ABSTRACT

, KUNAL TANEJA. Cost Reduction and Quiality Improvement of Softwre Maintenance. (Under the direction of Tao Xie.)

Successful software systems continue to evolve during their lifetimes. During their evolution, these systems undergo various kinds maintenance tasks including corrective maintenance tasks such as fixing defects, adaptive maintenance tasks such as modifications to adapt to a new/changed environment, perfective maintenance tasks such as performance improvements, and preventive maintenance tasks such as refactorings. However, Software Maintenance faces two major problems.

First, maintenance of evolving software is time consuming and labor-intensive. Software maintenance can consume as much as 90% of the total software development costs. Hence a majority of efforts on reducing software costs needs to be spent in reducing software maintenance costs. Second, maintenance of evolving software can result in unintended side effects (regression faults) in the software system. Regression testing is the process of detecting these faults. As the regression testing is a frequently performed activity during software maintenance phase, it occupies a large portion of the software maintenance budget. The current approaches in regression testing are not effective and efficient in finding regression faults.

In this thesis, we provide various solutions for the two preceding problems. First, we present Refaclib to reduce the cost of some adaptive changes made to evolving software by automating them. Second, we present DiffGen for effective regression testing of evolving Software. Third, we present eXpress for efficient regression testing. Fourth, we present ASSIST for integration-testing during maintenance. Fifth, we present PRIEST for generating regression test suite for testing software in presence of privacy laws.

Cost Reduction and Quiality Improvement of Softwre Maintenance

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

2012

APPROVED BY:

A	B
C	Tao Xie Chair of Advisory Committee

DEDICATION

To my parents.

BIOGRAPHY

The author was born in a small town ...

ACKNOWLEDGEMENTS

I would like to thank my advisor for his help.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Motivation

> *A program that is used in a real-world environment must change, or become progressively less useful in that environment. - Lehman and Belady [14]*

Traditionally, a software application 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. Software maintenance can be broadly categorized into four major types of maintenance tasks [15].

- **Corrective Maintenance.** Often a software application is released with existing defects in it. While some of these defects are often known, others are unkown at the time of delivery. The known defects are not fixed at the time of delivery due to tight deadlines or limited resources. The unknown defects are discovered in the field by the users of the software application. Moreover, new faults may be introduced in the software application while making changes to it during maintenance phase. Corrective software maintenance is the fixing of all these faults. Previous study [16] has found that corrective maintenance changes are the most difficult to make.

- **Adaptive maintenance.** Often a software application is dependent on its environment that includes hardware, operation system, and database. In addition, the software application may be dependent on other libraries that the software application uses. When the environment or the dependent libraries change to a new version, the software application

has to adapt to the changes. Adaptive maintenance includes making these changes to the software application.

- **Perfective Maintenance.** Often a software application is enhanced in terms of its efficiency, performance, usability, or functionality. Perfective software maintenance includes making changes to a software applicatication to make these enhancements.

- **Preventive Maintenance.** Continuous evolution of a software application often has an adverse effect on the structure of a software application [14], increasing its size, complexity and decreasing its comprehensability. As a result, the software application gets more difficult to maintain in the future. Preventive Software Maintenance includes making changes to a software application to prevent problems in the future. These changes do not aim to change the behavior of the application but aim to improve the future maintainability of the software application. These changes can include include refactorings, updating documentation or adding code comments.

Owing to long software lifetimes, software maintenance is an important software development phase. In addition, software maintenance is crucial for business success since the failure to a change software application quickly and reliably can result in losing business opportunities. However, there are two major challenges in changing a software application quickly and reliably.

