contain 1227 LOC in total. *DAL* implements the Data Access Layer of *Odyssey* to exchange data from business objects with the DB, while `BizContext` implements a business context to exchange business objects with `DAL`. `DAL` contains 33 methods containing SQL queries. The rest of the methods are helper or observer methods. We generated tests for the 33 methods that contain at least one query. The class `BizContext` does not itself contain any SQL query but executes the methods in `DAL`. There are 20 methods in the class `BizContext` that indirectly interact with the DB (by invoking methods of the class `DAL`). We generated tests for all these methods. Row 2 of Table 3.1 shows the results obtained for *Odyssey*.

In total, our approach achieves a high branch coverage (89.7%) of the methods under test, generating non-empty result-sets for all the encountered queries . In contrast, the Pex approach achieves 77.2% branch coverage. PMock also achieves 89% branch coverage of the code under test. PMock does not generate any false warnings for the `DAL` class since the class contains methods with at most one query. However, PMock results in false warnings for 8 methods of the class `BizContext` out of the total 20 methods that interact with the DB. In contrast, our approach does not generate any false warnings. The

[1] `http://odyssey.codeplex.com/`

```
void SaveCategory(Category category){
...
    if (category.Id == 0) Dal.CreateCategory(category);
    Dal.UpdateCategory(category);
    category.IsModified = false;
    ...
}

public void UpdateCategory(Category category){
   int n = Execute("update Category ...", category.Name,
    category.Order, category.Id);
   if (n != 1) throw new DBEntityNotUpdatedException();
}
```

Figure 3.5: Example in `Odyssey` code interacting with a DB.

detailed results and the subject classes are available on our project webpage[2].

Figure 3.5 shows a method in the class `BizContext` for which PMock results in false warnings. The method `SaveCategory` takes as an argument an object of type `Category`. If `Category.Id` is 0, the method invokes `CreateCategory` method of `DAL`, which inserts a record in the `Category` table. The record is then updated by invoking `UpdateCategory` method. The method `CreateCategory` throws an exception when the execution of the insert query returns 0, i.e., no record is inserted in the DB. Similarly, the method `UpdateCategory` throws an exception when the execution of the update query does not update any record. PMock results in tests with `Category.Id`$= 0$. The execution of the test causes an exception thrown by the `UpdateCategory` method even when the method `CreateCategory` did not throw any exception (i.e., `CreateCategory` creates a new record and inserts it in the DB). Since the record was created by the `CreateCategory` method (and should exist in the DB when `UpdateCategory` is invoked), the record should be updated by the `UpdateCategory` method in real situations.

In summary, results of our empirical studies show: **RQ1**. MODA achieves 15% higher branch coverage (on average) as compared to the Pex approach. In addition, the approach detects 2 faults that could not be detected by the Pex approach. **RQ2**. MODA does not result in any false warnings, while PMock results in false warning for 8 methods of *Odyssey*. In addition, MODA detects 3 faults (in the device) that could not be detected by PMock.

3.4 Discussion

In this section, we discuss some issues of the current implementation of our approach and how they can be addressed in future work.

[2] `http://sites.google.com/site/asergrp/projects/testdatabase`

3.4.1 Other Test Generation Techniques

Our current implementation uses Pex for test generation. Pex is a DSE-based test generation tool. Our approach can be seamlessly integrated with other approaches that are built on top of Pex and other DSE tools such as our DiffGen and eXpress approaches presented in Chapters 4 and 5, respectively. In addition, our approach can in general be used with test-generation tools based on other techniques such as evolutionary test generation [112] and random test generation [76]. The test-generation tools need to be extended to generate records to be inserted in the mock database in addition to program inputs.

3.4.2 Absence of Database Schema

Our approach requires as input database schema. However, our approach can be used without using the database schema. The tables in the database and the columns can be inferred by syntactically scanning the source code. However, if a column in the table is never used in the source code, only a partial schema could be inferred by scanning the source code. Although the partial schema would not affect test generation, the transformed tests might not be able to executed on the real database.

3.4.3 Mocking Other Environments

The spirit of our approach in mocking missing DBs can be applied for mocking other types of program environment, as long as the characteristics and behavior of such an environment can be precisely specified. We can use a test-generation tool to create an initial environment state so that various parts of a program (interacting with the environment) can be covered. The mock objects then keep track of environment states when different operations in the program are performed on the environment. For example, a file system and its operations can be mocked. A test generation tool can be used to generate an initial file system state (i.e., certain files can be added to the file system) so that various parts in the program can be covered. The mock objects then keep tract of the file system state (i.e., files in the file system) as different operations are performed on the file system in the program.

3.4.4 Complexity of Query Parsing Code

MODA uses a parser to parse the query code at runtime. Due to the complexity of parsing code, path constraints might get too complex to be solved by state-of-the-art constraint solvers used by DSE. To address the issue, we turn off the instrumentation of the parsing code so that no constraints are collected in the query parsing code. As an effect, MODA would turn the symbolic inputs involved in a query into concrete values observed at runtime, resulting in some uncovered branches.

3.5 PRIEST vs MODA

Both MODA and PRIEST have their own limitations but the limitation of one is the strength of another. Hence, we believe that our PRIEST approach (presented in Chapter 2) is complementary to our MODA approach.

On one hand, since MODA is based on DSE, MODA has various inherit limitations [22] of DSE such as scalability and constraint solving. Our eXpress approach (presented in Chapter 5) and other pieces of work [124, 125] show that DSE achieves a higher coverage when manually written tests are seeded. In situations when actual database records cannot be released to the testing teams, records anonymized by PRIEST can be used as seeds and can help MODA achieve higher code coverage.

On the other hand, as discussed in Chapter 3, strict requirements on privacy, result in loss of code coverage. MODA can complement PRIEST in generation of additional DB records for covering additional branches that could not be covered by PRIEST.

3.6 Related Work

Related work of our approach falls into two major categories: testing DCAs and mock object generation.

3.6.1 Testing DCAs

Emmi et al. [39] propose an approach that generates test inputs for DCAs. The approach extends DSE by considering constraints obtained from SQL queries in addition to the constraints from the FA. Since this approach inserts records in a real DB, it necessarily requires the DB to be in place. In contrast, our approach is designed to be used when the DB is not available. In addition, the tests generated using our approach can be executed on the mock DB, which is often desirable [88] for efficiently executing the tests.

Willmore and Embury [118] propose an approach that requires developers to specify what DB preconditions are desirable for testing in the format of constraints. Only DB states satisfying such constraints are generated in the subsequent testing process. In contrast, our approach does not require any specifications from the developers to generate initial (mock) DB states.

The AGENDA framework [24] uses some predefined inputs to generate DB states and various techniques [10] to generate test inputs for a DCA randomly. Since these techniques use predefined (or randomly generated) inputs, instead of using control- and data-flow information of the program, to generate test inputs, these techniques may achieve low code coverage. In contrast, our approach is capable of considering program structures and queries present in the program. Therefore, our approach is likely to achieve higher code coverage

Veanes et al. [114] propose an approach for generating test inputs for SQL queries. This approach generates test inputs (i.e., a DB state) for one query in isolation and does not consider the interaction

that the query has with the FA or with other queries in the program. In addition, the developers have to provide a certain test criterion (such as the result set of the query is non-empty). Reverse query processing [13] generates a possible DB state from a query and result set of the query provided as input. In contrast to these approaches, our approach mocks the real DB and focuses on generating test inputs for the DCA. The testing criterion is to cover all branches in the program. DB states are generated to cover various parts of the program. Hence the testing criterion for individual queries is obtained from the source code and does not have to be explicitly provided.

Kapfhammer and Soffa [62] propose testing criteria for DCAs. The criteria use data flow information associated with entities in the DB. Cabal and Tuya [102] also propose a technique to help improve a test suite with the purpose of detecting faults in SELECT queries. Halfond and Orso [50] propose a testing criterion called command form coverage. This criterion requires coverage of all the command forms that a given FA of a DCA can issue to the DB. In addition, there exist approaches for testing SQL queries [113, 117] and for finding security vulnerabilities [101]. All the preceding approaches are complementary to our approach.

3.6.2 Automated Mock Object Generation

Saff et al. [88] propose the idea of using mock objects for environment (including DB) to improve the efficiency of test execution. In particular, the approach factors the existing test suite by mocking the environment. The approach captures the environment states before and after the execution of a test and generates a mock with the same behavior for a particular test. However, mock objects generated by this approach mock the minimal behavior exercised by the test suite. The approach helps in efficiently executing the existing tests but cannot be used for test generation. In contrast, our approach helps in generating a test suite that achieves high code coverage.

Tillman and Wolfram [111] propose the notion of parameterized mock objects. Parameterized mock objects can be used to generalize mock objects and generate various possible return values automatically for mock methods. Developers can write a parameterized mock object so that a single call to a mock method of a mock object can return different values expected by the code under test. Although parameterized mock objects can be used to cover various parts of the program, the test inputs would not be valid in real situations resulting in false warnings. In contrast, our approach does not result in false warnings. In addition, the tests generated by our approach can be executed along with the original DCA.

Marri et al. [72], presented an empirical study to analyze the use of parameterized mock objects in unit testing with parameterized unit tests. Their study showed that using a parameterized mock object can ease the process of unit testing and identified challenges faced in testing code when there are multiple APIs that need to be mocked. Taking this study into account, our approach uses parameterized mock objects in test generation for DCAs. Moreover, our approach establishes an example on how to handle multiple APIs when mocking DB.

3.7 Conclusion

We presented an approach for testing a DCA. Our approach mocks the DB and its operations with a parameterized mock DB. In this way, our approach can generate, based upon a DSE-based test generation tool, necessary tests (including initial DB states) to cover most parts of the FA of the DCA under test. To evaluate the effectiveness of our approach, we conducted empirical studies on a medical device DCA and an open source DCA. The results show that our approach can achieve high code coverage of the FA with no false warnings whereas existing mocking techniques result in false warnings.

Chapter 4

Effective Regression Unit-Test Generation

4.1 Introduction

A software application[1] is developed in many phases including requirements phase, design phase, implementation phase, testing phase, and maintenance phase. A successful software application continues to evolve throughout its lifetimes undergoing various kinds of changes. Software maintenance is the process of making changes to an evolving software application throughout its lifetime (after its first release). Since software lifetimes are often long, software maintenance is the longest phase in the software development lifecycle and is the most expensive activity of software development, comprising from 50% to 90% of the overall software development costs [92].

Owing to long software lifetimes and the cost involved in maintaining a software application, software maintenance is an important software development phase. In addition, software maintenance is crucial for business success since the failure to change a software application quickly and reliably can result in losing business opportunities. However, changing an application quickly and reliably is challenging since software maintenance is error prone. Maintenance (counter intuitive to its name) can itself result in reducing the software quality. The changes made to a software application can result in unintended side effects (regression faults).

Regression testing is the activity of retesting a software application when developers make changes to the software application during software maintenance phase. Regression testing can give confidence to the software developers that (1) the changes they made are intended (2) and the changes do not introduce any unintended side effects (regression faults). If regression testing is not done effectively, some dormant faults can remain in the application and can manifest themselves 'in the field', leading

[1] In this chapter, we use the term software application for simplicity instead of using FA of a DCA. The approach presented in this chapter is applicable for both a non DCA and the FA of a DCA.

to customer dissatisfaction. Moreover, the later a fault is discovered, the more expensive it is to fix. Previous studies [14, 15] have found that faults found 'in the field' cost more than five times as much to correct as those faults corrected earlier. As a result, regression testing is an important activity not only to increase software quality but also to reduce cost of software maintenance.

Regression testing of a software application is carried out by executing an existing regression test suite (that was developed for a previous version of the application) against the current version of the application. The failing tests (that pass for the previous version) indicate behavioral differences[2] between the two versions of the FA. Software developers can then inspect the failing tests to find whether the tests fail due to intended changes or regression faults. As a result, effectiveness of regression testing depends on the effectiveness of the regression test suite in finding behavioral differences.

In some situations (such as in legacy systems), when developers do not have an existing test suite, it is expensive for developers[3] to write regression tests from the scratch. Even if the developers have an existing test suite, the test suite may not be sufficient for regression testing. In particular, the tests in the existing test suite may not be able to detect a fault that a developer introduced while modifying the application because the tests may not exercise the situations in which the two versions behave differently. Therefore, to detect regression faults, the developers (or testers) often need to augment the existing test suite with new tests. Since manual testing is labor intensive, various automated test generation techniques [21, 26, 45, 46, 63, 95, 115] are proposed, for generating a test suite that can achieve high structural coverage. However, generating tests with high structural coverage on both versions of a software application is not sufficient to expose all the behavioral differences between them.

Evans and Savoia [41] addressed the problem of detecting behavioral differences by generating tests (using `JUnit Factory` [61]) that achieve high structural coverage separately for an old and new versions of the software application under test. They detected behavioral differences by cross-running the test suites on the two versions of the software application under test. High structural coverage of the two application versions might ensure that the modified part of the software application is executed. However, only the execution of the modified part is not sufficient to expose behavioral differences according to the PIE model [116]. According to the PIE model of error propagation, a fault can be detected by a test if a faulty statement is executed (E), the execution of the faulty statement infects the state (I), and the infected state (i.e., error) propagates to an observable output (P). A change in the new version of a program can be treated as a fault and then the PIE model is applicable for effect propagation of the change. Hence, even if we have a test suite that has full structural coverage of the two program versions under test, we can only guarantee the condition E of the PIE model, which is not sufficient to detect behavioral differences.

We propose an approach called DiffGen that takes two versions of an application, and generates

[2] A behavioral difference between two versions of an FA can be reflected by the difference between the observable outputs produced the execution of the same test on the two versions.

[3] We use the term developers to indicate the people testing the software application.

regression tests that check if the observable outputs and states of the two versions differ. Each test that fails on execution exposes a state infection (I) or a modified behavior (P). To ensure that the modified part of the program is executed, and to increase the chances of detecting behavioral differences, we instrument the given program versions by adding new branches in the source code such that the coverage of these branches ensures at least the condition I of the PIE model. Therefore, if a test generation tool is able to generate tests to cover these branches, application state is guaranteed to be infected[4]. A developer can inspect the state or behavior deference to see if the state or behavior difference is due to a regression fault. If they indeed occur due to a regression fault, the developer can fix the regression fault.

DiffGen makes the following main contributions:

Approach. We propose an approach for generating regression tests that help in detecting state infections and behavioral differences between two versions of a given Java Class by checking observable outputs and receiver object states.

Evaluation. We evaluate our approach on detecting state infections[5] between 21 classes (taken from a variety of sources) and their versions. The experimental results show that our approach is more effective than a state-of-the-art approach [41] in satisfying condition I of the PIE model.

The rest of the chapter is organized as follows. Section 4.2 explains our approach through an illustrative example. Section 4.3 presents key aspects of our approach. Section 4.4 discusses evaluation of DiffGen and presents the experimental results. We discuss some of the limitations of our approach, and how these limitations can be addressed in Section 5.6. Section 5.7 describes related work. Finally, Section 4.8 concludes.

4.2 Example

We next illustrate our DiffGen approach with the aid of an example. DiffGen takes as input two versions of a Java Class and generates a suite of regression tests. The state differences between the two classes are exposed by executing the generated test suite. Tests that fail are the ones that represent inputs for which the states of the two versions have changed, while the passing tests represent the ones for which there is no state infection.

Consider the example of a Binary Search Tree shown in Figure 4.1. Figure 4.1 shows an old version of the class `BST`, while in a new version of the class `BST`, the statement at Line 14 (underlined in Figure 4.1) has been modified to the one shown in the comment at Line 14. In the old version, a new

[4]If two versions of an application are executed with the same inputs, the application state is infection if the value of some field in the application is different between the two versions. If the field is public the infection is a behavior difference since it can be observed. If the field is private, the state infection may be manifested as a behavioral difference by invoking an observer method on the object containing the infected field.

[5]Although a state infection does not guarantee to be manifested as a behavioral difference (or a failure due to regression fault), it is still an indication of a possible regression fault. Hence, an approach that is effective in infecting program state is more likely to find more regression faults.

```
1    class BSTOld implements set{
2       Node node;
3       int size;
4       static class Node{
5          MyInput value;
6          Node left;
7          Node right;
8       }
9       public BSTOld() {.....}
10
11      public boolean insert(MyInput m){
12         Node t = root;
13         while(true){
14            if(t.compareTo(m.key)>0)//if(t.compareTo(m.key)>=0
15               if(t.right == null){
16                  t.right = new Node(m);
17                  break;
18               }
19               else
20                  t = t.right;
21            else
22               if(t.left == null){
23                  t.left = new Node(m);
24                  break;
25               }
26               else
27                  t = t.left;
28         .....
29         }
30      }
31      public void remove(MyInput m){.....}
32      public void contains(MyInput m){.....}

34      .....
35   }
```

Figure 4.1: The `BSTOld` class as in an old version. In a new version, Line 14 is changed to the one shown in the comment.

node is inserted to the left subtree, if the key of the root node is *less than* that of the newly inserted node. In the new version, the new node is inserted to the left subtree, if the key of the root node is *less than or equal to* that of the newly inserted node. Even if we generate tests (for both the versions of `BST` classes) that cover all the branches in both classes, we may not be able to detect behavioral or state difference due to the changed statement at Line 14. As a result, if the changed statement at Line 14 is a regression fault, it cannot be detected. Suppose that initially the tree contained only one node with Key 5 as shown in Figure 4.2. Later nodes with Keys 3, 6, 2, and 7 are inserted. In particular, the insertion of a node with Key 3 covers the `if` branch at Lines 22-25. The insertion of a node with Key 6 covers the `if` branch at Lines 15-18. The insertion of a node with Key 2 covers the `else` branch at Lines 26-27, and finally the insertion of a node with Key 7 covers the `else` branch at Lines 19-20.

Therefore, the insertion of these four nodes in the order covers all the branches in the `insert` method in both versions of `BST`, and also results in the same tree as shown in Figure 4.2. However, only if we insert a node with a key equal to the key of an existing node in the tree, the state of the tree differs. For example, the insertion of nodes with Keys 3 and 5, respectively, in the two versions of the `BST` class results in a different tree as shown in Figure 4.3. A structural-coverage-based test generation tool might not generate such a test because the tool primarily targets at covering various branches and not specifically exposing behavioral or state differences. Hence, it can miss various regression faults.

