Applying program analysis augmented with empirical properties into software analysis, testing and maintenance

Ngo, Minh Ngoc

2009

Ngo, M. N. (2009). Applying program analysis augmented with empirical properties into software analysis, testing and maintenance. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/41497>

<https://doi.org/10.32657/10356/41497>

APPLYING PROGRAM ANALYSIS AUGMENTED WITH EMPIRICAL PROPERTIES INTO SOFTWARE ANALYSIS, TESTING AND MAINTENANCE

NGO MINH NGOC

School of Electrical and Electronic Engineering

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirement for the degree of
Doctor of Philosophy

2009

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to Prof. Tan Hee Beng Kuan for his continuous support, inspiring encouragement and great patience. He taught me how to clearly present my ideas and solutions and how to focus on the important problems. Without his guidance, this thesis would never be completed.

Collaborating with industry partners is one of the enriching experiences during my graduate years. I would like to thank Ulf Pettersson, Ong Wee Tat and Daryl Amores Versola for providing the codes and the environment for me to carry out some experiments and discussing some interesting insights into industrial practices.

An important part of my research is done in cooperation with undergraduate and Msc students. I really appreciate their inputs and participations in various experiments.

I wish to thank my friends and technician staffs at the Infocomm Research Lab who have created a friendly working environment. Especially, for many interesting lunch time discussions and useful advices for my thesis, thanks to Joseph.

I am eternally indebt of my parents for their unconditional love and support which the words cannot describe. Thanks to my little brother from whom I have learned that if you cannot explain to a 10-year-old kid, you do not understand the problem at all. I would not have made it here without my husband, whose love and emotional support have always been positive and motivating encouragement to me throughout this unforgettable journey.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	I
TABLE OF CONTENTS	II
SUMMARY	V
LIST OF FIGURES	VII
LIST OF TABLES	IX
CHAPTER 1 INTRODUCTION.....	1
1.1 MOTIVATIONS.....	1
1.1.1 Infeasible path detection	2
1.1.2 Recovery, testing and maintenance of input error correction ..	3
1.1.3 Automated extraction of database interactions	4
1.2 OBJECTIVES	4
1.3 MAJOR CONTRIBUTIONS OF THE THESIS	5
1.4 ORGANIZATION OF THE THESIS	6
CHAPTER 2 BACKGROUND.....	8
2.1 TRADITIONAL PROGRAM ANALYSIS.....	8
2.1.1 Control flow analysis	9
2.1.2 Control dependence analysis	10
2.1.3 Data flow analysis.....	13
2.1.4 Program slicing	14
2.1.5 Challenges.....	16
2.2 EMPIRICAL APPROACHES TO PROGRAM ANALYSIS.....	17
2.2.1 Empirical methods	17
2.2.2 The role of experiments in software engineering	21
2.2.3 The use of empirical methods program analysis	23
2.3 CONCLUSION	27
CHAPTER 3 RELATED WORK	28
3.1 INFEASIBLE PATH DETECTION	29
3.1.1 Precise methods	31
3.1.2 Approximate methods.....	32
3.2 INPUT ERROR CORRECTION RECOVERY, TESTING AND MAINTENANCE.....	34
3.2.1 Input error correction	34
3.2.2 Design recovery	36
3.2.3 Integrated functional and structural testing.....	39
3.2.4 Using slicing in software maintenance	43
3.3 REVERSE ENGINEERING OF WEB APPLICATIONS.....	46

3.4	SUMMARY	49
CHAPTER 4 A GENERIC INFRASTRUCTURE TO		
SUPPORT EMPIRICAL PROGRAM ANALYSIS		51
4.1	EMANALYZER	51
4.1.1	Soot	52
4.1.2	PathGen	56
4.1.3	EmViewer	58
4.2	EMVALIDATOR	60
4.2.1	TestGen	62
4.2.2	JTracer	63
4.2.3	Kaveri	65
4.3	SUMMARY	66
CHAPTER 5 INFEASIBLE PATH DETECTION		68
5.1	INTRODUCTION	68
5.2	INFEASIBLE PATH PATTERNS	70
5.2.1	Conflicting-decision pattern	70
5.2.2	Mutually-exclusive-decision pattern	72
5.2.3	Check-then-do pattern	74
5.2.4	Looping-by-flag pattern	77
5.3	STATISTICAL VALIDATION	78
5.4	DETECTING INFEASIBLE PATHS	82
5.4.1	The algorithms	82
5.4.2	Compare with existing approaches	87
5.4.3	Time complexity	87
5.5	EVALUATION	88
5.5.1	The prototype system	88
5.5.2	Experiment	89
5.5.3	Results	91
5.6	ARISPATH: AN INFEASIBLE PATH DETECTOR FOR C	94
5.7	SUMMARY	96
CHAPTER 6 AUTOMATED RECOVERY, TESTING AND		
MAINTENANCE OF INPUT ERROR CORRECTION		99
6.1	INTRODUCTION	99
6.2	AN EXAMPLE APPLICATION	101
6.3	A THEORY FOR INFERRING INPUT ERROR CORRECTION	104
6.3.1	Modeling user input error	104
6.3.2	Inferring effect error	106
6.3.3	Inferring input error correction	108
6.4	AUTOMATED RECOVERY	115
6.5	INTEGRATING FUNCTIONAL AND STRUCTURAL TESTING	118
6.6	AIDING MAINTENANCE OF INPUT ERROR CORRECTION	125