- **Software Maintenance is Expensive.** Software maintenance is the most expensive activity of software development, comprising from 50% to 90% of the overall lifecycle costs [19]. It is expensive to maintain a software application due to various reasons. First, as a software application continues to evolve undergoing various kinds of changes[1], the application becomes more and more complex [14] and its structure tends to degrade. As a result, the application becomes less maintainable and software maintenance becomes labor intensive and time consuming. Second, often the software developers maintaining the software application are not the ones who developed it. As a result, it is more time consuming for these developers to maintain the software application resulting in high maintenance costs. Due to the preceding reasons, as a software application continues to evolve, the cost of maintenance tends to increase more and more. The software application continues to evolve until it reaches to a point when it becomes unprofitable to maintain and it is scraped. As a result, it is important to reduce the cost of software maintenance to sustain the software application in the market.

- **Software Maintenance can reduce software quality.** Software Maintenance (counter intuitive to its name) can itself result in reducing the software quality. The changes made

[1] Changes that are not a part of preventive maintenance tasks

to a software application can result in unintended side effects (regression faults). Even while performing corrective maintenance, the developers can introduce regression faults in the software application. As the software application becomes more and more complex, it become more and more error-prone to change. Since software maintenance is a long phase, it can result in a steady stream of new regression faults throughout the lifetime of the software application. It is no coincidence that efforts spent in corrective maintenance can be up to 37% of the total software maintenance efforts [7, 8, 12, 21, 16].

1.2 Problems and Proposed Solutions

This dissertation proposes a framework to address several problems that arise from the two major challenges mentioned in Section 1.1. The framework comprises of various approaches that aim to reduce the cost of software maintenance and increase the quality of an evolving software application. We next describe the respective solutions proposed by our framework.

- **P1.** To reduce development costs, software applications often reuse existing software libraries. In theory, the Application Programming Interfaces (APIs) of a library should be stable, but in practice they do change (as the library evolves) and thus require changes in a software system that reuses the library. Hence, the software application (using the old libaby API) needs to go through adaptive maintence to adapted to the new library API. Manually, adapting the software application to such changes is error-prone and time consuming. In addition, it is disruptive to other maintenance tasks. As a result, manually adapting the software application increases software maintenance costs and increases the chance of introducing regression faults in the software system.

  Previous studies [3] have shown that 80% of API breaking changes are refactorings. If these refactorings can be detected automatically, there are existing tools [9] that can automatically change the underlying software application that reuse the refactored libraries to adapt these software systems to the new library versions. As a result, cost of manually adapting the systems can be saved and the software developers can focus on other maintenance tasks. Our framework address the problem by proposing a solution to automatically detecting refactorings between two versions of a library.

- **P2.** Changes made to an evolving software system can introduce regression faults. Regression testing is the activity to detect whether the changes made to the software system are intended and do not introduce any side effects. Regression testing is carried out by executing the existing test suite (that was developed for a previous version of the software system) against the current version. The failing tests (that pass for the previous

software version) indicate behavioral differences between the two versions. 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 test suite may not be effective in finding behavioral differences making software maintenance error prone. Hence, regression tests that are effective in finding behavioral differences are needed. New tests can be generated using existing test generation tools [20, 4, 13, 1, 2, 23, 5, 22, 10], However, these tools aim at generating tests that can achieve high structural coverage. Hence, achieving high structural coverage is not sufficient for finding behavioral differences. To address the preceding issue, we propose an approach to bridge the gap between finding behavioral differences and coverage-based test generation tools.

- **P3.** 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) [20, 4, 13, 1, 2, 23, 5] and search-based test generation [22, 10] focus their efforts on increasing structural coverage and do not specically focus on detecting behavioral differences between two versions of a program. As a result, these techniques are inefcient 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 on detecting behavioral differences. As a result behavioral differences, are likely to be ditected more efficiently.

- **P4.** Database-centric (DCAs) are common in enterprise computing, and they use non-trivial databases [11]. Often these DCAs are outsourced for regression testing to data centers [18] to achieve lower cost and higher quality. 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 eld Nationality are

replaced with the generic value Human, DCAs may execute some paths that result in exceptions or miss certain paths [6]. As a result, test centers report worse test coverage (such as code coverage) and fewer uncovered regression faults, thereby reducing the quality of applications and obliterating the benets of test outsourcing [17]. To address the producing problem, we propose an approach that helps organizations can balance the level of privacy with needs of testing.