To detect behavioral and state differences, DiffGen first conducts a fast syntactic check to detect all the methods that have been changed over the two versions syntactically. DiffGen detects that the `insert` method has been changed syntactically in the new version. DiffGen then synthesizes a test driver for a test generation tool such as jCUTE [94] (or JUnit Factory [61]). In this section, we give an example of a synthesized driver for jCUTE. The synthesized driver for the `BST` example is shown in Figure 4.4. jCUTE is a symbolic-execution-based test generation tool, which tries to generate tests to explore feasible paths in the program under test. In the test driver shown in Figure 4.4, we first invoke the constructors of both `BSTOld` and `BST` (the new version of BST) to create objects `bstOld` and `bstNew`, respectively. Next we invoke different methods for both the `bstOld` and `bstNew` objects inside the `switch` statement at Line 6 in Figure 4.4. Note that `cute.Cute.input.Integer()` at Line 6 generates various integer values trying to explore all feasible paths in the driver and eventually generates tests that contain all combinations of method sequences (whose lengths are up to five, reflected by the `for` loop in Line 5). Note that inside all the case statements, we invoke the same method for both the `bstOld` and `bstNew` objects with the same method argument. For example, we invoke the `insert` method for both `bstOld` and `bstNew` inside the `case:0` branch at Line 7 of Figure 4.4. These symbolic method arguments help in the exploration of all feasible paths within the methods of the class under test. If a method has a non-primitive argument, we make a deep copy of the argument and invoke the methods in the two versions with different copies of the argument. Note that we generate method arguments using jCUTE. jCUTE generates the arguments in such a way that these arguments help to cover as many branches as possible in the program under test.

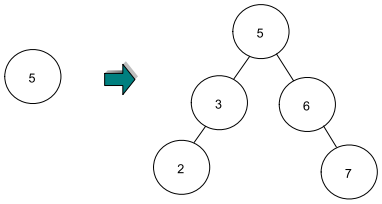
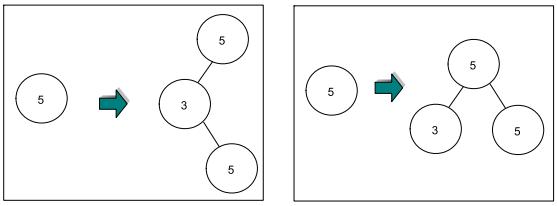


Figure 4.2: The `BST` object states before and after nodes with Keys 3, 6, 2, and 7 are inserted, respectively.



(a) (b)

Figure 4.3: The `BST` object states before and after nodes with Keys 3 and 5 are inserted, respectively for **(a)** the old version of class `BST` and **(b)** the new version of class `BST`.

After building the object state with various method call sequences, we invoke the methods that were modified between the two versions. As shown at Lines 24 and 25 of Figure 4.4, we invoke the `insert` method for both versions of the program under test. We then compare the return values obtained by invoking the insert method on the two objects `bstNew` and `bstOld`, as shown in Line 26. We finally compare the object states of the resulting `BSTNew` and `BSTOld` objects. As shown in Lines 28 and 30, we compare the object fields `size` and `Node`. If jCUTE is able to cover the branches at Line 27, we have detected a behavioral difference and if jCUTE is able to cover one of the branches at 29 and 31, we have detected a state infection. In the instrumentation process, we also change the modifiers of all the fields transitively reachable by objects of the classes under test to `public`, so that we are able to compare these fields directly to detect state infection. In the test generation phase, DiffGen uses jCUTE to generate a test suite for the driver (in Figure 4.4) synthesized in the previous phase. In the test suite generated by jCUTE, the failing tests cover at least one of the branches at Lines 27, 29, and 31 and hence expose the behavioral differences or state infection between the classes under test.

4.3 Approach

DiffGen takes as input two given versions of a Java class. For each modified class in the class, DiffGen generates regression tests, which on execution expose state differences between the two versions.

```
1  class BSTTestDriver implements set{
2   public static void driver(){
3     BSTOld bstOld = new BSTOld();
4     BST bstNew = new BST();
5     for(int i=0; i< 5; i++)
6       switch(cute.Cute.input.Integer()){
7         case 0:
8         MyInput key = cute.cute.Input.Object(MyInput);
9         bstOld.insert(key);
10        bstNew.insert(key);
11        case 1:
12        key = cute.cute.Input.Object(MyInput);
13        bstOld.remove(key);
14        bstNew.remove(key);
15        case 2:
16        key = cute.cute.Input.Object(MyInput);
17        bstOld.contains(key);
18        bstNew.contains(key);
19        .....
20        default: break;
21      }
22    }
23    MyInput key = cute.cute.Input.Object(MyInput);
24    boolean b1 = bstOld.insert(key);
25    boolean b2 = bstNew.insert(key);
26    if(b1 != b2)
27      Assert(false);
28    if(bstOld.size != bstNew.size)
29      Assert(false);
30    if(bstOld.node.equals((bstNew.node))//Checks if its a deep copy
31      Assert(false);
32   }
33 }
```

Figure 4.4: Test driver synthesized by DiffGen for the two versions of the `BST` class.

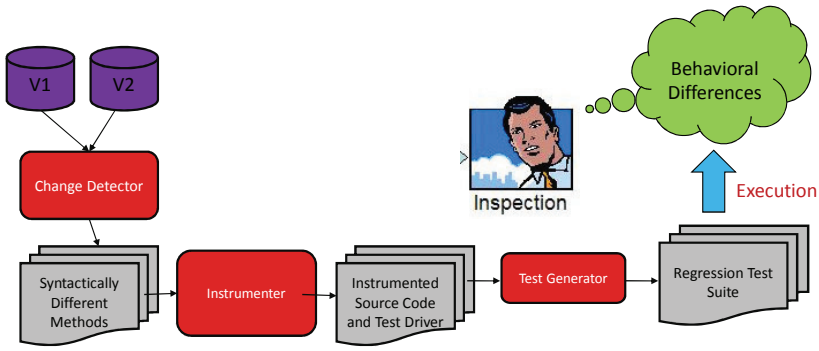


Figure 4.5: Overview of of our DiffGen approach.

We next describe our DiffGen approach in detail.

Figure 4.5 shows the various components involved in DiffGen. Given two versions of the class under test, first the *Change Detector* component detects all the syntactically different methods in the two versions of the class under test. *Instrumenter* component then instruments the source code and generates a test driver. The *Test Generator* component then generates a regression test suite. The generated regression test suite is executed (in the *Test Execution* phase) to detect state differences.

4.3.1 Change Detector

The *Code Detector* component first detects a set of all the modified classes between the two application versions. We refer to all these classes as classes under test. For all the corresponding method pairs in the two versions of a class under test, the *Change Detector* checks for textual similarity between the two versions. The *Change Detector* selects the corresponding method pairs by matching their names and signatures. If the two methods in a method pair are textually the same, they cannot be semantically different and thus are considered to have the same behaviors. The *Change Detector* filters out all the pairs with textually the same methods, and selects all the methods that are different textually. For the BST class (Figure 4.1), *Change Detector* detects that the method `insert` has been modified.

Instrumenter

The *Instrumenter* component instruments the source code of the two versions of the class under test, and synthesizes a test driver for the *Test Generator* component to generate tests. The *Instrumenter* component first changes the modifier of all the fields transitively reachable from objects of the two given classes to *public*. This mechanism enables us to compare the object states after a sequence of method invocations on the objects of the two versions of the input class under test directly by comparing the object fields. We synthesize two different types of drivers for two test generation tools (jCUTE and JUnit

Factory) to generate tests. The first kind of driver that we synthesize for jCUTE can in general be used for other symbolic-execution based tools, while the second kind of driver that we synthesize for JUnit Factory can be used for other test generation tools. We next describe the two kinds of drivers in detail.

`jCUTE` Test Driver. The `jCUTE` test driver synthesized by the *Instrumenter* component for the `BST` example (Figure 4.1) is shown in Figure 4.4. In the driver, we first make new objects of both versions of the class under test by invoking one of the constructors. If a constructor requires one or more primitive arguments, we first generate symbolic primitive argument values by using jCUTE and pass the same argument values to invocations of both constructors. If the constructors require a non-primitive argument, we generate a symbolic argument object by using jCUTE, which generates a symbolic object state for the non-primitive argument. Once we have the objects created for the new and old versions of the class under test, we make a sequence of method invocations on the objects, For each method invocation, we pass the same arguments. If a method requires a non-primitive argument, we generate it using jCUTE, make a deep copy of the generated argument, and pass a different copy to invoke the methods in the two versions. If the non-primitive argument type contains static variables, we make a deep copy of the static variable as well, and replace the values of these static copies after the execution of the first method. We set the maximum length of the sequence of method invocations to be the number of methods in the original version of the class under test. If a change is found only in the constructor of the class under test, we do not invoke any sequence of method calls.

Once we have made a sequence of method invocations, we invoke the methods that were found to be textually different by the *Change Detector* component. If any of these methods has a non-void return type, we compare the return values of the method. If the return values are different, we know that there is a behavioral difference between the two versions of the class under test. Therefore, we add a branch in the driver containing `assert(false)`(as shown in Line 27), which is executed if the return values of these methods are different. If the only textually different method is the constructor, we do not invoke it again. Finally, we check the equivalence of resulting states of the two objects. If the same series of method invocation results in different receiver object states, we know that the behaviors of two methods are different. If the resulting receiver object states are the same, it means that the behaviors of the two versions are the same for this given input. For checking the equivalence of the resulting states, we compare the fields inside the class under test. We add branches in the driver containing `assert(false)`(as shown in Lines 29 and 31 of Figure 4.4), which are executed if any pair of the fields is not equivalent in the two versions. The execution of any of the branch at Line 27 guarantees exposing behavior differences, while the execution of one of the branches at Lines 29, and 31 of Figure 4.4 ensures state infection. Therefore, the instrumentation converts the problem of finding behavioral and state differences between two versions of a class into a branch coverage problem. If we are able to generate tests that cover any of the branches at Lines 27, 29, and 31, we are able to find a state infection or a behavioral difference between the two versions.

JUnit Factory Test Driver. The JUnit Factory test driver synthesized by the *Instrumenter* component

```
1   public class BSTJUFDriver{
2      public void compareInsert(BSTOld oldBST,
            MyInput input){
3         BST bstNew = new BST();
4         bstNew = copyObject(oldBST);
5         boolean b1 = bstOld.insert(input);
6         boolean b2 = bstNew.insert(input);
7         if(b1 != b2)
8            Assert(false);
9         if(bstOld.size != bstNew.size)
10           Assert(false);
11        if(!bstOld.root.equals(bstNew.root))
12           Assert(false);
13     }
14  }
```

Figure 4.6: Test Driver synthesized for JUnit Factory

for the `BST` example (Figure 4.1) is shown in Figure 4.6. The driver class contains one method for every changed method inside the class under test. These methods are a kind of parameterized unit tests [93]. Each method has an object of the original version of class under test as an argument. The rest of the arguments of the method are the same as the arguments of the changed method in the class under test. For example, in Figure 4.6, the method `compareInsert` compares the behaviors of the two versions of the method `insert`. An argument of `compareInsert` is an object (`bstOld`) of `BSTOld`. Inside the body of `compareInsert`, we make a new object of `BST` (Line 3) and deep copy the field of `bstOld` to the fields of `bstNew` (Line 4). We then invoke the changed method from the objects of the two versions of class under test (Line numbers 5 and 6). We next compare the return values and the resulting object states of `bstNew` and `bstOld` with new branches (Lines 8, 10 and 12) .

The formats of the outputs such as the receiver-object state can be different for the method under test and the reference method. In such situations, developers can provide the `equals` methods for mapping objects that are deemed to be equivalent and are needed in our test drivers.

4.3.2 Test Generator

The Test Generator component can use either jCUTE [94] or JUnit Factory for test generation. jCUTE is a Dynamic Symbolic Execution(DSE)-based test generator tool. DSE generates inputs that guide the execution of an application to alternate paths. In particular, symbolic execution collects the constraints from all the branching points involved in a path and solves these constraints to generate an input for a particular path. Whenever the symbolic state is too complex to be handled by a constraint solver, concrete values are used to replace the symbolic values. JUnit Factory is a commercial automated characterization test generator based on Agitar [16].

The input to the *Test Generator* component is the instrumented code and the test driver generated by the Instrumenter component. jCUTE (or JUnit Factory) uses the test driver to generate regression

tests. jCUTE tries to generate test inputs to cover all the branches in the test driver and the two versions of the classes under test. In particular, it solves the path constraints to explore feasible paths in the classes under test to generate concrete test inputs. Hence it tries to solve the path constraints to cover the branches shown in Lines 27, 29, and 31 of Figure 4.4. All the tests that cover either of the branches fail because of the invocation of `assert(false)`. Hence, all the generated tests that fail on execution expose a behavioral or state difference. Similarly, JUnit Factory also tries to achieve maximum structural coverage and thus tries to generate inputs so that branches at Lines 8 and 10 of Figure 4.6 can be covered.

4.4 Experiments

This section presents our experiments conducted to address the following research question:

- Can the regression test suite generated by DiffGen effectively detect state infections that cannot be detected by previous state-of-the-art techniques [41]?

If the answer is yes, then DiffGen can be used to complement state-of-the-art techniques to improve regression-fault detection capability. Although a state infection does not guarantee to be manifested as a behavioral difference (or a failure due to regression fault), it is still an indication of a possible regression fault. Hence, an approach that is effective in infecting program state is more likely to find more regression faults.

4.4.1 Experimental Subjects

Table 5.1 lists eight Java classes that we use in the experiment. `UBStack` is the illustrating example taken from the experimental subjects used by Stotts et al. [100]. `IntStack` was used by Henkel and Diwan [52] in illustrating their approach of discovering algebraic specifications. `ShoppingCart` is an example for JUnit [25]. `BankAccount` is an example distributed with Jtest [77]. The remaining four classes are data structures previously used to evaluate Korat [17]. The first four columns show the class name, the number of methods, the number of public methods, and the number of non-comment, non-blank lines of code for each subject, respectively. The last column shows the coverage achieved by tests generated by JUnit Factory for the original version.

4.4.2 Experimental Setup

Although our ultimate research question is to investigate if DiffGen can detect regression faults not detected by previous state-of-the-art techniques, our subject classes were not equipped with such faults; therefore, we used MuJava [69], a Java mutation testing tool, to seed faults in these classes. MuJava modifies a single line of code in an original version in order to produce a faulty version. For each mutant and the original class version, we generate tests using JUnit Factory [61].

```
1    class BSTTestDriver implements set{
2       public static void driver(){
3          BST bst = new BST();
4          for(int i=0; i< 5; i++)
5             switch(cute.Cute.input.Integer()){
6                case 0:
7             MyInput key = cute.cute.Input.Object(MyInput);
8             bst.insert(key);
9          case 1:
10            key = cute.cute.Input.Object(MyInput);
11            bst.remove(key);
12         case 2:
13            key = cute.cute.Input.Object(MyInput);
14            bst.contains(key);
15         .....
16         default: break;
17            }
18         }
19      }
20   }
```

Figure 4.7: Test driver used to generate tests for the original and a mutated version of the BST class using the SeparateGen approach.

To evaluate the fault-detection capability of our DiffGen approach, we compare the fault-detection capability of the test suite generated by DiffGen with the fault-detection capability of test suites generated by JUnit Factory, separately for two versions of the class under test.

Experiments using jCUTE. Figure 4.7 shows the test driver that we use for generating tests separately for the original class and the mutant version for the BST example in Section 4.2. As shown in the figure, we make an object of only one version of BST, we later invoke the sequence of methods on this object in a similar way as in the driver synthesized by DiffGen (shown in Figure 4.4). The test driver for other subjects also has the same structure as the driver for BST. We use the same driver to generate tests for both the original and mutant version separately. Once the tests are generated, we augment the tests to add assertions for checking the fields (which we made public) and the return value of the modified method under test. We then execute these tests to find behavioral or state differences. This approach is similar to the one proposed by Evans and Savoia [41] (discussed in Sections 4.1 and 5.7). For simplicity, we refer to the preceding approach as SeparateGen in the rest of this chapter.

The second to the last column of Table 5.1 shows the code coverage achieved by tests generated by jCUTE on average for the original as well as a mutant version of the Java class under test, while the last column shows the code coverage achieved by tests generated by JUnit Factory. To generate tests for the original and a mutant versions used in SeparateGen, we set the upper limit on the number of paths to be explored by jCUTE to 2000.

We next measure the number of killed and unkilled mutants by SeparateGen. For all the mutants that were not killed by SeparateGen, we use DiffGen to generate regression tests, and record the num-

ber of killed mutants among the unkilled mutants by SeparateGen. We consider a mutant killed if a JML-postcondition-violating exception is thrown while executing these tests. Note that there are some mutants whose behaviors are the same as the behaviors of the original version. On manual inspection, we found out that in many mutants, the value of a local parameter is increased (or decreased) using a postfix increment (or decrement) operator, and the variable is never used after the mutated statement. We also count such mutants among the mutants unkilled by DiffGen.

Our DiffGen approach is more expensive than the SeparateGen approach for achieving a similar coverage of classes under test. To be fair about the time given to both approaches to find behavioral differences, we set the maximum number of paths to be explored by jCUTE (while generating regression tests using DiffGen) to be 200 (which is much less as compared to the number of paths explored by jCUTE while generating tests using SeparateGen). We also observed that the time taken to generate tests using our DiffGen approach (time for exploring 200 paths) was substantially less than the time taken to generate tests using SeparateGen (time for exploring 2000 paths). This observation shows that our experimental setup at least does not favor our DiffGen approach in terms of resource allocation.

Experiments using JUnit Factory. We generate tests separately for the two versions of class under test with JUnit Factory. We run the generated tests for the original version on a mutant version, and run the generated tests for the mutant version on the original version of the class under test to expose behavioral differences. This approach is the one used by Evans and Savoia [41]. We then use DiffGen to find behavioral differences between the original version of class under test and the mutants that were left unkilled by SeparateGen.