6.6.1	Effect-oriented decomposition slicing	125
6.6.2	Removing error correction for a type of input errors.....	127
6.6.3	Adding error correction for a type of input errors	129
6.7	THE PROTOTYPE SYSTEM	129
6.8	VALIDATION AND EVALUATION	133
6.8.1	Statistical validation.....	133
6.8.2	Exploratory study 1: Testing input error correction	138
6.8.3	Exploratory study 2: Aiding maintenance	141
6.9	COMPARISON WITH RELATED WORK	144
6.10	SUMMARY.....	146
CHAPTER 7 DATABASE INTERACTION EXTRACTION FOR WEB APPLICATIONS		148
7.1	INTRODUCTION	148
7.2	AN EXAMPLE APPLICATION	150
7.3	ANALYSIS MODEL	152
7.4	HYBRID SYMBOLIC EXECUTION.....	155
7.4.1	Binary-expression rule.....	156
7.4.2	If-statement rule.....	157
7.4.3	Assignment-statement rule	158
7.4.4	Loop-statement rule	160
7.4.5	Function-call rule.....	160
7.5	AUTOMATED EXTRACTION.....	161
7.5.1	Stage 1 – Symbolic expressions derivation	161
7.5.2	Stage 2 – Database Interactions Inference	163
7.6	EXPERIMENTS AND EVALUATION	167
7.6.1	The prototype system.....	167
7.6.2	Experiments	168
7.6.3	Time complexity	177
7.7	SUMMARY.....	177
CHAPTER 8 CONCLUSIONS & RECOMMENDATIONS..		179
8.1	CONCLUSION	179
8.2	LIMITATION OF THE STUDY	184
8.3	RECOMMENDATIONS FOR FURTHER RESEARCH	184
APPENDIX A		187
APPENDIX B		189
AUTHOR’S PUBLICATIONS.....		190
BIBLIOGRAPHY		191

SUMMARY

Program analysis techniques are often used to deduce and infer targeted characteristics of software systems for various software engineering tasks such as testing, debugging, maintenance and program comprehension. Recently, researchers and practitioners have recognized the importance of empirical knowledge in the evolution of software engineering discipline. The use of empirical knowledge, which is validated statistically, might be able to draw inferences about the properties of software systems in cases in which more traditional analyses have not succeeded.

Our research project explores a new approach toward software analysis, testing and maintenance through augmenting program analysis techniques with empirical properties which are validated statistically. Through applying the augmented method, we establish solutions for: (1) infeasible path detection, (2) input error correction recovery, testing and maintenance, (3) database interaction extraction for web applications.

Infeasible path detection is one of the biggest challenges in program analysis. We discover that many infeasible paths exhibit some common properties which are caused by four code patterns. Through realizing these properties from source code, many infeasible paths can be precisely detected. Binomial tests have been conducted which give strong statistical evidences to support the validity of the empirical properties. Our experimental results show that even with some limitations in the current prototype tool, the proposed approach accurately detects more than 80% of all the infeasible paths.

Input error correction plays an important part in any database application. Unfortunately, recovery, testing and maintenance of this feature are tedious and error-prone tasks as there are many types of user input errors, each of which might cause several erroneous effects. In this thesis, we formulate a theory for inferring input error correction from source code. Based on the theory, we propose an approach for the automated recovery testing and maintenance of input error correction. The theory has been validated statistically. Case studies have also been conducted to evaluate the usefulness of the proposed approach.

Automated extraction of database interactions is useful for web application testing and maintenance as most web applications interact with one or more databases. In this thesis, we propose an automated approach to extract database

interactions by using symbolic evaluation and inference rules. The inference rules are used to deduce the interaction types from the set of symbolic expressions derived during the symbolic evaluation. All the empirical inference rules have been validated statistically. Experiments have been conducted which shows that the proposed approach gives an average precision of 79.2%, which is 44.4% more than that of the conservative approach.

We believe that the use of program analysis augmented with empirical properties is a promising direction, which opens a new avenue for improving existing program analysis techniques as well as creating new techniques to support various software engineering activities.