1.3 Contributions

This dissertation makes the following contributions

- **An approach for effectively finding refactoring between an evolving library.** We have implemented an efficient tool,RefacLib, to detect refactorings with practical accuracy inrealistic software systems. We have used RefacLib to nd several refactorings in ve real-world components. We compared Refac Lib with the previous state-of-the-art tool to detect refactorings and found out that our tool is comparable in most cases and better in others.

- **An approach for effective regression test generation.** We propose an approach, called DiffGen [citation] for generating regression tests that help in detecting behavioral differences between two versions of a given software system 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 expose behavioral differences that cannot be detected by previous state-of-the-art techniques [Cite Differential testing] based on achieving structural coverage on either version separately.

- **An approach for efficient regression test generation.** We propose an approach called eXpress [citation] for efcient 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 efciently by the PBTG technique with eXpress than without eXpress. We have implemented our eXpress approach in a tool as an extension for Pex [21], 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) [5] 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 nd behavioral differences by exploiting an existing test suite than exploration without using the test suite.

- **An approach for effective regression testing of software in presence of privacy laws.** We design and implement a technique using program analysisfor determining how values of database attributes affecttest coverage of DCAs that use this data . We combine our privacy framework with this technique inPRIEST to enable business analysts to balance data privacywith test coverage. We evaluate PRIEST using three open-source Java DCAsand one large Java DCA that handles logistics of one ofthe largest supermarket chains in Spain. We show that withPRIEST, test coverage of regression tests can be preserved at a higher level bypinpointing database attributes that should be anonymizedbased on their effect on corresponding DCAs.

1.4 Outline

The remainder of this dissertation is organized as follows. Chapter 2 describes REfaclib, Chapter 3 describes DiffGen. Chapter 4 describes eXpress. Chapter 5 describes PRIEST. Chapter 6 surveys related work. Finally, Chapter 7 concludes with suggestions for future work.

Chapter 2

DiffGen: Effective Generation of Regression Tests

Chapter 3

Automatically Synthesizing Integration Tests Using Acceptance and Unit Tests

Chapter 4

PRIEST

Chapter 5

Related Work

Chapter 6

Future Work

Chapter 7

Conclusion

REFERENCES

[1] 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.

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

[3] Danny Dig and Ralph Johnson. How do apis evolve? a story of refactoring: Research articles. *J. Softw. Maint. Evol.*, 18(2):83–107, March 2006.

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

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

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

[7] Les Hatton. How accurately do engineers predict software maintenance tasks? *Computer*, 40(2):64–69, 2007.

[8] Glenn L. Helms and Ira R. Weiss. Application software maintenance: can it be controlled? *SIGMIS Database*, 16(2):16–18, 1984.

[9] Johannes Henkel. Catchup!: Capturing and replaying refactorings to support api evolution. In *Proc. ICSE*, pages 274–283, 2005.

[10] 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.

[11] Gregory M. Kapfhammer and Mary Lou Soffa. A family of test adequacy criteria for database-driven applications. In *ESEC/FSE*, pages 98–107, 2003.

[12] Chris F. Kemerer, Sandra, and Sandra A. Slaughter. Determinants of software maintenance profiles: An empirical investigation. *Journal of Software Maintenance*, 9:235–251, 1997.

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

[14] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[15] Bennet P. Lientz and Burton E. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, 1980.

[16] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *Proc. ICSM*, pages 120–129, 2000.

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

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

[19] N. F. Schneidewind. The state of software maintenance. *IEEE Trans. Softw. Eng.*, 13(3):303–310, 1987.

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

[21] Harry M. Sneed. Modelling the maintenance process at zurich life insurance. In *Proc. ICSM*, pages 217–226, 1996.

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

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