4.4.3 Measures

We measure the total number of mutants generated for each subject, the number of unkilled mutants by SeparateGen (denoted by `u`), the number of mutants killed by DiffGen among the ones not killed by SeparateGen (denoted by `k`). We also measure the number of mutants that had the same behavior as the original version (denoted by `s`) among the mutants that were not killed by DiffGen or SeparateGen. We next measure Improvement Factor (`IF1`) of DiffGen over SeparateGen. `IF1` = $\frac{k}{u}$. We measure another improvement factor (`IF2`) of DiffGen over SeparateGen by excluding the mutants with same behavior as the original version of class. `IF2` = $\frac{k}{u-s}$. The values of IF1 and IF2 indicate the extra fault-detection capability of DiffGen over SeparateGen.

4.4.4 Results

Table 4.2 shows the results from the experiment that we conducted. Column 1 shows the name of the subject. Column 2 shows the total number of mutants. Columns 3 shows the number of unkilled mutants by SeparateGen. Columns 4 shows the number of mutants killed by DiffGen, among the mutants that were not killed by SeparateGen. Column 5 shows the number of mutants with the same behavior as the

```
1   public boolean allDifferent(){
2       int n = size - 1;
3          for (int i = 0; i < n; i++){
4             for (int j = i + 1; j < n; j++) {//for (int j = i + 1; j < n--; j++){
5          if (elements[i] == elements[j])
6             return false;
7             }
8          }
9       return true;
10   }
```

Figure 4.8: A method from `DisjSet` class. Line 4 was mutated to the one shown in the comment.

original class version. Columns 6 and 7 show the Improvement Factors `IF1` and `IF2`, respectively of DiffGen. From Table 4.2, we observe that DiffGen has an Improvement Factor `IF2` varying from 40% to 100% for all the subjects with the exception of `DisjSet`, which has an improvement factor of 23.4%.

On manual inspection of the mutants generated for `IntStack`, we found out that most of the mutants unkilled by DiffGen had the fault in branches that were not covered by the generated tests. In particular, these branches were the ones for handling the behavior of `IntStack` when the stack size exceeded the `Initial Capacity` of `IntStack`. jCUTE could not generate tests that inserted as many elements in the stack such that the stack size became equal to the `Initial Capacity` of `IntStack`. Therefore, these branches could not be covered either by the tests generated for SeparateGen or by the tests generated by DiffGen. Hence these mutants were not killed by either approach.

On manual inspection of unkilled mutants for other subjects, we found that many of the unkilled mutants contained a fault injected inside a loop. Consider a representative of such types of mutants in Figure 4.8. Line 4 of the method `allDifferent` (in class `DisjSet`) was mutated to the one shown in the comment at the same line. Expression `n` was mutated to `n--` in the inner loop. We suspect the reason for not killing such a mutant may be the difficulty of symbolic execution in dealing with loop bounds.

We also infer that SeparateGen approach kills more mutants using JUnit Factory for test generation in comparison to SeparateGen approach using jCUTE for test generation. This is expected because of the high coverage achieved by the test suites generated by JUnit Factory.

In summary, the evaluation answered the question the we mentioned in the beginning of Section 4.4 . DiffGen can detect a substantial percentage of faults that were not detected by SeparateGen, which is the representative of previous state-of-the-art techniques in test generation. In particular, DiffGen was able to detect from around 23% to 100% of faults that SeparateGen could not detect.

4.4.5 Experiments on Larger Subjects

We conducted additional experiments on larger subjects to validate that our approach is useful for these subjects. These subjects and their faults are taken from Subject Infrastructure Repository (SIR) [35]. We conducted experiments on three available versions of JTopas [60] subject from SIR. Among the

Table 4.1: Experimental Subjects.

class	meths	public	ncnb	Cov
IntStack (IS)	5	5	44	100%
UBStack (UBS)	11	11	106	100%
ShoppingCart (SC)	9	8	70	100%
BankAccount (BA)	7	7	34	100%
BinSearchTree (BST)	13	8	246	100%
BinomialHeap (BH)	22	17	535	87%
DisjSet (DS)	10	7	166	100%
FibonacciHeap (FH)	24	14	468	98%

three versions, we chose the classes that had faults available at SIR. There were 13 such classes and 38 faults in total were available for them. We tested on versions of these classes prepared by seeding all the available faults in the SIR repository for these classes one by one. These were the same subjects used by Evans and Savoia [41].

We did not use jCUTE in our experiments for these subjects because jCUTE cannot deal with strings and many of the methods in these classes had strings as arguments; In addition, many of the fields in these classes were of interface types; hence, jCUTE was unable to instantiate an object for these classes. We used JUnit Factory to generate tests for the original and each faulty versions of the subjects. We used the approach used by Evans and Savoia [41] to check whether the faulty versions were detected using the approach. We used our approach to detect the seeded-faults that were not detected by the preceding approach. Our subject classes are shown in Table 4.3. Column 1 of the table shows the version of JTopas. Column 2 shows the name of the class. Column 3 shows the lines of code in the class. Column 4 shows the number of faults available in the repository for that class. Column 5 shows the faulty versions that were not detected by the approach used by Evans and Savoia. Finally the last column shows the number of faulty versions detected by our approach among the ones not detected by the approach used by Evans and Savoia.

The results show that our DiffGen approach detects 5 of the 7 faults that were not detected by the approach used by Evans and Savoia.

4.5 Discussion

In this section we discuss some of the limitations of current implementation of our tool and how they can be addressed.

Changes on methods or signatures. The current implementation of DiffGen cannot deal with refactorings or other maintenance activities that change the name or signature of a method (such as Rename-Method and Changed-Method-Signature Refactorings). The *Change Detector* component detects cor-

Table 4.2: Experimental Results

class	#M	#JC Unk Mutants	DG Killed	#Same Behavior	IF1 (%)	IF2 (%)	#JUF Unk	DG Killed	IF1 (%)	IF2 (%)
IntStack	85	44	5	21	11.4	21.7	21	0	0	-
UBStack	187	18	6	7	30	54.5	15	6	40	75
ShoppingCart	18	10	5	4	50	83.3	7	3	42.85	100%
BankAccount	35	17	8	6	47	72.7	6	0	0	-
BinSearchTree	125	18	11	4	61.1	78.6	13	4	30.77	44.44
BinomialHeap	281	63	25	19	39.7	56.8	39	8	20.51	40
DisjSet	385	87	22	33	25.28	40.7	97	15	15.46	23.44
FibonacciHeap	339	137	45	43	32.8	47.87	53	5	9.43	50

Table 4.3: Experimental Results

Ver	class	LOC	F	U	D
v1	ExtIOException	78	3	0	-
v1	AbstractTokenizer	1672	3	1	1
v1	Token	159	1	0	-
v1	Tokenizer	287	1	0	-
v1	ExtIndexOutOfBoundsException	67	2	0	-
v2	ExtIOException	89	2	0	-
v2	ThrowableMessageFormatter	137	2	0	-
v2	AbstractTokenizer	2966	4	2	2
v2	Token	447	4	0	-
v3	EnvironmentProvider	240	3	1	0
v3	PluginTokenizer	407	1	0	-
v3	StandardTokenizer	1992	8	2	2
v3	StandardTokenizerProperties	2736	4	1	0
Total	13 classes		38	7	5

responding methods in the two versions by comparing the method names and signatures. In particular, DiffGen considers two methods to be corresponding if their names and signatures match. A refactoring detection tool [33] can be used to find corresponding methods that were refactored in the new version of the given class. DiffGen can then detect the behavioral differences for the methods whose names were changed. However, for methods or constructors whose signatures were changed (methods with a modified, added, or deleted parameter), developers would need to write a conversion method to convert the input format of the methods or constructors in the original version to the inputs required by the new version.

Changes on fields. The current implementation of DiffGen compares objects by directly comparing the fields in the objects of the input classes. However, if a field is deleted, added, or modified in the new version of the input class, DiffGen cannot correctly compare the receiver object states. This situation

can be addressed by invoking various observer methods on objects under comparison and comparing the return values of these observer methods.

Multiple changes. Although in our experiment we detect behavioral changes between an original and a mutant version of a class with a single fault injected. Our current implementation can deal with multiple faults injected in the same method, since we work at the granularity of a method. To detect multiple faults spread in multiple methods, the synthesized test driver can be modified to invoke all the textually different methods in a switch statement (instead of invoking just one method) after the invocation of a sequence of methods . The return values of the invoked methods and the resulting receiver object states can then be compared to detect behavioral differences.

Guided exploration. The runtime performance of the current implementation of DiffGen can be improved by conducting a guided exploration of paths. Currently, DiffGen uses jCUTE [94] to explore various paths, in order to eventually cover the path leading to behavioral differences between the two given versions. However, many paths need not be explored, if these paths do not help in exposing behavioral differences between the two given versions (e.g., these paths do not involve any changed code location). The runtime performance of DiffGen can be improved by guiding the path exploration to avoid these paths. A static analysis can be done to find those branches or paths that do not lead to the changed locations. We can then prevent jCUTE from exploring these branches or paths by making jCUTE backtrack for these branches or paths.

Testing multiple classes. The current implementation of DiffGen tests two versions of one class at a time. There can be some behavioral differences missed by our approach. Consider that a method in some other class is invoked from the original and modified version of the class under test. There is a change in the invoked method, but there is no change in the two versions of the class under test. Our change detector would find no textual difference between the two versions of the class under test. Hence, no behavioral differences are found. Such situations can be addressed by either testing all the classes or extending the change detector to find textual differences between the transitive closure all the methods invoked from the class under test.

Other limitations. Since we use jCUTE [94] or JUnit Factory for test generation, the limitations in jCUTE or JUnit Factory are inherited by DiffGen. For example, jCUTE does not provide effective support for generating method sequences that produce desirable receiver-object states or non-primitive-argument states. Currently, in our test driver, we generate method sequences of lengths up to the number of methods inside the class under test. However, many bugs may be detected by sequences of only a larger length. Due to this limitation in jCUTE, we could not detect many behavioral differences in mutants of `IntStack`. To deal with this situation, we can use testing tools such as Evacon [54] that address this limitation.

4.6 Threats to Validity

The threats to external validity primarily include the degree to which the subject programs are representative of true practice. Our subjects are from various sources and the Korat data structures have nontrivial size for unit testing. These threats could be further reduced by experiments on more subjects and third-party tools. We currently use third part tools jCUTE [94], a state-of-the-art test generation tool that is based on concolic execution and JUnit Factory [61] a commercial test generation tool. The main threats to internal validity include faults in our tool implementation and faults in the third party tools that we use to generate tests. To reduce these threats, we have manually inspected the execution traces for several program subjects. These threats can be further reduced by conducting experiments using more test generation tools. The main threats to construct validity include the uses of those measurements in our experiment to assess our approach. To assess the effectiveness of our tool, we measure the percentage of faults detected by our tool but not detected by the SeperateGen approach. These faults were seeded by a mutation testing tool called `MuJava` [69] to approximate the real regression faults introduced as an effect of changes made in the maintenance process. Although empirical studies showed that faults seeded by mutation testing tools yield trustworthy results [7], these threats can be reduced by conducting more experiments on real regression faults or by conducting experiments using more mutation testing tools.

4.7 Related Work

The Orstra approach [120] automatically augments an automatically generated test suite with extra assertions for guarding against regression faults. Orstra first runs the given test suite and collects the return values and receiver-object states after the execution of the methods under test. Based on the collected information, Orstra synthesizes and inserts new assertions in the test suite for asserting against the collected method-return values and receiver object states. However, this approach observes the behavior of the original version to insert assertions in the test suite generated for only the original version. Therefore, the test suite might not include test inputs for which the behavior of a modified version differs from the original version.

Evans and Savoia [41] recently proposed an automated differential testing approach in which they generate test suites for the two given versions of a software system (say *V1* and *V2*) using JUnit Factory. Let the generated test suites for the two versions *V1* and *V2* be *T1* and *T2*, respectively. Their approach then ran test suite *T1* on *V2*, and test suite *T2* on *V1*. They found 20-30% more behavioral differences, as compared to the traditional regression testing approach, i.e., executing test suite *T1* on Version *V2*. In our experiments using JUnit Factory for test generation, we compared our approach to the one used by Evans and Savoia [41].

Sometimes the quality of the existing tests might not be sufficient enough to expose behavioral differences between two program versions by causing their outputs to be different. Then some previous regression test generation approaches generate new tests to expose behavioral differences. DeMillo and

Offutt [32] developed a constraint-based approach to generate unit tests that can exhibit program-state deviations caused by the execution of a slightly changed program line (in a mutant produced during mutation testing [31]) in Fortran programs. Winstead and Evan [119] proposed a preliminary approach to generate differential tests for C programs. They use genetic algorithms to find inputs for which the two method versions behave differently. They deal with only primitive argument types. Hence their approach cannot be used for any object-oriented language, since there are more complexities involved. For example, the receiver object state may change after an invocation of a method, but the return values of the two methods under test may be the same. Korel and Al-Yami [64] proposed an approach for differential test generation for Pascal programs. They use a search-based chaining approach to generate inputs for which the two methods under test take a different execution path. They require that there is only a slight change in the methods in the two versions. Although the preceding approaches for procedural programs can be applied on individual methods of the class under test, DiffGen is developed to conduct regression unit testing of object-oriented programs.

Apiwattanapong et al. [8, 91] use data and control dependence information along with state information gathered through symbolic execution, and provides guidelines for testers to augment an existing regression test suite. This approach does not generates any test case, but provides guidelines for testers to augment an existing test suite.

Some existing capture and replay techniques [37] capture the inputs and outputs of the unit under test during system-test execution. These techniques then replay the captured inputs for the unit as less expensive unit tests, and can also check the outputs of the unit against the captured outputs.

Ren et al. propose a change impact analysis tool called Chianti [86]. Chianti uses a test suite to produce an execution trace for two versions of a software system, and then categorizes those changes into different types. Since Chianti uses only the old test suite, it might not exercise behavioral differences between the two versions of the software system under test.

Dig et al. proposed a tool called *RefactoringCrawler* [33] that detects refactorings between two versions of a software system. *RefactoringCrawler* detects refactorings in two phases. It uses syntactic analysis to get a list of entity pairs that are similar textually in the first phase. In the second phase, it uses references among entities between the two versions to refine the results obtained from the first phase. Our previous work [104] extends *RefactoringCrawler* to detect refactorings in software libraries. *RefactoringCrawler* is complementary to DiffGen since *RefactoringCrawler* finds refactored entities in two versions of a software system and DiffGen finds methods with behavioral differences. The output of both tools helps the developers of API client code to adapt their systems to the evolved APIs.

Harman et al. proposed the idea of testability transformation [51]. Testability transformation aims to transforms a program to improve the performance of a test generation technique. The transformation might not preserve the semantics of the program under test. However, the generated tests can be executed on the original version of the program under test. Our technique of transforming the program by making public the fields transitively reachable by objects of the classes under test is an instance of testability

transformation.

4.8 Conclusion

A successful software application continues to evolve throughout its lifetimes undergoing various kinds of changes. These changes can introduce unintended side-effects called regression faults. It is desirable to find these regression faults as soon as possible to reduce the cost involved in fixing them. Regression testing is an approach of retesting a software application to find regression faults. Regression testing is conducted by executing an existing test suite on the modified software application to find behavioral or state differences. These behavioral or state differences can be inspected to find regression faults. Hence, the effectiveness of regression testing depends on the effectiveness of the existing test suite in terms of finding behavioral or state differences. We have developed an approach and its implementation called DiffGen. DiffGen takes as input two versions of an application and generates a regression test suite for the two given versions. The behavioral or state differences between the two versions are exposed on executing the test suite. Experimental results show that DiffGen can effectively detect state and behavioral differences that cannot be detected by the state-of-the-art techniques.

Chapter 5

Effecient Generation of Regression Tests

5.1 Introduction

Software applications (referred to as application in this Chapter) continue to evolve throughout their lifetime undergoing various kinds of changes. While making changes to a application, software developers may introduce regression faults in the application. It is highly desirable to detect these regression faults as quickly as possible to reduce the cost of developers in fixing the introduced faults. Continuous testing [89] tests an application as soon as developers make changes to the application and these changes are compilable. To detect regression faults quickly, existing continuous testing techniques [89] execute an existing test suite as soon as the changes are saved in an editor. The tests that fail on the modified application version (and pass on the original version) expose behavioral differences[1] between the two versions. Developers can inspect these behavioral differences to determine whether they are intended or unintended (i.e., regression faults). However, the effectiveness of existing continuous testing techniques depends on the capability of the existing test suite in detecting behavioral differences between the original and the new application versions.

The existing test suite might not be able to detect behavioral differences as it is usually created (or generated) without taking into consideration the changes to be made in the future. Then the existing test suite can be augmented using existing test generation techniques to improve the capability of the test suite in terms of detecting behavioral differences. Existing test generation techniques such as path-exploration-based test generation (PBTG) [21, 26, 45, 46, 63, 95, 115] and search-based test generation [55, 112] focus their efforts on increasing structural coverage and do not specifically focus on detecting behavioral differences between two versions of an application. As a result, these techniques are ineffective and inefficient for regression test generation, even with increasing computing power thanks to multi-core architectures and cloud computing.

[1] A behavioral difference between two versions of an application can be reflected by the difference between the observable outputs produced by the execution of the same test (referred to as a difference-exposing test) on the two versions.

To address the issue, we propose an approach called eXpress for efficient regression test generation with PBTG techniques. PBTG techniques (such as dynamic symbolic execution [110, 122] and concolic test generation [45, 95]) are gaining popularity due to their effectiveness in generating a test suite that achieves high structural coverage. To achieve high structural coverage, PBTG techniques try to explore all feasible paths in the application under test, and such exploration is typically quite expensive. However, if our aim is to detect behavioral differences between two versions of an application, we do not need to explore all these paths in the application since some of these paths are irrelevant paths, i.e., paths whose executions can never detect any behavioral differences. These irrelevant paths need not be explored to make regression test generation efficient. eXpress prunes out these irrelevant paths from the exploration space of a PBTG technique. In general, our path pruning can be used to optimize both path-oriented (such as PBTG) as well as goal-oriented test generation techniques (such as eToc [112]) for regression test generation. Goal-oriented techniques generate tests to execute a goal (such as a branch). For regression test generation, the goal can be the changes made to an application. Our approach can optimize goal-oriented techniques by pruning irrelevant branches (i.e., branches whose executions can never detect any behavioral differences) from the list of branches that may lead to the changes (and propagate the change effects).

eXpress includes a novel practical application of a theoretical fault model: the Propagation, Infection, and Execution (PIE) model [116] of error propagation. According to the PIE model, a fault can be detected by a test if a faulty statement is executed (E), the execution of the faulty statement infects the state (I), and the infected state (i.e., error) propagates to an observable output (P). A change in the new version of an application can be treated as a fault and then the PIE model is applicable for effect propagation of the change. Our key insight is that execution of many paths in an application guarantees not to satisfy any of the conditions E, I, or P of the PIE model. These paths can be pruned out from the exploration space of a PBST technique, directing its efforts towards regression test generation. In particular, eXpress first determines a set of paths ($P_{\neg E}$) that cannot lead to any changed code region and a set of paths ($P_{\neg P}$) through which a state infection cannot propagate to any observable output. eXpress then prunes paths $P_{\neg E}$ and $P_{\neg P}$ (from the exploration space of a PBST technique). In addition, eXpress prunes other irrelevant paths ($P_{\neg I}$) that are determined during exploration with a PBST technique (see Section 5.4); these paths do not cause state infection immediately after the execution of any changed code region. Our technique for pruning paths $P_{\neg E}$ can be used in general to achieve new code coverage (or violate assertions) by treating not-covered locations (or assertions) as changed code regions, while the pruning of paths $P_{\neg P}$ and $P_{\neg I}$ is specific for regression test generation.

There are two technical challenges that our approach addresses. First, to find paths $P_{\neg E}$ and $P_{\neg P}$ (before path exploration is started), one needs to build an inter-procedural control-flow graph (CFG), and often the construction of an inter-procedural CFG is not scalable for real-world applications. To address the preceding challenge, we build a minimal inter-procedural CFG for which our purpose of finding sets $P_{\neg E}$ and $P_{\neg P}$ can still be served. Second, to determine $P_{\neg I}$ (during path exploration), we need

to determine whether the application state is infected by a generated test. A PBST technique could be modified to simultaneously explore both the application versions to determine whether the application state is infected by a generated test, but realizing such simultaneous exploration can be challenging. To address the preceding challenge, eXpress explores only the new application version and executes a generated test (that executes a changed code region) on the original application version to determine whether the application state is infected by the generated test.

We have implemented eXpress as a search strategy for Dynamic Symbolic Execution (DSE) [45,95], a state-of-the-art PBTG technique. In particular, our implementation guides DSE to avoid from flipping branching nodes[2], whose unexplored side is guaranteed to lead to an irrelavant path[3]). In addition, eXpress can exploit the existing test suite (if available) for the original version by seeding the tests in the test suite to further optimize exploration. Our seeding technique efficiently augments an existing test suite so that various changed code regions of the application (that are not covered by the existing test suite) are covered by the augmented test suite. As a result, behavioral differences are likely to be found earlier in path exploration.

eXpress makes the following major contributions:

Path Exploration for Efficient Regression Test Generation. We propose an approach called eXpress for efficient generation of regression tests. To optimize the search strategy of a PBTG technique, eXpress prunes paths whose execution guarantees not to satisfy condition E, I, or P. As a result, behavioral differences are found efficiently by the PBTG technique with eXpress than without eXpress.

Incremental Exploration. We develop a technique for exploiting an existing test suite, so that path exploration focuses on covering the changes rather than starting from the scratch. As a result, behavioral differences are found more efficiently by the PBTG technique with eXpress based on an existing test suite than starting from the scratch.

Implementation. We have implemented our eXpress approach in a tool as an extension for Pex [110], an automated structural testing tool for .NET developed at Microsoft Research.

Evaluation. We have conducted experiments on 67 versions (in total) of four applications with two from the Subject Infrastructure Repository (SIR) [35] and two from real-world open source projects. Experimental results show that Pex using eXpress requires about 36% less amount of time (on average) to detect behavioral differences than without using eXpress. In addition, Pex using eXpress detects four behavioral difference that could not be detected without using eXpress (within a time bound). Furthermore, Pex requires 67% less amount of time to find behavioral differences by exploiting an existing test suite than exploration without using the test suite.

[2]A branching node in the execution tree of an application is an instance of a conditional statement in the source code. A branching node consists of two sides (or more than two sides for a `switch` statement): the true branch and the false branch. Flipping a branching node is flipping the execution of the application from the true (or false) branch to the false (or true) branch. Flipping a branching node for a switch statement is flipping the execution of the current branch to another unexplored branch.

[3]An irrelevant path here is a path whose execution guarantees not to satisfy any of the E, I, and P of the PIE model.

5.2 Dynamic Symbolic Execution

We implement our approach for Pex [110], an engine for Dynamic Symbolic Execution (DSE), a state-of-the-art PBST technique. Pex starts path exploration with some default inputs. Pex then collects constraints on application inputs from the predicates at the conditional statements executed in the application. We refer to these constraints at conditional statements as branch conditions. The conjunction of all branch conditions in the path followed during execution of an input is referred to as a path condition. Pex (and other PBST techniques) keeps track of the previous explored paths to build an execution tree. Each node in the tree is an instance of some conditional statement in the source code, each edge in the tree is an instance of some branch in the application source code (or its CFG), and different paths in the tree are (already explored) execution paths. The nodes of the tree are referred to as branching nodes. Pex, in the subsequent run[4], chooses one of the branching nodes in the execution tree (explored thus far), such that not all outgoing branches of the node have been explored yet. Pex flips the chosen branching node to generate a new input whose execution follows a new path. Intuitively, flipping a branching node in an old path is to construct a new path that shares the prefix (in the execution tree) to the node with an old path (containing the flipped node), but then deviates and takes a different branch of the node. Pex uses various heuristics [122] for choosing a branching node (to flip next) applying various search strategies with an objective of achieving high code coverage fast. Hence, the path exploration in PBST techniques (including DSE) is realized through flipping branching nodes.

We next present definitions of some terms that we use in the rest of this chapter.

Instance of a Conditional Statement. Multiple instances of a conditional statement in the source code (of an application) can be present in the execution tree. The branching nodes in the execution tree corresponding to a conditional statement s in the source code are referred to as instances of s. Note that statements other than conditional statements are abstracted away from the execution tree.

Branch. A branch $<s_i, s_j>$ in the source code (or its CFG) is an edge connecting conditional statement s_i to statement s_j in the CFG. A conditional statement typically has two branches (or more than two branches for a `switch` statement): the true branch and the false branch.

Branch Instance. Let $<s_i, s_j>$ be a branch in the source code (or its CFG) from statement s_i to statement s_j such that s_i is a conditional statement. Let s_k be the first conditional statement that is encountered (in the CFG) after taking branch $<s_i, s_j>$. Note that if s_j is a conditional statement, $s_k = s_j$. A branch $br_{ik} = <b_i, b_k>$ in the execution tree[5] of a program is an instance of $<s_i, s_j>$ iff branching node b_i is an instance of conditional statement s_i and branching node b_k is an instance of conditional statement s_k.

Unexplored Branch Instance. An unexplored branch instance of a branching node b_i in the execution tree of a program is a branch instance $br_{ik} = <b_i, b_k>$ of b_i such that the branch instance br_{ik} is not taken

[4] A run is an exploration iteration.

[5] A branch in the execution tree is an edge in the execution tree. Every edge in the execution tree is an instance of a branch in the source code. In the rest of this chapter, we refer to a branch in the execution tree as a branch instance.

yet.

Discovered Node. A discovered branching node (in short as a discovered node) is a branching node that is explored in the current DSE run but whose corresponding conditional statement in the source code was not executed in previous runs.

Path. A path P in the execution tree of an application is a list of branching nodes $P = <b_1, b_2, b_3, .., b_n>$ in the execution tree such that the branching nodes $b_1, b_2, b_3, ...b_n$ are executed in order.

Unexplored Branch Instances along a Path. Let B be the set of all branches of all the branching nodes in path $P = <b_1, b_2, b_3, .., b_n>$. Unexplored branch instances along path P are all those branch instances $B_u \in B$ such that $\forall br_{ik} = <b_i, b_k> \in B_u$, br_{ik} is unexplored.

Path Prefix. Two paths P_1 and P_2 have a common prefix up to a branching node b_i iff the two lists P_1 and P_2 have a maximum prefix ending with branching node b_i.

Branch Pruning. Let b_i be a branching node (an instance of an `if` or a `while`/`for` statement) such that at least one of the branch instances $br_{ik} = <b_i, b_k>$ of b_i is unexplored. Pruning of branch instance br_{ik} is the removal of b_i from the exploration space of a PBST technique. As a result, b_i is prevented from being flipped by the PBST technique. The effect is pruning all such paths that share the prefix up to branching node b_i, and take an unexplored branch instance of branching node b_i.

Branch Pruning along a Path. Pruning a branch $<s_i, s_j>$ (in the source code) along path $P = <b_1, b_2, b_3, .., b_n>$ is the pruning of all the instances of branch $<s_i, s_j>$ from P.

Changed Code Region. A changed code region (in a new application version) is a minimal set of statements S that contains all modified (in the new or original application version), added (in the new application version), or deleted (in the original application version) statements in a method such that the nodes corresponding to S in the CFG of the new application version form a single-entry-single-exit subgraph[6].

5.3 Example

In this section, we illustrate our eXpress approach with an example. eXpress takes as input two versions of an application and produces as output a regression test suite, with the objective of detecting behavioral differences (if any exist) between the two versions of the application. Although eXpress analyzes assembly code of C# applications, in this section, we illustrate the eXpress approach using application source code.

To make a PBST technique efficient, eXpress prunes paths from the exploration space of the PBST technique so that the PBST technique focuses its efforts on regression test generation. To prune various paths from the exploration space of the PBST technique, eXpress determines certain branches (referred to as irrelevant branches) before the path exploration by the PBST technique. eXpress then uses these

[6]The requirement of single-entry-single-exit subgraph ensures that instrumented code (see Section 5.4.3), inserted just after a changed code region δ, post-dominates δ.

```
   static public int TestMe(char[] c){
1    int state = 0;
2    if(c == null || c.Length == 0)
3     return -1;
4    for(int i=0; i< c.Length; i++){
5      if(c[i] == "[") state =1;
6      else if(state == 1 && c[i] == "{") state =2;
7      else if(state == 2 &&  c[i] == "<") state =3;
8      else if(state == 3 && c[i] == "*"){
9       state =4;
10      if(c.Length==15)//Added in new version
11        state = state + 1;//Added in new version
       }
12     if(c[i]==' ')
13      return state;
14     if(!(c[i] >= 'a' && c[i] <= 'z')){
15      state=-1; return state;
       }
      }
    }
16  if(c.Length >=15 && c[15] == '}')
17   return state;
18  return -1;
  }
```

Figure 5.1: An example application.

irrelevant branches to prune irrelevant paths (during path exploration) from the exploration space of the PBST technique. To explain the path pruning by eXpress, we use DSE as a representative PBST technique.

Consider an example application `TestMe` in Figure 5.1. Lines 10 and 11 of the application are added in a new version. We next walk through our approach using the example.

5.3.1 Detection of Irrelevant Branches.

Figure 5.2 shows the CFG of the application in Figure 5.1. The labels of vertices in the CFG denote the corresponding line numbers in Figure 5.1. The black vertices denote the newly added statements at Lines 10 and 11. The gray vertices denote the conditional nodes (for the conditional statements in the application), while the white vertices denote the other statements in the application. From the CFG, eXpress first determines two categories of branches B_E and B_P. eXpress then uses B_E and B_P to prune irrelevant paths during path exploration.

- **Category B_E.** On statically traversing the CFG in Figure 5.2, eXpress detects that taking the branches $<2,3>$, $<4,16>$, $<16,17>$, $<16,18>$, $<12,13>$, and $<14,15>$ (dotted edges in Figure 5.2), the application execution cannot reach any of the black vertices. Hence, the execution of these branches guarantees not to execute the changed statements.

- **Category B_P.** In addition, eXpress statically detects that among the source vertices of the six branches

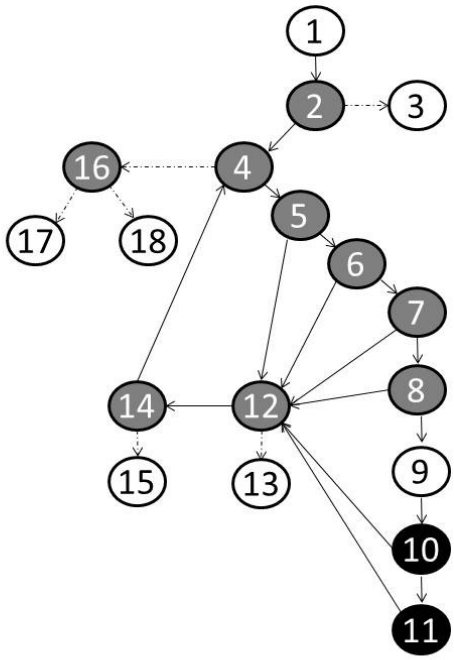


Figure 5.2: The CFG for the application in Figure 5.1.

in Category B_E, there is no path from any of any black vertices to vertex 3. Hence, a state infection after the execution of any black vertex cannot propagate through branch $< 2, 3 >$.

5.3.2 Path Pruning

To cover the changed statements at Lines 10 and 11 (in Figure 5.1) and detect behavioral differences, DSE needs at least 6 DSE runs (starting from an empty input array c). However, the number of runs depends on the choice of the branching node that DSE flips in each run. In each run, DSE has the choice of flipping 8 new branching nodes (apart from the branching nodes that accumulate in previous runs) in the application. For `TestMe`, Pex takes 441 DSE runs to cover the true branch of the statement at Line 10. The number of runs can be much more if the number of branching statements in the application increases. Since path exploration is realized in DSE through flipping of branching nodes, eXpress dynamically prunes certain instances of branches in the dynamic execution tree. As an effect, eXpress prunes paths containing the instances of the branches.

- **Category $P_{\neg P}$.** eXpress prunes all instances of branches B_P from the exploration space of DSE since executing these branches guarantees not to execute a changed code region or propagate a state infection to an observable output. As an effect, all the paths containing instance of any node in B_P are pruned from the exploration space.

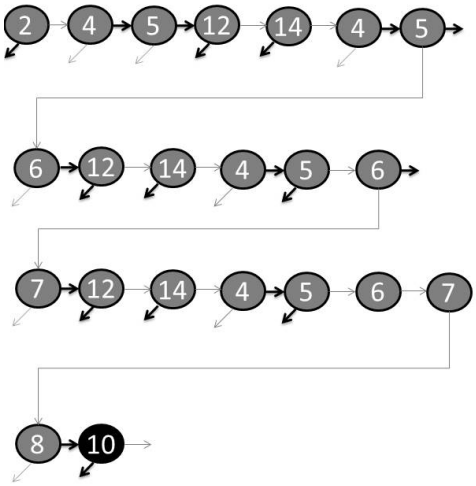


Figure 5.3: A part of the execution tree of the application for test I = {"[", "{", "<", "*" }.

- **Category $P_{\neg E}$.** If along an already explored path no black vertex is executed, all the instances of branches in $B_E - B_P$ along the path are pruned. Note that category $P_{\neg P}$ already includes the paths containing B_P. Hence, to make the three categories exclusive, we prune $B_E - B_P$ instead of B_E. For example, if a test $I = \{"["\}$ is generated that follows a path $P_I = <..., 5, 12, 14, 4, 16, 18>$ and does not execute any black vertex, the unexplored branch instances $<12, 13>$, $<14, 15>$, and $<16, 17>$ (along the path P_I) are pruned since exploring these branch instances guarantees not to execute the black vertex along the path prefix shared with P_I.
- **Category $P_{\neg I}$.** If along an already explored path, some black vertex is executed but the program state is not infected after the execution of the black vertex, all the instances of branches in B_E after the execution of the last black vertex in the path are pruned. For example, if a test $I = \{"[","{","¡","*"\}$ is generated to follow a path $P_I = <..., 10, 12, 14, 4, 16, 18>$ is explored, the unexplored branch instances $<12, 13>$, $<14, 15>$, and $<16, 17>$ (along P_I) are pruned since the program state is not infected and executing these branch instances guarantees not to execute any black vertices along the path prefix shared with P_I.

5.3.3 Incremental Exploration

eXpress can reuse an existing test suite for the original version so that changed code regions of the program can be explored efficiently due to which test generation is likely to find behavioral differences earlier in path exploration. Assume that there is an existing test suite covering all the statements in the original version of the program in Figure 5.1. Suppose that the test suite has a test I = {"[", "{", "<", "*"

}. The input covers all the statements in the new version of `TestMe` except the newly added statement at Line 11. If we start the path exploration from scratch (i.e., with default inputs), Pex takes 441 runs to cover the statement at Line 11. However, we can reuse the existing test suite for exploration to cover the new statement efficiently. Our approach executes the test suite to build an execution tree for the tests in the test suite. Our approach then starts path exploration using the dynamic execution tree built by executing the existing test suite instead of starting from an empty tree. Some branching nodes in the tree may take many runs for Pex to discover if starting from an empty tree. Figure 5.3 shows a part of the execution tree for the input I. A gray edge in the tree indicates the false branch instance of a branching node while a black edge indicates the true branch instance. To generate an input for the next DSE runs, Pex flips a branching node b whose other side has not yet been explored and generates an input so that program execution takes the unexplored branch instance of b. Pex chooses such branching node for flipping using various heuristics for covering changed code regions of the program. It is likely that Pex chooses branching node 10 (colored black), which on execution covers the added statement at Line 11. Using our approach of seeding the tests from the existing test suite, Pex takes 39 runs (in contrast to 441) to flip the branching node and cover the statement at Line 11.

5.4 Approach

eXpress takes as input the assembly code of two versions $v1$ (original) and $v2$ (new) of the application under test. In addition, eXpress takes as input the name and signature of a parameterized unit test[7](PUT). When an existing test suite is available for the original version, eXpress conducts incremental exploration that exploits the test suite for generating tests for the new version. We next discuss in detail the eXpress approach.

5.4.1 Code-Difference Identification

eXpress analyzes the two versions $v1$ and $v2$ to find pairs $<M_{i1}, M_{i2}>$ of corresponding methods in $v1$ and $v2$, where M_{i1} is a method in $v1$ and M_{i2} is a method in $v2$. A method M is defined as a triple $<FQN, Sig, I>$, where FQN is the fully qualified name[8] of the method, Sig is the signature[9] of the method, and I is the list of assembly instructions in the method body. Two methods $<M_{i1}, M_{i2}>$ form a corresponding pair if the two methods M_{i1} and M_{i2} have the same FQN and Sig. For each pair $<M_{i1}, M_{i2}>$ of corresponding methods, eXpress finds a set of differences Δ_{i1} (for the original version) and Δ_{i2} (for the new version) between the list of instructions $I_{M_{i1}}$ and $I_{M_{i2}}$ in the body of Methods M_{i1} and M_{i2}, respectively. Δ_{i1} (Δ_{i2}) includes all those instructions such that each instruction ι in Δ_{i1} (Δ_{i2}) is

[7] A parameterized unit test [110] is a test method with parameters. Such a method serves as a driver for path exploration.

[8] The fully qualified name of a method m is the combination of the method's name, the name of the class c declaring m, and the name of the namespace containing c.

[9] The signature of a method m is the combination of parameter types of m and the return type of m.

an instruction in $I_{M_{i1}}$ ($I_{M_{i2}}$ for Δ_{i2}), and ι is added (deleted for Δ_{i2}) or modified from list $I_{M_{i1}}$ to form $I_{M_{i2}}$. We denote the set of all added, modified, or deleted instructions in v_1 as Δ_1 and in v_2 as Δ_2. Currently our approach considers as different methods the methods that have undergone Rename Method or Change Method Signature refactoring. A refactoring detection tool [33] can be used to find such corresponding methods.

5.4.2 Graph Building

eXpress efficiently constructs the inter-procedural CFG $g <V,E>$ of the application under test such that each vertex $v \in V$ corresponds to an instruction $\iota \in M$ (denoted as $v \leftrightarrow \iota$), where M is some method in v_2. eXpress starts the construction of the inter-procedural CFG from the Parametrized Unit Test (PUT) τ provided as input. The inter-procedural CFG is used by eXpress to find branches (in the graph) via which the execution cannot reach any vertex containing a changed instruction in the graph.

Since a moderate-size application can contain a large number of method invocations (including those in its dependent libraries), often the construction of its inter-procedural CFG is not scalable to real-world situations. Hence, we build a minimal inter-procedural CFG for which our purpose of finding branches whose execution cannot later reach some changed code region in the application can be served. The pseudo code for building the inter-procedural CFG is shown in Algorithm 9. Initially, the algorithm `InterProceduralCFG` is invoked with the argument as the PUT τ. The algorithm first constructs an intra-procedural CFG g for method τ. For each method invocation vertex[10] (invoking method c) in g, the algorithm `InterProceduralCFG` is invoked recursively with the invoked method c as the argument (Line 25 of Algorithm 9), after adding c to the call stack (Line 24). After the control returns from the recursive call, the method c is removed from the call stack (Line 26) and added to the set of visited methods (Line 27). The inter-procedural graph `cg` (with c as an entry method) resulting from the recursive call at Line 25 is merged with the graph g (Line 28). The algorithm `InterProceduralCFG` is not invoked recursively with c as the argument in the following situations:

c **is in call stack.** If c is already in the call stack, `InterProceduralCFG` is not recursively invoked with c as the argument (Lines 9-10). This technique ensures that our approach is not stuck in a loop in method invocations. For example, if method A invokes method B, and method B invokes method A, then the construction of the inter-procedural graph stops after method A is encountered the second time.

c **is already visited.** If c is already visited, `InterProceduralCFG` is not recursively invoked with c as the argument (Lines 16-17). This technique ensures that we do not build the same subgraph again.

c **is in `CanReachChangedRegion`.** The set `CanReachChangedRegion` is populated whenever a changed method[11] is encountered. In particular, if a changed method is encountered, the methods currently in the call stack are added to the set `CanReachChangedRegion` (Lines 19-23). If c is in

[10] A method invocation vertex is a vertex representing a call instruction.

[11] A changed method M_i is a method for which the set $\Delta_i \neq \emptyset$.

Algorithm 9 Pseudo code of Construction of Inter-Procedural Control Flow Graph

Input: A test method τ.

Output: The inter-procedural Control Flow Graph (CFG) of the application under test.

```
InterProceduralCFG(τ)
 1: g ← GenerateIntraProceduralCFG(τ)
 2: MethodCallStack ← ∅, CanReachChangedRegion ← ∅
 3: ChangedMethods ← FindChangedMethods()
 4: AChangedMethod ← m ∈ ChangedMethods
 5: acg ← GenerateIntraProceduralCFG(AChangedMethod)
 6: for all Vertex v ∈ g.Vertices do
 7:     if v.Instruction = MethodInvocation then
 8:         c ← getMethod(v.Instruction)
 9:         if c ∈ MethodCallStack then
10:             goto Line 2 //To handle loops or recursions
11:         end if
12:         if c ∈ CanReachChangedRegion then
13:             g ← GraphUnion(acg, g, v)
14:             goto Line 2
15:         end if
16:         if c ∈ Visited then
17:             goto Line 2
18:         end if
19:         if c ∈ ChangedMethods then
20:             for all Method m ∈ MethodCallStack do
21:                 CanReachChangedRegion.Add(m)
22:             end for
23:         end if
24:         MethodCallStack.Add(c)
25:         cg ← InterProceduralCFG(c)
26:         MethodCallStack.Remove(c)
27:         Visited.Add(c)
28:         g ← GraphUnion(cg, g, v)
29:     end if
30: end for
31: return g
```

`CanReachChangedRegion`, `InterProceduralCFG` is not recursively invoked with c as the argument, while merging CFG of some changed method with g (Lines 12-15).

Note that if method `m` can invoke (directly or indirectly) a changed method `cm`, not all the branches in this method `m` may be able to reach[12] the changed region in `cm` (e.g., due to *return* statements). Branches

[12] A branch can reach a changed region if the edge (in the CFG) corresponding to the branch can reach the nodes corresponding to the changed region.

whose execution cannot reach any changed code region (i.e., irrelevant branches) are found by eXpress. If a node b can (or cannot) reach a changed code region in Inter-Procedural CFG g built without using the preceding optimization, the node b can (or cannot) reach a changed code region (maybe a different one) in the graph built using the preceding optimizations. Since our aim of building the inter-procedural CFG is to find irrelevant branches, i.e., those in the graph via which the execution cannot reach any changed code region, the preceding three optimizations help achieve the aim while reducing the cost of building the inter-procedural CFG. In addition, the size of the inter-procedural CFG is reduced resulting in reduction in the cost of finding irrelevant branches.

5.4.3 Code Instrumentation

The code instrumentation helps eXpress in determining whether the application state is infected by a generated test. For each changed method pair $<M_{i1}, M_{i2}>$ (i.e., $\Delta_{i1} \neq \emptyset$ or $\Delta_{i2} \neq \emptyset$) , eXpress finds changed code regions δ_{i1} and δ_{i2} (for the original and new application versions, respectively) containing all the changed instructions in the application. At the end of each changed code region δ_{i1} and δ_{i2}, eXpress inserts instructions to save the application state. In particular, eXpress finds the set of variables v_{di} that can potentially be defined in $\delta_i 1$ (and $\delta_i 2$). eXpress then inserts instructions to capture the value of each variable in v_{di} as an assertion (if the variable is of primitive type). If the variable is of a non-primitive type, eXpress captures the object state of the variable and linearizes the state to a string. These observed values (or object states) are stored and inserted in assertions in a generated test to compare with the observed application state. A PBST technique is used to generate tests for the new version v_2. During path exploration, whenever a test is generated (for v_2) by a PBST technique to execute a changed code region, eXpress executes the generated test on the original application version (v_1) to determine whether application state is infected immediately after the execution of the changed code region. If any of the captured values (or object states) is different across the two versions, an assertion fails for indicating application state infection. The instrumentation enables to perform only one instance of path exploration on the new version instead of performing two instances of path exploration: one on the original and the other on the new application version. Performing two instances of DSE can be technically challenging since the two DSE instances need to be performed in a controlled manner such that both versions are executed with the same input and the execution trace is monitored for both the versions by a common exploration strategy to decide which branching node to flip next in the two versions.

5.4.4 Irrelevant-Branch Identification

There can be an infinite number of paths in the CFG of the application and many of them are infeasible. Hence, it is often not feasible to enumerate all irrelevant paths in the application thus far. Moreover, path exploration is realized through flipping branching nodes by PBST techniques. Hence, eXpress first finds branches whose corresponding branching nodes need not be flipped (under certain situations), and uses

these branches for path pruning. In particular, eXpress traverses g to find a set of branches B_E (in the CFG) via which the execution cannot reach any of the instructions in Δ_2, and a set of branches B_P via which no state infection can propagate to any observable output. eXpress then uses these branches (and the already explored paths) to prune various paths (during path exploration) that need not be explored to find behavioral differences. A branch b in CFG g is an edge $e =<v_i, v_j>: e \in E$, $v_i \in V$ with an outgoing degree of more that one. We next describe the sets B_E and B_P.

Let $V = \{v_1, v_2, .., v_l\}$ be the set of all vertices in CFG $g<V,E>$ such that $v_i \in V$ and $v_i.degree > 1$. Let $E_i = \{e_{i1}, e_{i2}, ..., e_{im}\}$ be the set of outgoing edges (branches) from v_i. Let C be the set of vertices in the CFG g such that $\forall v \in C, \exists \mathfrak{u} \in \Delta_1 \bigcup \Delta_2 : v \leftrightarrow \mathfrak{u}$. $\rho(v_i, v_j, e_{ij})$ denotes the set of paths from a source vertex v_i to a destination vertex v_j such that these paths take the branch e_{ij} (if $\rho(v_i, v_j, e_{ij}) = \emptyset$, there is no such path from v_i to v_j), and $\rho(v_i, v_j)$ denotes the set of paths from a source vertex v_i to a destination vertex v_j (if $\rho(v_i, v_j) = \emptyset$, there is no such path from v_i to v_j).

Branches B_E. $B_E \subseteq E$ is a set of branches such that $\forall e_{ij} =< v_i, v_j > \in B_E \wedge \forall c_k \in C : \rho(v_i, c_k, e_{ij}) = \emptyset$. Since $\rho(v_i, c_k, e_{ij}) = \emptyset$, after taking a branch $e_{ij} \in B_E$, the application execution cannot reach a changed code region. Hence, the application state cannot be infected.

Branches B_P. $B_P \subseteq E$ is the set of branches such that $\forall e_{ij} =< v_i, v_j > \in B_P \wedge \forall c_k \in C : \rho(v_i, c_k, e_{ij}) = \emptyset \wedge \rho(c_k, v_i) = \emptyset$. Since $\rho(c_k, v_i) = \emptyset$, a state infection after a changed vertex c_k cannot reach v_i. Hence, the state infection cannot propagate through e_{ij}. Note that $B_P \subseteq B_E$.

5.4.5 Path Pruning

eXpress prunes various paths for a PBST technique to make path exploration efficient for regression test generation. During path exploration, eXpress uses the set of branches B_E and B_P to determine paths that can be pruned from the exploration space of a PBST technique. These paths are guaranteed not to be able to detect behavioral differences between v_1 and v_2. We next describe the three categories of paths that eXpress prunes from the exploration space of a PBTG technique:

Category $P_{\neg P}$. Along all the paths already explored by a PBST technique, eXpress prunes all the instances of branches $b \in B_p$. As an effect, $P_{\neg P}$ contains all paths (in the application) that include an instance of some branch $b \in B_P$. Since all branches in B_P cannot reach a changed code region and no changed code region has a path to any branch in B_P, each path in $P_{\neg P}$ guarantees not to execute any of the changed code regions or propagate a state infection to an observable output. Hence, along these paths, no behavioral differences could be found. Note that the branches in $B_P - B_E$ may be able to propagate a state infection along some path in which a changed code region is executed. Hence, it is not safe to prune all paths including these branches.

Category $P_{\neg E}$. Let P_{ne} be the set of all paths (in the explored execution tree) that do not execute any changed code region, eXpress prunes all the instances of branches in $B_E - B_P$. Since along the paths P_{ne}, no changed code region is executed, the application state cannot be infected along these paths. Hence,

it is safe to prune branches $B_E - B_P$ along these paths since they cannot lead to a changed code region. As an effect, eXpress prunes all the paths that have a common prefix with some path in P_{ne} up to a branching node b_1 such that $b = <b1, b2>$ is an instance of branch br and $br \in B_E - B_P$, and b is not explored yet.

Category $P_{\neg I}$. Let P_{ni} be the set of all paths (in the explored execution tree) that execute some changed code region. However, the application state is not infected after the execution of any changed code region. Along each path in P_{ne}, eXpress prunes the instances of branches B_e that are explored after the execution of the last changed code region. Since the state is not infected along any path in P_{ne} and the branches in B_E cannot reach any changed code region (again), it is safe to prune these branches.

5.4.6 Incremental Exploration

A regression test suite achieving high code coverage may be available along with the original version of an application. However, the existing test suite might not be able to cover all the changed code regions of the new version of the application. Our approach can reuse the existing test suite so that changed code regions of the application can be executed efficiently due to which test generation is likely to find behavioral differences earlier in path exploration. Our approach executes the existing test suite to build an execution tree for the tests in the test suite. Our approach then starts the path exploration using the execution tree instead of starting from an empty tree. Our approach of seeding tests can help efficiently cover the changed code regions of the application with two major reasons:

Discovery of hard-to-discover branching nodes. By seeding the existing test suite for DSE to start exploration with, our approach executes the test suite to build an execution tree of the application. Some of the branching nodes in the built execution tree may take a large number of DSE runs (without seeding any tests) to get discovered. Flipping some of these discovered branching nodes whose corresponding branches are closer in CFG to the changed parts of the application has more likelihood of covering the changed code regions of the application [20]. Although our approach currently does not specifically flip first branching nodes whose corresponding branches are near the changed code regions, our approach can help these branching nodes to get discovered (by executing the existing test suite), which might take a large number of DSE runs as shown in the example in Section 5.3.

Priority of DSE to cover not-covered regions of the application. DSE techniques typically prioritize branching nodes for flipping so that high coverage can be achieved faster. Thus, DSE techniques choose a branching node from the execution tree (built thus far) such that flipping it has a high likelihood of covering changed code regions (that are not covered by the existing test suite for the original version). By seeding the existing test suite to path exploration, the DSE techniques do not waste time on covering the regions of the application already covered by the existing test suite. Instead, the DSE techniques give high priority to branching nodes that can cover the application's not-covered regions, which include the changed code regions. Hence, it is likely to cover the changed code regions earlier in path exploration.

5.5 Experiments

We conduct experiments on four applications and their 67 versions (in total) collected from three different sources. In our experiments, we address the following research questions:

RQ1. How high percentage of paths explored by Pex belong to the three categories of irrelevant paths ($P_{\neg P}$, $P_{\neg E}$, and $P_{\neg I}$ as described in Section 5.4) being pruned by eXpress?

RQ2. How many fewer DSE runs and how much less amount of time does Pex using eXpress require to find behavioral differences than Pex without using eXpress?

RQ3. How many fewer DSE runs and how much less amount of time does Pex require to find behavioral differences when the path exploration is seeded with an existing test suite?

5.5.1 Subjects

To address the research questions, we conducted experiments on four subjects. Table 5.1 shows the details about the subjects. Column 1 shows the subject name. Column 2 shows the number of classes in the subject. Column 3 shows the number of classes that are covered by tests generated in our experiments. Column 4 shows the number of versions (not including the original version) used in our experiments. Column 5 shows the number of lines of code in the subject.

`replace` and `siena` are applications available from the Subject Infrastructure Repository (SIR) [35]. `replace` and `siena` are written in `C` and `Java`, respectively. `replace` is a text-processing application, while `siena` is an Internet-scale event notification application. We choose these two subjects (among the others available at SIR) in our experiments as we could convert these subjects into C# using the Java 2 CSharp Translator[13]. We could not convert other subjects available at the SIR because of extensive use of *C* or *Java* library APIs in these subjects. We seed all the 32 faults available for `replace` at SIR one by one separately on the original version to synthesize 32 new versions of `replace`. For `siena`, SIR contains 8 different sequentially released versions of `siena` (versions 1.8 through 1.15). Each version provides enhanced functionalities or corrections with respect to the preceding version. We use these 8 versions in our experiments. In addition to these 8 versions, there are 9 seeded faults available at SIR. We seeded all the 9 faults available at SIR one by one separately on the original version to synthesize 9 new versions of `siena`. In total, we conduct experiments on these 17 versions of `siena`. For `replace`, we use the `main` method as a PUT for generating tests. We capture the concrete value of the string `sub` at the end of the `PUT` using the `PexStore.ValueForValidation("v", v)` statement. This statement captures the current value of *v* in a particular run (i.e., an explored path) of DSE. In particular, this statement results in an assertion `Assert.AreEqual(v, cv)` in a generated test, where *cv* is the concrete value of *v* in the test during the time of exploration. This assertion is used to find behavioral differences when the tests generated for a new version are executed on the original version.

[13] `http://sourceforge.net/projects/j2cstranslator/`

For `siena`, we use the methods `encode` (for changes that are transitively reachable from `encode`) and `decode` (for changes that are transitively reachable from `decode`) in the class `SENP` as PUTs for generating tests. We capture the return values of these methods using the `PexStore` statement in the PUTs.

The method `encode` requires non-primitive arguments. Pex currently cannot handle non-primitive argument types effectively but provides support for using factory methods for non-primitive types. Hence, we manually write factory methods for the non-primitive types in `SENP`. In particular, we write factory methods for classes `SENPPacket`, `Event`, and `Filter`. Each factory method invokes a sequence (of length up to three) of the public state-modifying methods in the corresponding class. The parameters for these methods, and the length of the sequence (up to three) are passed as inputs to the factory methods. During exploration, Pex generates concrete values for these inputs to cover various parts of the application under test.

STPG[14] is an open source application hosted by the codeplex website, which contains snapshots of check-ins in the code repositories for STPG. We collect three different versions of the subject STPG from the three most recent check-ins. We use the `Convert(string path)` method as the PUT for generating tests since `Convert` is the main conversion method that converts a string path data definition to a `PathGeometry` object. We capture the return value of `Convert` using the `PexStore` statement in the PUT.

`structorian`[15] is an open source tool for binary-data viewing and reverse engineering. `structorian` is hosted by Google's open source project hosting website. The website also contains snapshots of check-ins in the code repositories for `structorian`. We collect all the versions of snapshots for the classes `StructLexer` and `StructParser`. We chose these classes in our experiments due to three factors. First, these classes have several revisions available in the repository. Second, these classes are of non-trivial size and complexity. Third, these classes have corresponding tests available in the repository. For classes `StructLexer` and `StructParser` , we generalized some of the available concrete test methods by promoting primitive types to arguments of the test methods. Furthermore, we convert the assertions in the concrete test methods to `PexStore` statements. For example, if an assertion `Assert.IsEqual(v, 0)` exists in a concrete test, we convert the assertion to `PexStore.ValueForValidation("v", v)`. We use these generalized test methods as PUTs for our experiments. `structorian` contains a manually written test suite. We use this test suite for seeding the exploration for addressing RQ3.

To address questions RQ1-RQ2, we use all the four subjects, while to address question RQ3, we use `structorian` because of two major factors. First, `structorian` has a manually written test suite that can be used to seed the exploration. Second, revisions of `structorian` contain non-trivial changes that cannot be covered by the existing test suite. Hence, our technique of seeding the existing test suite in the

[14] `http://stringtopathgeometry.codeplex.com/`
[15] `http://code.google.com/p/structorian/`

Table 5.1: Experimental subjects

Project	Classes	Classes Covered	Versions	LOC
replace	1	1	32	625
STPG	1	1	2	684
siena	6	6	17	1529
structorian	70	8	16	6561

path exploration is useful for covering these changes. `replace` contains changes to one statement due to which most of the changes can be covered by the existing test suite. Similarly, the changes in `siena` are covered by the existing test suite. Hence, our incremental exploration technique is not beneficial for the version pairs of `replace` or `siena` under test. `STPG` does not have an existing test suite to use.

5.5.2 Experimental Setup

For `replace` and `siena`, we conduct regression test generation between the original version and each version *v*2 synthesized from the available faults and released versions (if any) in the SIR. We use eXpress and the default search strategy in Pex [110, 122] to conduct regression test generation. In addition to the versions synthesized by seeding faults, we also conduct regression test generation between each successive versions of `siena` (versions 1.8 through 1.15) available in SIR, using eXpress and the default search strategy in Pex [110, 122]. For `STPG` and `structorian`, we conduct regression test generation between each pair of two successive versions that we collect.

To address RQ1, we categorize all the irrelevant paths explored by Pex (without using eXpress) as one of the three categories described in Section 5.4 and measure the percentage of paths in each category. To address RQ2, we compare the number of runs and the amount of time required by Pex with the number of runs required by Pex using eXpress (referred to as Pex+eXpress in the rest of this chapter) to find behavioral differences between two versions of an application under test. To address RQ3, we compare the number of DSE runs and the amount of time required by Pex (and Pex+eXpress) to identify behavioral differences with and without seeding the path exploration (with the existing test suite).

Currently, we have not automated our code-instrumentation technique. We instrument each version manually for our experiments. In future work, we plan to automate the technique. The rest of the approach is fully automated and is implemented in a tool as an extension[16] to Pex [110]. We have developed its components to statically find irrelevant branches as a .NET Reflector[17] AddIn.

To find behavioral differences between two versions, we execute on the original version the tests generated for a new version. Behavioral differences are detected by a test if an assertion in the test fails.

[16] `http://pexase.codeplex.com/`

[17] `http://www.red-gate.com/products/reflector/`

5.5.3 Experimental Results

In this section, we present the experimental results to address the Research Questions RQ1, RQ2, and RQ3.

RQ1: Path Categorization

We next address RQ1 regarding the categorization of different irrelevant paths (described in Section 5.4) explored by Pex. Table 5.2 shows the categorization of paths explored by Pex. Column *Subject* shows the subject name. Column $\#B_P$ shows the average number of branches in the set B_P. Column $\#B_E$ shows the average number of branches in the set B_E. Column #T shows the average number of branches in the CFG. Column P_1 shows the percentage of irrelevant paths (on average) in Category $P_{\neg P}$ among all the paths explored by Pex. Column P_2 shows the percentage of irrelevant paths (on average) in Category $P_{\neg E}$ among all the paths explored by Pex. Column P_3 shows the percentage of irrelevant paths (on average) in Category $P_{\neg I}$ among all the paths explored by Pex. Column $T(Irr)$ shows the percentage of irrelevant paths (on average) among all the paths explored by Pex. Note that the branch set $B_P \subseteq B_E$, while the path sets $P_{\neg P}$, $P_{\neg E}$, and $P_{\neg I}$ are disjoint.

The number of branches in B_P is substantially less than the number of branches in B_E. We observe that the number of branches in B_P is higher when a change is deeper inside the CFG, i.e., at a larger distance from the start node of the CFG. For example, the versions 5, 6, 8, 15, 16, and 24-27 of `replace` have a higher number of branches in B_P. As a result, the number of paths in Category $P_{\neg P}$ (P_1) is substantially higher (than the average) in these versions. In contrast, there are hardly any branches in B_P for the versions of STPG. Hence, there is no path in $P_{\neg P}$ (P_1) for these versions. The number of paths in $P_{\neg E}$ (P_2) is higher (on average) than the number of paths in $P_{\neg P}$ (P_1) as there are more branches in B_E. The number of paths in $P_{\neg I}$ (P_3) is smaller than in $P_{\neg P}$ (P_1) and $P_{\neg E}$ (P_2) as not many generated tests execute a changed code region such that the application state is not infected. We also observe that the number of irrelevant paths increases with the increase in the number of irrelevant branches. In total, 46% of the total paths explored by Pex are irrelevant paths. This percentage indicates the benefit that our path pruning techniques can potentially achieve in optimizing a PBST technique for regression test generation.

RQ2: Path Pruning

Table 5.3 shows the experimental results of applying our path pruning techniques. In this dissertation, we provide only the total, average, and median metric values of the versions for which behavioral differences were found by both Pex and Pex+eXpress. The detailed results for experiments on all the versions of these subjects are available on our project web[18].

[18] `https://sites.google.com/site/asergrp/projects/express/`

Table 5.2: Categorization of the paths explored by Pex.

Subject	$\#B_P$	$\#B_E$	# T	P_1	P_2	P_3	$T(Irr)$
replace	4	84	206	23%	25%	8%	55%
siena	9	29	157	6%	18%	5%	29%
STPG	2	13	145	0%	12%	1%	13%
structorian (SL)	1	13	69	0%	11%	13%	24%
structorian (SP)	21	49	447	13%	28%	0%	41%

Table 5.3: Results of path pruning.

S	V	P_{Pex}	P_{Red}	M_p	T_{pPex}	$Ts+ T_d$	Tp_{Red}
replace	32	9812	63%	37%	711	305	57%
siena	17	6914	33%	11%	1011	718	29%
STPG	2	378	23%	23%	353	286	19%
SL	6	4326	37%	26%	144	98	31%
SP	10	49889	68%	77%	5hr	3.25hr	35%

Column S shows the name of the subject. The class `StructLexer` is denoted by SL and the class `StructParser` is denoted by SP. Column V shows the number of version pairs. Column P_{Pex} shows the total number of DSE runs required by Pex for satisfying P for all version pairs. Column P_{Red} shows the average percentage reduction in the number of DSE runs by Pex+eXpress for satisfying P (i.e., finding behavioral differences). Column M_p shows the median percentage reduction in the number of DSE runs by Pex+eXpress for satisfying P. Column T_{pPex} shows the time (in seconds) taken by Pex for satisfying P. Column $T_s + T_d$ shows the time (in seconds) taken by Pex+eXpress for satisfying P. This time includes the time taken to statically identify irrelevant branches. Column T_{PRed} shows the average percentage reduction in amount of time taken by Pex+eXpress for satisfying P.

Results of replace. For the `replace` subject, among the 32 pairs of versions, the changed code regions cannot be executed for 4 of theses version pairs (version pairs 0-14, 0-18, 0-27, and 0-31, where 0 is the original version) by Pex or by Pex+eXpress in 1000 DSE runs. We do not include these version pairs while calculating the sum of DSE runs for satisfying I and E of the PIE model. For 3 of the version pairs (version pairs 0-12, 0-13, and 0-21), the changes are in the fields due to which there are no benefits of using Pex+eXpress. We exclude these 3 version pairs from the experimental results shown in Table 5.3. For 3 of the version pairs (version pairs 0-3, 0-22, and 0-32), a changed code region was executed but the application state is not infected (by Pex or Pex+eXpress) in the time bound of 5 minutes. In addition, for 3 of the version pairs, the state infection is not propagated to an observable output within the bound of 1000 DSE runs. We do not include these version pairs while calculating the sum of DSE runs for finding behavioral differences. We observe that, for `replace`, Pex+eXpress takes 63% fewer runs

(median 37%) and 57% less amount of time in finding behavioral differences.

Results of siena. We observe that the behavioral differences between 7 of the version pairs of `siena` are found within 20 runs by Pex and Pex+eXpress. For these version pairs, there is no reduction in the number of runs. The reason for the preceding phenomenon is that changes in these version pairs are close to the start node in the CFG. Hence, these changes can be covered within a relatively small number of runs. In 2 of the version pairs, changed code regions are not covered by either Pex+eXpress or Pex. An exception is thrown by the application before these changes could be executed. Pex and Pex+eXpress are unable to generate a test to avoid the exception. Changes between 2 of the version pairs are refactorings due to which the application state is never infected. We observe that, for `siena`, Pex+eXpress finds behavioral differences in 33% fewer runs (median 11%) and 29% less amount of time than Pex.

Results of STPG. We observe that for the 2 version pairs of STPG, Pex+eXpress finds behavioral differences in 23% fewer runs (median 23%) and 19% less amount of time than Pex.

Results of structorian. For two versions of `StructLexer`, neither Pex nor Pex+eXpress is able to find behavioral differences. For the others, Pex+eXpress takes 37% fewer runs (median 26%) and 31% less amount of time to find behavioral differences. Neither Pex+eXpress nor Pex is able to find behavioral differences between all version pairs of class `StructParser` within 5 minutes (a bound that we use in our experiments for all subjects). For these version pairs, we increase the bound to 1 hour (or 10000 runs). Pex is not able to find behavioral differences for 8 version pairs even within 1 hour, while Pex+eXpress finds behavioral differences for 4 of these 8 version pairs. If Pex is unable to detect behavioral differences, for a version pair, within the bound of 1 hour, we use 1 hour (for the version pair) to calculate the total in column T_{Ppex}. In addition, we use the number of runs as 10000 (the bound on the number of runs) to calculate the total in column P_{pex}. Changes between two version pairs (40-45 and 40-47) could not be covered by either Pex or Pex+eXpress. One of the changes (between version pairs 47-50) is a refactoring. For this version pair, application state is infected but no behavioral differences are detected by either Pex or Pex+eXpress. In summary, for structorian, Pex+eXpress is able to detect behavioral differences for 4 of the version pairs that could not be detected by Pex. On average, Pex is able to find behavioral differences in 68% fewer runs (median 77%) and 35% less amount of time. The reduction in the number of runs is substantially larger than reduction in the amount of time due to non-trivial time taken by eXpress in identifying irrelevant branches.

Overall for all the subjects, Pex is able to find behavioral differences in 62% fewer runs and 36% less amount of time.

RQ3: Incremental Exploration

Table 5.4 shows the results of using the existing test suite to seed the path exploration. Column *C* shows the class name. Column *V* shows the pair of version numbers. The next four columns show the number of runs and time taken by the four techniques: Pex, Pex with seeding, Pex+eXpress, and Pex+eXpress with

seeding, respectively, for finding behavioral differences. Note that DSE runs required by our incremental exploration also include the seeded test runs. In Table 5.4, if none of the changed blocks is covered, we take the number of runs as 10,000 (the maximum number of runs that we run our experiments with). For 9 of the 16 version pairs of `structorian` that we used in our experiments, the existing test suite of `structorian` could not find behavioral differences. Therefore, we consider these 9 version pairs for our experiments for RQ3. Pex could not find behavioral differences for 5 of the 9 version pairs in 10,000 runs. Seeding the path exploration with the existing test suite helps Pex in finding behavioral differences for 3 of 5 version pairs under test. Pex+eXpress could not find behavioral differences for 3 of the 9 version pairs in 10,000 runs. Seeding the path exploration with the existing test suite helps Pex+eXpress in finding behavioral differences for 2 of these 3 version pairs under test.

In summary, Pex requires around 68% of the original runs and 67% less time (than time required by Pex without test seeding) and Pex+eXpress requires around 74% of the original runs and 70% less time (than time required by Pex+eXpress without test seeding). In terms of time, Pex with seeding marginally wins over Pex+eXpress with seeding due to time taken by Pex+eXpress in identifying irrelevant branches.

Summary

In summary, this section addresses research questions RQ1, RQ2, and RQ3.
RQ1. Among the total paths explored by Pex, 46% (on average) are irrelevant. 14%, 26%, and 6% of all the paths explored by Pex, respectively belong to $P_{\neg P}$, $P_{\neg E}$, and $P_{\neg I}$.
RQ2. Pex+eXpress requires 36% less amount of time (on average) to detect behavioral differences than without using eXpress.
RQ3. Pex with test seeding requires 67% less amount of time to find behavioral differences than Pex without test seeding.

5.6 Discussion

In this section, we discuss some of issues of the current implementation of our approach and how they can be addressed.

5.6.1 Added/Deleted and Refactored Methods

If a method M (or a field F) is added or deleted from the original application version, eXpress does not identify M (or F) as a changed code region. The change is identified if a method call site (or reference to F) is added or deleted from the original application version. If the added or deleted method (or field) is never invoked (or accessed), the behavior of the two versions is the same unless M is an overriding method. We plan to incorporate support for handling such overriding methods that are added

Table 5.4: Results of seeding the existing test suite.

C	V	N_{Pex}/T	Np_{sd}/T	N_{eXp}/T	Ne_{sd}/T
SP	2-5	**10000/60***	**10000/60***	2381/35	181/17
SP	37-39	3699/26	60/1	851/22	47/11
SP	39-40	**10000/60***	304/2	**10000/60***	251/12
SP	45-47	**10000/60***	**10000/60***	**10000/60***	**10000/60***
SP	47-50	**10000/60***	81/1	**10000/60***	64/10
SP	62-124	**10000/60***	59/1	7228/58	41/10
SL	169-174	478/1	324/1	34/1	18/1
SL	150-169	299/1	37/1	52/1	29/1
SL	9-139	2988/2	69/1	1002/1	52/1
Tot		64476/330	20934/128	41568/309	10683/123

*If behavioral differences are not detected, we take the number of runs as 10,000 (the maximum number of runs that we run our experiments with).

or deleted. Similarly, if a method *M* is refactored between the two versions, eXpress does not identify *M* as a changed code region. However, when a method is refactored, its call sites are changed accordingly (unless the method undergoes `Pull Up` or `Push Down` refactoring). Hence, eXpress identifies the method containing call sites of *M* as changed. In our experiments, we consider versions of `replace` in which a method signature is changed, and versions of `structorian` in which a method is renamed.

5.6.2 Granularity of Changed Code Region

In our current implementation, a changed code region is the list of continuous instructions that include all the changed instructions in a method. One method can have only a single changed code region. Hence, a changed code region can be as big as a method and as small as a single instruction. The granularity of a changed code region can be increased to a single method or reduced to single instruction. Changing the granularity to single method *M* can affect the efficiency of our approach in reducing DSE runs since some of the branches (in *M*) that should be considered irrelevant would not be considered irrelevant. In contrast, reducing the granularity to a single instruction makes our approach more efficient in reducing DSE runs. However, the overhead cost of our approach is increased due to state checking at multiple points in the application. In future work, we plan to enhance eXpress to allow users to choose from different levels of granularity.

5.6.3 Original/New application Version

In our current implementation, we perform DSE on the new version of an application. We then execute a test (generated after each run) on the original version. We can also perform DSE on the original

version instead of the new version. One approach may be efficient than the other depending on the types of changes made to the application. In future work, we plan to conduct experiments to compare the efficiency of the two approaches with respect to the types of changes.

5.6.4 Prioritization of Branching Nodes.

eXpress currently prunes branches whose execution guarantees not to satisfy the E or I condition in the PIE model. However, some remaining branches in the application code can be more promising in achieving these conditions than others. Branching nodes can be prioritized based on the distance of a branching node to a changed code region in the CFG. The distance $d(n1, n2)$ between any two nodes $n1$ and $n2$ in a CFG g is the number of nodes with $degree > 1$ between $n1$ and $n2$ along the shortest path between $n1$ and $n2$. Hence, the distance between a node b and a changed code region Δ is the number of nodes with degree ¿ 1 between b and the node δ representing the first instruction in Δ. The intuition behind this prioritization is that the shorter the distance between a CFG node and δ, the easier it is to generate inputs to cause the execution of the changed code region δ. This kind of branch prioritization is used by Burnim and Sen [20] for achieving high structural coverage. We can also prioritize branching nodes based on the probability to cause infection and to propagate the infection to an observable output. Moreover, we can prioritize branching nodes based on data dependence from a changed code region.

5.6.5 Pruning of Branches for Propagation

In future work, we plan to prune more categories of branches whose execution guarantees not to satisfy Propagation (P). Consider that a changed code region is executed and the application state is infected after the execution of the changed code region; however, the infection is not propagated to any observable output. Let χ be the last location in the execution path such that the application state is infected before the execution of χ but not infected after its execution. χ can be determined by comparing the value spectra [121] obtained by executing the test on both versions of the application. All the branching nodes after the execution of χ can be removed from the exploration space of a PBST technique.

5.6.6 Changes in Fields.

Currently, eXpress does not detect changes (in application code) that is outside method bodies. For example, if the declaration of a field f is modified, eXpress cannot help in reducing DSE runs to detect behavioral differences that may be introduced in the application due to the change. In such situations, the source code can be searched to find the references of f. The corresponding instructions for all these statements referring to f can be considered as changed. If a field is added or deleted, eXpress can still be helpful in reducing DSE runs as in the case of added or deleted methods as discussed earlier in this section.

5.6.7 Factors Affecting Test Seeding

The effectiveness of our incremental exploration technique is dependent on the characteristics of the existing test suite. In future work, we plan to conduct more extensive experiments with test suites of different characteristics, as done by Xu et al. [123, 124].

5.6.8 Incremental Call Graph Analysis

In our current implementation, the static analysis to construct the inter-procedural CFG and identify irrelevant branches is done from the scratch. However, the static analysis can be applied incrementally [98] to amortize the cost of static analysis across versions to further reduce the cost of incremental exploration.

5.6.9 Pruning for Other Test Generation Techniques

Our approach currently prune paths from the exploration space of a PBTG test generation technique. However, our approach can be applied to other test generation approaches that target branch coverage. For example, our pruning techniques can help search-based test generation techniques [112] in effective regression test generation. In particular, the set of branches B_E can be pruned from the target set of branches so that the search-based techniques focus their efforts on regression test generation. Once the changes are covered, branches B_P can be pruned from the target set of branches.

5.7 Related Work

Previous approaches [41, 57, 105] generate regression unit tests achieving high structural coverage on both versions of the class under test. However, these approaches explore all the irrelevant paths, whose execution guarantees not to satisfy any of the conditions E, I, or P in the PIE model [116]. In contrast, we have developed a new search strategy for DSE to avoid exploring these irrelevant paths.

Some existing search strategies [20, 122] guide DSE to efficiently achieve high structural coverage in an application under test. However, these techniques do not specifically target covering a changed code region. In contrast, our approach guides DSE to avoid exploring paths whose execution guarantees not to satisfy any of the conditions E, I, or P of the PIE model.

Santelices et al. [91] use data and control dependence information along with state information gathered through symbolic execution, and provide guidelines for testers to augment an existing regression test suite. Unlike our approach, their approach does not automatically generate tests but provides guidelines for testers to augment an existing test suite. Differential symbolic execution [78] determines behavioral differences between two versions of a method (or an application) by comparing their symbolic summaries [44]. Summaries can be computed only for methods amenable to symbolic execution.

However, summaries cannot be computed for methods whose behavior is defined in external libraries not amenable to symbolic execution. Our approach still works in practice when these external library methods are present since our approach does not require summaries. Qi et al. [82] propose an approach for guided test generation for evolving applications. The approach guides path exploration towards executing a change and propagating state infection to an observable output. However, their approach cannot deal with multiple interacting changes in the application in contrast to our approach. In addition, our approach can prune some paths (belonging to $P_{\neg E}$ and $P_{\neg I}$) that are explored by their approach.

Orstra approach [120] automatically augments an automatically generated test suite with extra assertions for guarding against regression faults. Orstra first runs the given test suite and collects the return values and receiver-object states after the execution of the methods under test. Based on the collected information, Orstra synthesizes and inserts new assertions in the test suite for asserting against the collected method-return values and receiver object states. However, this approach observes the behavior of the original version to insert assertions in the test suite generated for only the original version. Therefore, the test suite might not include tests for which the behavior of a new version differs from the original version.

Li [67] prioritizes source code portions for testing based on dominator analysis. In particular, her approach finds a minimal set of blocks in the application source code, which, if executed, would ensure the execution of all of the blocks in the application. Horwitz [53] prioritizes portions of source code for testing based on control and flow dependencies. These two approaches focus on testing in general. In contrast, our approach focuses specifically on regression testing.

Ren et al. develop a change impact analysis tool called Chianti [86]. Chianti uses a test suite to produce an execution trace for two versions of an application, and then categorizes and decomposes the changes between two versions of an application into different atomic types. Chianti uses only an existing test suite and does not generate new tests for regression testing. In contrast, our approach focuses on regression test generation.

Some existing capture and replay techniques [38, 75, 88] capture the inputs and outputs of the unit under test during system-test execution. These techniques then replay the captured inputs for the unit as less expensive unit tests, and can also check the outputs of the unit against the captured outputs. However, the existing system tests do not necessarily exercise the changed behavior of the application under test. In contrast, our approach generates new tests for regression testing.

Joshi et al. [59] use the path constraints of the paths followed by the tests in an existing test suite to generate inputs that violate the assertions in the test suite. The generated test inputs follow the same paths already covered by the existing test suite and do not explore any new paths. In contrast, our approach exploits the existing test suite to explore new paths. Majumdar and Sen [71] propose the concept of hybrid concolic testing. Hybrid concolic testing seeds the application with random inputs so that the application exploration does not get stuck at a particular application location. In contrast, our approach exploits the existing test suite to seed the application exploration. Since the existing test suite

is expected to achieve a higher structural coverage, the existing test suite is expected to discover more hard-to-discover branching nodes in comparison with random inputs.

Law and Rothermel [66] propose an impact analysis technique, called PathImpact. PathImpact uses method execution traces to find out impacted methods when a method is modified. Our technique of building an inter-procedural graph by pruning certain method call chains that are found to be reachable to a changed region is similar to the technique. However, our technique works on a static interprocedural graph due to which our technique is safe in contrast to PathImpact. In addition, our technique further reduces the size of Inter-procedural graph by not adding the already visited methods from the inter-procedural graph.

Rothermel and Harrold [87] introduce the notion of dangerous edges and use these dangerous edges for regression test selection. Our irrelevant branches for execution and infection (set B_{E+I}) of a changed region is the inverse of these dangerous edges. However, our approach also finds irrelevant branches that cannot help in propagating a state infection to observable output.

Godefroid et al. [46] propose a DSE based approach for fuzz testing of large applications. Their approach uses a single seed for application exploration. In contrast, our approach seeds multiple tests to application exploration. Seeding multiple tests can help application exploration in covering the changes more efficiently as discussed in Section 5.4.6. Xu and Rothermel [125] propose a directed test generation technique that uses the existing test suite to cover parts of the application not covered by the existing test suite. In particular, the approach first collects the set of branches B not covered by the existing test suite. To cover a branch $<s_i, s_j> \in B$, the approach selects all the tests T that cover statement s_i. For each test $t_i \in T$, the approach collects the path condition p_i of the path followed by t_i until the first instance of s_i, negates the predicate at the first instance of s_i from p_i to get path condition p'_i. The approach then generates a test that covers the branch $<s_i, s_j>$ by solving the path condition p'_i . However, if all the preceding path conditions p'_i derived from the paths followed by the tests T are not solvable, the approach cannot generate a test to cover the branch $<s_i, s_j>$, which can furthermore compromise the coverage of additional branches. In contrast, our incremental exploration technique can still generate a test to cover such branches. In addition, Xu et al. [123, 124] propose a search-based test augmentation technique that seeds the existing test suite for test generation. All these techniques focus on satisfying condition E of the PIE model. In contrast, our approach helps in satisfying E, I, and P of the PIE model.

DiffGen approach (presented in Chapter 4) instruments an application to add branches such that behavioral differences can be found effectively. However, a test generation tool needs to explore branches in both the original and new versions of the application to detect behavioral differences. In contrast, eXpress prunes irrelevant branches to find behavioral differences efficiently. Both approaches are complementary and can be combined for effective and efficient regression test generation.

Jin et al. [57] propose an approach, called BERT, for behavioral regression testing. BERT finds all the classes in an application that are modified and generates tests for these classes that capture the behavior of these classes. BERT then executes these tests on the original version to find behavioral

differences. In contrast to our approach, BERT does not focuses on efficiently generating tests to find behavioral differences. Hence, our approach is complementary to BERT.

5.8 Conclusion

Regression testing aims at generating tests that detect behavioral differences between two versions of an application. To expose behavioral differences, test execution needs to satisfy three conditions: Execution (E), Infection (I), and Propagation (P), as stated in the PIE model [116]. Path-exploration-base test generation (PBTG) techniques can be used to generate tests for satisfying these conditions. PBTG techniques explore paths in the application to achieve high structural coverage, and exploration of all these paths can often be expensive. However, the execution of many of these paths in the application guarantees not to satisfy any of the three conditions in any way. In this chapter, we have presented an approach and its implementation called eXpress for efficient regression test generation. eXpress prunes paths or branches whose execution guarantees not to satisfy the E, I, or P condition such that these conditions are more likely to be satisfied earlier in path exploration. In addition, our approach can exploit the existing test suite for the original version to efficiently execute the changed code regions (if not already covered by the test suite). Experimental results on various versions of applications have shown that our approach can efficiently find behavioral differences than without using our approach.

Chapter 6

Future Work

In this dissertation, we have have presented approaches for improving the quality assurance of DCAs. In Chapter 2, we have presented an approach, called PRIEST, for testing of DCAs in presence of privacy laws. In Chapter 3, we have presented an approach, called MODA, for bridging the gap between the FA and the DB so that a test generation tool can be applied to generate tests for a DCA. In Chapters 4, and 5, we have presented approaches for testing of changes in the FA of DCAs. Encouraged by our existing results, in future work, we plan to extend this research in the various as outlined in the rest of the chapter.

6.1 Data Anonymization for Other Purposes

In Chapter 2, we have presented an approach to anonymize the data in the DB to balance the need of privacy with the utility of testing DCAs. In future work, we plan to extend our research in two major directions.

6.1.1 Other Utilities

Current approaches for data anonymization are effective in protecting the sensitive records in a database from getting leaked out. However, the approaches can destroy the utility of the data. Hence, a specialized approach for data anonymization has to be developed to balance a given utility of the data and data privacy. In Chapter 2, we have presented an approach for data anonymization to balance the utility of testing with data privacy. In future work, we plan to devise anonymization techniques for balancing other utilities of the anonymized data (related to software engineering tasks) such as fault localization, debugging, and fault fixing.

6.1.2 Behavior Preserving Data Anonymization

With the emergence of cloud computing, more and more sensitive data is hosted in the cloud. However, the data stored in the cloud is at risk to attacks. As a result, privacy of people is at a risk. In 2012 alone, from January to June, there have been 189 data breaches and more than 13 million records have been exposed [97]. As a result, more and more focus has been on securing the data hosted on cloud. Data anonymization is one technique to anonymize data, so that an attacker is not able to get any private information from the data even if he or she gets access to the data. However, consider an application providing insurance quotes based on the data of its existing customers. The data in the DB cannot be anonymized for the application to work properly. In future work, we plan to develop anonymization techniques so that behavior of the DCAs that use the anonymized data is preserved.

6.2 Testing of Changes in DB Schema and Query Code

In this disseration, we develop approaches for testing changes in the FA of a DCA. However, we do not address the problem of testing changes in the DB schema. When changes are made to the schema of the database, the FA using the DB can often break. For example, if from the schema an attribute *age* in the DB is deleted and if the application is expecting the *age* attribute in a *ResultSet* object, the application will break. In future, we plan to develop efficient and effective approaches for testing the changes made to the schema.

6.3 Quality Assurance of Other Kinds of Databases

In this disseration, we have presented approaches for quality assurance of DCAs that use Relational Database Management Systems (RDBMSes) as their back-end databases. In future work, we plan to extend our research to other kinds of back-end databases.

In recent years, cloud computing has captured attention as a promising new computing platform for deploying service oriented DCAs. One important benefit of cloud computing is elastic scaling, where virtually unlimited throughput is achieved by adding servers if the workload increases, and operation cost is reduced by removing servers if the workload decreases. Traditionally, database-centric applications rely on RDBMSes to manage data and provide data consistency in the presence of concurrent client requests. RDBMSs guarantee strong data consistency by providing transactional support with ACID (Atomic, Consistent, Isolated, and Durable) property [11]. The ACID property is important in ensuring correctness of many DCAs.

On the other hand, supporting ACID transactions over a distributed system such as cloud computing environment often comes with significant performance overhead [109], and hinders scalability. Moreover, CAP theorem [43] states that consistency, availability and partition-tolerance cannot be achieved at

the same time. Thus, preserving consistency in the presence of network partition leads to unavailability. RDBMSes provide ACID property at the expense of performance and availability.

Recently many vendors offer non-relational database management systems called NoSQL systems [81], which provide higher performance, scalability, and availability in cloud computing environments by forgoing the ACID property. As a result, the performance benefits come at the cost of data inconsistency. In particular, NoSQL databases can result in data inconsistency between various replicas of the databases. Such inconsistencies can sometimes result in stale data reads or conflicting data in two different replicas of the databases.

Some kinds of conflicts can be tolerated, while others can result in failures. For example, if you are looking at the likes of a Facebook post, it is no big deal if the likes are read from a replica that contains stale values. However, if two people are buying the same book from a bookstore at the same time and there is only one book left in the database, the bookstore might commit to selling the book to both the people. Such kinds of scenarios are not desirable for the application vendors.

NoSQL databases provide various options of consistencies to developers, while making reads and writes to the databases. For example, the Oracle NoSQL Database allows developers to choose the level of read consistency from four possible levels. When developers are in charge of making choices, they can often make mistakes while developing a DCA, resulting in faulty applications, which can result in conflicts leading to failures. Before deploying a DCA, DCA vendors need to make sure that conflicts (that cannot be tolerated) do not occur in practice. However, to the best of our knowledge, there are no existing approaches to find inconsistencies in such applications before they are deployed. We plan to develop approaches for automated consistency checking of NoSQL-based DCAs in future work.

6.4 Testing of Front-End Applications

In Chapters 5, we have presented an approach for efficient testing of changes made to the FA of a DCA. In particular, eXpress prunes various branches that cannot help in satisfying either of the conditions E, I, or P of the PIE model. In future work, we plan to extend the work on two major directions. First, we plan to develop techniques for effectively achieving condition P of the PIE model after the application state is infected. Second, we plan to develop techniques for analyzing the failing tests (exposing behavioral differences), grouping the failing tests as representing intended or unintended behaviors, and presenting a concise test report to the developers.

6.4.1 Effective Infection Propagation

The eXpress approach presented in this dissertation currently prunes out irrelevant branches that cannot help in any of the three conditions in the PIE model. In addition, the approach inserts branches for effective infection of application state. However, `eXpress` does not have any technique for effectively

propagating a state infection to an observable output.

In our evaluation (presented in Section 5.5), we observe that there are four versions of `replace` and one version of `siena` from the SIR for which Pex takes non-trivial time to find behavioral differences after the application state is infected. In addition, there are three such versions of the `StructLexer` in structorian. These applications consist of various paths that lead to an exception. For most of these eight versions, the state infection is not propagated to an observable output since both the application versions result in the exception. The state is infected just before the exception is thrown.

To effectively propagate a state infection, a test generation strategy needs to (1) figure out the application point at which state infection stops propagation and (2) prioritize flipping of certain branches that are likely to push the state infection further. To figure out the stopping point of state infection, we plan to develop a technique (called shadow execution) for executing the two application versions, simultaneously. Shadow execution will instrument the application in such a way that both versions of the application are executed simultaneously, enabling us to compare application states of the two versions at various application points to locate the statement S at which infection propagation stops. The approach will prioritize the branches nearby S (in the control-flow graph) to further push the state infection towards an observable output. Moreover, the approach will prune out all the branches that are after the stopping point.

6.4.2 Inference of Unintended Behaviors

Currently, developers need to manually inspect the tests that detect behavioral differences (failing tests) to decide whether they represent an intended behavior or an unintended behavior (i.e., a regression fault). Depending on the changes made to the application, there can be a large number of failing tests. Furthermore, most of these failing tests may represent intended behavioral differences. As the number of failing tests grow, manual inspection of the failing tests to ensure that the tests represent intended behavioral difference becomes prohibitively time consuming. Furthermore, manual inspection is often prone to errors. To reduce the efforts of developers in deciding whether a generated test represents an unintended behavior, we plan to develop an approach for automatically classifying the tests as representing intended or unintended behavior. To classify the failing tests, the approach will (1) cluster the failing tests based on distances between the paths followed by the tests, and (2) learn from version histories of the software application.

To infer whether a generated test t represents an intended or unintended behavior, the approach will cluster and analyze generated (failing) tests that follow paths that are at a distance $d <= \varepsilon$ from the path followed by t, where distance is a metric of similarity of paths followed by two different tests and ε is a threshold value. In addition, the approach will analyze the changed regions executed by each test and the application state at the changed regions to cluster the tests. The approach will interact with the developers to help label one representative test from each cluster as representing intended or unintended

behavior. The approach will use version history in addition to the labels to rank the failing tests.

6.4.3 Generation of Concise Test Reports

In addition to ranking the tests (exposing behavioral differences) based on their likeliness of representing unintended behaviors, we plan to develop an approach for synthesizing concise test report. The report will concisely present to the developers with conditions under which the two versions have a different behavior. The developers can inspect the report to effectively find out whether the behavioral differences are intended. The approach will use the path conditions followed by the failing tests in the original and modified application versions, find the differences between the two path conditions, and present concise conditions for which the behaviors differ.

Chapter 7

Assessment and Conclusions

In this chapter, we summarize the contributions of this dissertation and discuss the important lessons learned during the course of this dissertation.

7.1 Summary of Contributions

In this dissertation, we have presented a framework that improves the quality assurance of Database Centric Applications (DCAs). The framework contains a suite of approaches for improving the quality assurance of DCAs.

First, in presence of privacy laws, when the data in the backend-end database (DB) of the DCA under test cannot be released to testing teams, we have presented an approach, called PRIEST, for data anonymization to balance the need for testing and privacy. PRIEST includes a scalable program analysis technique to determine how the database attributes affect test coverage of the front-end application (FA) of the DCA. We combine the program analysis technique with a new privacy framework that includes a novel combination of guessing anonymity-based privacy metrics and a technique of data swapping anonymization to enable organizations to keep original values in sanitized data; keeping such values is important for improving the effectiveness of testing. As a result, business analysts are able to balance data privacy with test coverage. To evaluate our approach, we have conducted experiments on three open source DCAs and one large real-world DCA that handles logistics of one of the largest supermarket chains in Spain. Our results strongly suggest that our approach helps achieve higher test coverage for given levels of privacy for subject applications when compared with a data anonymization tool based on data suppression and generalization techniques.

Second, when the DB of the DCA under test contains insufficient records to cover various FA branches, we have presented an approach, called MODA, for generating tests for the DCA. The tests not only include inputs for the FA of the DCA but also for the DB such that various FA branches that are dependent on the DB can be covered. MODA bridges the gap between the FA and the DB by synthesiz-

ing mock objects for the DB. As a result, a test generation tool generates inputs for the mock DB such that various FA branches that are dependent on the DB can be covered. To evaluate MODA, we have conducted experiments on an open source DCA and a real-world medical device DCA. Experimental results show that MODA can help a test generation tool in achieving 20% higher branch (on average) coverage than what the tool achieves without MODA.

Third, we have presented an approach, called DiffGen, for effective generation of regression tests for the FA of the DCA under test. DiffGen takes two versions (an old and a modified versions) of an application and automatically generates effective regression tests. In particular, DiffGen instruments the application under test to insert branches such that if these branches can be covered, behavioral (or state) differences between the two versions can be exposed. Developers can inspect the tests that detect behavioral (or state) differences to find regression faults. To evaluate DiffGen, we have conducted experiments on 21 open source classes and their versions. The experimental results show that our approach can effectively detect various behavioral and state differences (between original and faulty versions) that cannot be detected by a state-of-the-art approach.

Fourth, we have presented an approach, called eXpress, for efficient generation of regression tests for the FA of the DCA under test. To efficiently generate regression tests, eXpress prunes various paths or branches, improving the search strategy of path-exploration-based test generation (PBTG) tools. The execution of the pruned paths and branches guarantees not to satisfy the conditions to find behavioral differences between two versions of an application under test. As a result, behavioral differences are more likely to be found earlier in path exploration. In addition, eXpress can exploit the existing test suite for the original version to efficiently execute the changed code regions (if not already covered by the test suite). To evaluate eXpress, we have conducted experiments on 67 versions (in total) of four open source applications. Experimental results show that Pex (a PBTG tool) using eXpress requires about 36% less amount of time (on average) to detect behavioral differences than without using eXpress. In addition, Pex using eXpress detects four behavioral difference that could not be detected without using eXpress (within a time bound). DiffGen and eXpress can be applied for regression testing of the whole DCA in combination with MODA.

7.2 Lessons Learned

We next summarize some lessons learned during the course of this dissertation.

7.2.1 Tool Automation

Development of a fully automated research prototype helps in making a bigger impact of the research. On one hand, most parts of DiffGen (such as synthesis of a test driver) were not automated; on the other hand, PRIEST is fully automated and has a graphical user interface for people to try out. As a result,

PRIEST has been tried out by various Accenture teams helping their clients facing the same problem addressed by PRIEST, making it easier for industry adoption.

7.2.2 Building on Top of Industrial Tools

We built DiffGen on top of jCUTE and JUnit Factory. As jCUTE is a research prototype with many limitations, we could not scale DiffGen to conduct experiments on larger subject applications. However, using JUnit Factory, we were able to do the experiments on larger subject applications. We built eXpress on top of Pex, an automated structural testing tool for .NET developed at Microsoft Research. Pex has been applied internally in Microsoft to test core components of the .NET runtime infrastructure and found serious faults. Using Pex not only helps in making our approach scalable but also helps in making an impact in the industry.

7.2.3 Adapting Techniques from Other Research Areas

When existing approaches from a different field are applied to software engineering, they often face problems providing opportunities for novel research. Our PRIEST approach applies data anonymization techniques from the field of data privacy to the field of software testing. However, the existing approaches for data anonymization resulted in poor test coverage. As a result, a new anonymization framework was needed to balance the need for testing and data anonymization, providing opportunities of novel research in the area.

7.2.4 Help from Humans

In our PRIEST approach, we require humans to help in linking application variables to database attributes using annotations. Annotating is not much work for a developer but a big help for our tool in improving its precision. Those annotations could be added automatically but they might reduce the precision of the tool.

7.2.5 Problem-driven Methodology

Our PRIEST approach demonstrated the effectiveness of using a problem-driven methodology rather than a solution-driven methodology. We came to know about the problem that organizations face, while anonymizing test data, after interviewing professionals at IBM, Accenture, two large health insurance companies, a biopharmaceutical company, two large supermarket chains, and three major banks. To the best of our knowledge, we were the first one to highlight and address the problem.

REFERENCES

[1] Charu C. Aggarwal. On k-anonymity and the curse of dimensionality. In *Proc. VLDB*, pages 901–909, 2005.

[2] Charu C. Aggarwal. On randomization, public information and the curse of dimensionality. In *Proc. ICDE*, pages 136–145, 2007.

[3] Charu C. Aggarwal and Philip S. Yu. On static and dynamic methods for condensation-based privacy-preserving data mining. *TODS*, 33:1–39, 2008.

[4] Charu C. Aggarwal and Philip S. Yu. *Privacy-Preserving Data Mining: Models and Algorithms*. Springer, 2008.

[5] J. Trent Alexander, Michael Davern, and Betsey Stevenson. Inaccurate age and sex data in the census pums files: Evidence and implications. Working Paper 15703, National Bureau of Economic Research, January 2010.

[6] Scott Ambler. Robust persistence layers. *Softw. Dev.*, 6(2):73–75, 1998.

[7] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. ICSE*, pages 402–411. ACM, 2005.

[8] Taweesup Apiwattanapong, Raul Santelices, Pavan Kumar Chittimalli, Alessandro Orso, and Mary Jean Harrold. Matrix: Maintenance-oriented testing requirement identifier and examiner. In *Proc. TAICPART*, pages 137–146, 2006.

[9] W. Aspray, F. Mayades, and M. Vardi. *Globalization and Offshoring of Software*. ACM, 2006.

[10] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. A Genetic Approach for Random Testing of Database Systems. In *Proc. VLDB*, pages 1243–1251, 2007.

[11] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[12] Carl Bialik. Census bureau obscured personal data – too well, some say. *The Wall Street Journal*, February 2010.

[13] C. Binnig, D. Kossmann, and E. Lo. Reverse Query Processing. In *Proc. ICDE*, pages 506–515, 2007.

[14] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.

[15] B.W. Boehm and P.N. Papaccio. Understanding and controlling software costs. *TSE*, 14:1462–1477, 1988.

[16] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In *Proc. ISSTA*, pages 169–180, 2006.

[17] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *Proc. ISSTA*, pages 123–133, 2002.

[18] Justin Brickell and Vitaly Shmatikov. The cost of privacy: destruction of data-mining utility in anonymized data publishing. In *Proc. KDD*, pages 70–78, 2008.

[19] Aditya Budi, David Lo, Lingxiao Jiang, and Lucia. *b*-anonymity: a model for anonymized behaviour-preserving test and debugging data. In *Proc. PLDI*, pages 447–457, 2011.

[20] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proc. ASE*, pages 443–446, 2008.

[21] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Proc. CCS*, pages 322–335, 2006.

[22] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56:82–90, 2013.

[23] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. Better bug reporting with better privacy. In *Proc. ASPLOS*, pages 319–328, 2008.

[24] David Chays, Yuetang Deng, Phyllis G. Frankl, Saikat Dan, Filippos I. Vokolos, and Elaine J. Weyuker. An AGENDA for Testing Relational Database Applications. *STVR*, 14:17–44, 2004.

[25] Mike Clark. Junit primer. Draft manuscript, 2000.

[26] Lori Clarke. A system to generate test data and symbolically execute programs. *TSE*, 2(3):215–222, 1976.

[27] James Clause and Alessandro Orso. Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting. In *ISSTA*, pages 249–260, 2009.

[28] James A. Clause and Alessandro Orso. Camouflage: automated anonymization of field data. In *Proc. ICSE*, pages 21–30, 2011.

[29] Graham Cormode and Divesh Srivastava. Anonymized data: generation, models, usage. In *Proc. SIGMOD*, pages 1015–1018, 2009.

[30] Datamonitor. Application testing services: global market forecast model. *Datamonitor Research Store*, August 2007.

[31] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[32] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *TSE*, 17(9):900–910, 1991.

[33] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automatic detection of refactorings in evolving components. In *Proc. ECOOP*, pages 404–428, 2006.

[34] Irit Dinur and Kobbi Nissim. Revealing information while preserving privacy. In *Proc. PODS*, pages 202–210, 2003.

[35] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *ESE*, 10(4):405–435, 2005.

[36] Josep Domingo-Ferrer and David Rebollo-Monedero. Measuring risk and utility of anonymized data using information theory. In *Proc. EDBT/ICDT*, pages 126–130, 2009.

[37] Sebastian Elbaum, Hui Nee Chin, Matthew Dwyer, and Jonathan Dokulil. Carving differential unit test cases from system test cases. In *Proc. FSE*, pages 253–264, 2006.

[38] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Matthew Jorde. Carving and replaying differential unit test cases from system test cases. *TSE*, 35(1):29–45, 2009.

[39] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic Test Input Generation for Database Applications. In *Proc. ISSTA*, pages 151–162, 2007.

[40] A. Endres and Dieter Rombach. *A Handbook of Software and Systems Engineering*. Pearson Addison-Wesley, 2003.

[41] Robert B. Evans and Alberto Savoia. Differential testing: A new approach to change detection. In *Proc. FSE*, pages 549–552, 2007.

[42] B. C. M. Fung, K. Wang, A. W.-C. Fu, and P. S. Yu. *Introduction to Privacy-Preserving Data Publishing: Concepts and Techniques*. Data Mining and Knowledge Discovery. Chapman & Hall/CRC, August 2010.

[43] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. In *ACM SIGACT News*, pages 25–36, 2002.

[44] Patrice Godefroid. Compositional dynamic test generation. In *Proc. POPL*, pages 47–54, 2007.

[45] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. *Proc. PLDI*, pages 213–223, 2005.

[46] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proc. NDSS*, pages 151–166, 2008.

[47] Mark Grechanik, Christoph Csallner, Chen Fu, and Qing Xie. Is data privacy always good for software testing? In *Proc. ISSRE*, pages 368–377, 2010.

[48] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *Proc. ESEM*, pages 11:1–11:10, 2010.

[49] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Syst. J.*, 41(1):4–12, 2002.

[50] William G. J. Halfond and Alessandro Orso. Command-Form Coverage for Testing Database Applications. In *Proc. ASE*, pages 69–80, 2006.

[51] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *TSE*, 30(1):3–16, 2004.

[52] Johannes Henkel and Amer Diwan. Discovering algebraic specifications from Java classes. In *Proc. ECOOP*, pages 431–456, 2003.

[53] Susan Horwitz. Tool support for improving test coverage. In *Proc. ESOP*, pages 162–177, 2002.

[54] Kobi Inkumsah and Tao Xie. Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In *Proc. ASE*, pages 425–428, 2007.

[55] Kobi Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proc. ASE*, pages 297–306, 2008.

[56] Automated Testing Institute. Software testing: The fastest growing it service. *http://tinyurl.com/softwaretestingfastest*, 2009.

[57] Wei Jin, Alex Orso, and Tao Xie. Automated behavioral regression testing. In *Proc. ICST*, pages 137–146, 2010.

[58] Capers Jones. *Software Engineering Best Practices*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010.

[59] Pallavi Joshi, Koushik Sen, and Mark Shlimovich. Predictive testing: Amplifying the effectiveness of software testing. In *Proc. ESEC-FSE*, pages 561–564, 2007.

[60] Jtopas website, 2006. `http://jtopas.sourceforge.net/jtopas/`.

[61] Junit factory website, 2006. `http://www.JunitFactory.com/`.

[62] Gregory M. Kapfhammer and Mary Lou Soffa. A Family of Test Adequacy Criteria for Database-Driven Applications. In *Proc. ESEC/FSE*, pages 98–107, 2003.

[63] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[64] Bogdan Korel and Ali M. Al-Yami. Automated regression test generation. In *Proc. ISSTA*, pages 143–152, 1998.

[65] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.

[66] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proc. ICSE*, pages 308–318, 2003.

[67] J. Jenny Li. Prioritize code for testing to improve code coverage of complex software. In *Proc. ISSRE*, pages 75–84, 2005.

[68] Tiancheng Li and Ninghui Li. On the tradeoff between privacy and utility in data publishing. In *Proc. KDD*, pages 517–526, 2009.

[69] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: an automated class mutation system: Research articles. *SSTVR*, 15(2):97–133, 2005.

[70] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-Testing: Unit Testing with Mock Objects. pages 287–301, 2001.

[71] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proc. ICSE*, pages 416–426, 2007.

[72] Madhuri R Marri, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. An Empirical Study of Testing File-System-Dependent Software with Mock Objects. In *Proc. AST*, pages 149–153, 2009.

[73] Ellen Messmer. International data privacy laws. *http://www.informationshield.com/intprivacylaws.html*, 2010.

[74] Thomas E. Murphy. Managing test data for maximum productivity. *http://www.gartner.com/DisplayDocument ?doc_cd=163662&ref=g_economy_2reduce*, December 2008.

[75] Alessandro Orso and Bryan Kennedy. Selective capture and replay of program executions. In *Proc.WODA*, pages 1–7, 2005.

[76] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding Errors in .NET with Feedback-Directed Random Testing. In *Proc. ISSTA*, 2008.

[77] Parasoft Jtest manuals version 4.5. Online manual, 2003. `http://www.parasoft.com/`.

[78] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proc. FSE*, pages 226–237, 2008.

[79] Sanju Pillai. Outsourcing regression testing to experts a way to improve your softwares quality. *http://tinyurl.com/regressionoutsource*, August 2011.

[80] Program Ofce Strategic Planning and Economic Analysis Group. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards & Technologay*, 2002.

[81] Jaroslav Pokorny. NoSQL databases: A step to database scalability in web environment. In *Proc. IIWAS*, pages 278–283, 2011.

[82] Dawei Qi, Abhik Roychoudhury, and Zhenkai Liang. Test generation to expose changes in evolving programs. In *Proc. ASE*, pages 397–406, 2010.

[83] Yaron Rachlin, Katharina Probst, and Rayid Ghani. Maximizing privacy under data distortion constraints in noise perturbation methods. In *Proc. PinKDD*, pages 92–110, 2008.

[84] Steven P. Reiss. Practical data-swapping: the first steps. *ACM Trans. Database Syst.*, 9:20–37, March 1984.

[85] Steven P. Reiss, Mark J. Post, and Tore Dalenius. Non-reversible privacy transformations. In *Proc. ODS*, pages 139–146, 1982.

[86] Xiaoxia Ren, Barbara G. Ryder, Maximilian Stoerzer, and Frank Tip. Chianti: A change impact analysis tool for java programs. In *Proc. ICSE*, pages 664–665, 2005.

[87] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *TOSEM*, 6(2):173–210, 1997.

[88] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for Java. In *Proc. ASE*, pages 114–123, 2005.

[89] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *Proc. ISSRE*, pages 281–292, 2003.

[90] Pierangela Samarati. Protecting respondents' identities in microdata release. *TKDE*, 13(6):1010–1027, 2001.

[91] Raul Andres Santelices, PavanKumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Test-suite augmentation for evolving software. In *Proc. ASE*, pages 0–0, 2008.

[92] N. F. Schneidewind. The state of software maintenance. *TSE*, 13(3):303–310, 1987.

[93] Wolfram Schulte. Pex–an intelligent assistant for rigorous developer testing. In *Proc. ICECCS*, page 161. IEEE Computer Society, 2007.

[94] Koushik Sen and Gul Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *Proc. CAV*, pages 419–423, 2006. (Tool Paper).

[95] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proc. FSE*, pages 263–272, 2005.

[96] Hina Shah, Saurabh Sinha, and Mary Jean Harrold. Outsourced, offshored software-testing practice: Vendor-side experiences. In *Proc. ICGSE*, pages 131–140, 2011.

[97] Information Shield. The worst data breaches of 2012 (so far). *http://www.cio.com/slideshow/detail/52577*, 2012.

[98] Amie L. Souter and Lori L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. In *Proc. ICSM*, pages 682–691, 2001.

[99] "MySQL API Specifications". `http://msdn.microsoft.com/en-us/library/system.data.sqlclient.aspx`.

[100] David Stotts, Mark Lindsey, and Angus Antley. An informal formal method for systematic JUnit test case generation. In *Proc. XP/Agile Universe*, 2002.

[101] Zhendong Su and Gary Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proc. POPL*, pages 372–382, 2006.

[102] María José Suárez-Cabal and Javier Tuya. Using an SQL Coverage Measurement for Testing Database Applications. *SIGSOFT Softw. Eng. Notes*, 29:253–262, 2004.

[103] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[104] Kunal Taneja, Danny Dig, and Tao Xie. Automated detection of API refactorings in libraries. In *Proc. ASE*, pages 377–380, November 2007.

[105] Kunal Taneja and Tao Xie. DiffGen: Automated regression unit-test generation. In *Proc. ASE*, pages 407–410, 2008.

[106] Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. express: Guided path exploration for efficient regression test generation. In *Proc. ISSTA*, pages 1–11, 2011.

[107] Kunal Taneja, Yi Zhang, and Tao Xie. MODA: Automated test generation for database applications via mock objects. In *Proc. ASE*, 2010.

[108] Bennie G. Thompson. *H.R.6423: Homeland Security Cyber and Physical Infrastructure Protection Act of 2010*. U.S.House, 111th Congress, November 2010.

[109] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proc. SIGMOD*, pages 1–12, 2012.

[110] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.

[111] Nikolai Tillmann and Wolfram Schulte. Mock-Object Generation with Behavior. In *Proc. ASE*, pages 365–368, 2006.

[112] Paolo Tonella. Evolutionary testing of classes. In *Proc. ISSTA*, pages 119–128, 2004.

[113] Javier Tuya, Ma José Suárez-Cabal, and Claudio de la Riva. Mutating Database Queries. *Information and Software Technology*, 49:398–417, 2007.

[114] Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. Symbolic Query Exploration. In *Proc. ICFEM*, 2009.

[115] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java Pathfinder. In *Proc. ISSTA*, pages 97–107, 2004.

[116] J.M. Voas. PIE: A dynamic failure-based technique. *TSE*, 18(8):717–727, 1992.

[117] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. *TOSEM*, 16:14, 2007.

[118] David Willmor and Suzanne M. Embury. An Intensional Approach to the Specification of Test Cases for Database Applications. In *Proc. ICSE*, pages 102–111, 2006.

[119] Joel Winstead and David Evans. Towards differential program analysis. In *Proc. WODA*, 2003.

[120] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. ECOOP*, pages 380–403, 2006.

[121] Tao Xie and David Notkin. Checking inside the black box: Regression testing by comparing value spectra. *TSE*, 31(10):869–883, 2005.

[122] Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. DSN*, pages 359–368, 2009.

[123] Zhihong Xu, Myra B. Cohen, and Gregg Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *Proc. GECCO*, pages 1365–1372, 2010.

[124] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *Proc. FSE*, pages 257–266, 2010.

[125] Zhihong Xu and Gregg Rothermel. Directed test suite augmentation. In *Proc. APSEC*, pages 406–413, 2009.

[126] Qian Yang, J. Jenny Li, and David M. Weiss. A survey of coverage-based testing tools. *Comput. J.*, 52(5):589–597, 2009.

[127] Shin Yoo and Mark Harman. Regression testing minimisation, selection and prioritisation: A survey. *STVR*, *to appear*, 2011.