LIST OF FIGURES

Figure 2-1. An example procedure	11
Figure 2-2. Experiment principles (adapted from [214]).....	18
Figure 2-3. A hierarchy for program analysis (from Zeller [227]).....	24
Figure 3-1. The compute_grade procedure.....	30
Figure 3-2. CFG of the compute_grade procedure	30
Figure 3-3. Structure of a Web Application (Adapted from [87]).....	46
Figure 4-1. EmAnalyzer architecture.....	52
Figure 4-2. Part of the class diagram of Soot	54
Figure 4-3. Different types of CFGs in Soot	55
Figure 4-4. Soot drivers	56
Figure 4-5. Algorithm findBasisPaths	57
Figure 4-6. Algorithm findRPaths	57
Figure 4-7. CFG display in EmAnalyzer	59
Figure 4-8. Source code view in EmAnalyzer	60
Figure 4-9. EmValidator Architecture	61
Figure 4-10. An example program for Choco.....	62
Figure 4-11. Creating a new monitor object using InsectJ	65
Figure 5-1. Coverage rate vs. Effort (Taken from [44])	69
Figure 5-2. Conflicting-decision pattern.....	70
Figure 5-3. Infeasible paths in conflicting-decision pattern	71
Figure 5-4. Nested-if pattern.....	72
Figure 5-5. Mutually-exclusive-decision pattern example and its CFG.....	73
Figure 5-6. Check-then-do pattern.....	75
Figure 5-7. Two types of infeasible paths in check-then-do pattern.....	76
Figure 5-8. Looping-by-flag pattern	77
Figure 5-9. Detect infeasible paths in a given set of paths	82
Figure 5-10. Detect infeasible paths in conflicting-decision pattern	83
Figure 5-11. Detect infeasible paths in mutually-exclusive-decision pattern.....	84
Figure 5-12. Detect infeasible paths in check-then-do pattern	85
Figure 5-13. Detect infeasible paths in looping-by-flag pattern.....	86
Figure 5-14. InfeasibleDetector tool.....	88
Figure 5-15. A case not detected by the proposed approach (1).....	92
Figure 5-16. A case not detected by the proposed approach (2).....	93
Figure 6-1. Procedure process_order and its CFG.....	102
Figure 6-2. Procedure cancel_item and its CFG	102
Figure 6-3. Procedure cancel_order and its CFG	103

Figure 6-4. Procedure change_header and its CFG	104
Figure 6-5. Refined IEC-specification of S_1	124
Figure 6-6. The prototype system	131
Figure 7-1. register_course.jsp.....	150
Figure 7-2. delete_course.jsp	151
Figure 7-3. CFG of register_course.jsp.....	153
Figure 7-4. Algorithm createIFG	155
Figure 7-5. IFG of register_course.jsp.....	155
Figure 7-6. Binary-expression rule	156
Figure 7-7. If-statement rule	157
Figure 7-8. Assignment-statement rule.....	158
Figure 7-9. Array-rule 1	159
Figure 7-10. Array-rule 2.....	159
Figure 7-11. Loop-statement rule	160
Figure 7-12. Ipath-loop rule.....	162
Figure 7-13. Filtering procedure for T_A	165
Figure 7-14. Prototype system	167
Figure B-1. An example program using Choco	189

LIST OF TABLES

Table 2-1. Relevant sets for (8, a).....	15
Table 2-2. Tasks in experimental program analysis (from Ruthruff et al. [180]).....	26
Table 4-1. Probes for instrumentable entities	63
Table 5-1. Samples for testing empirical properties	80
Table 5-2. Descriptions of target systems.....	89
Table 5-3. False-positive and true-negative identification	90
Table 5-4. Experimental results	91
Table 5-5. Sample program taken from SIR.....	95
Table 5-6. Experiment results with C programs	95
Table 5-7. Comparison of experiment results with Java and C	96
Table 6-1. R-paths in the E-Order application.....	113
Table 6-2. Input structures and input blocks.....	113
Table 6-3. Basic collections of error correction paths	114
Table 6-4. CIE partition of E-Order.....	119
Table 6-5. SIE partition of E-Order	121
Table 6-6. Non-empty intersection of SIE classes and CIE classes	123
Table 6-7. Description of systems for hypothesis testing	133
Table 6-8. Sample size for testing empirical properties	135
Table 6-9. Z-scores	137
Table 6-10. Fault detected by each approach	138
Table 6-11. Study results	143
Table 7-1. Sample for testing Inference rules 2, 3 and 4	170
Table 7-2. Sample Java systems	170
Table 7-3. Sample PHP systems	173
Table 7-4. Results of experiment 2	176
Table A-1. General program analysis notations	187
Table A-2. Slicing notations	187
Table A-3. Additional notations used by Chapter 6	188
Table A-4. Additional notations used by Chapter 7	188

Chapter 1

INTRODUCTION

1.1 Motivations

Reasoning about and understanding program artifacts is a time-consuming, yet key activities in many important software engineering tasks such as testing, debugging, maintenance and program comprehension. Program analysis techniques are often used to automatically extract information from software systems to support these activities. Although many program analysis techniques have been developed, they are rarely applied in practice. One reason for this is their current inability to scale to large systems [83]. This fact has lead researchers to realize the importance of empirical knowledge in the evolution of software engineering discipline.

The use of empirical knowledge, which is validated statistically, is very common in many fields such as physics, medicines and manufacturing. For example, in the area of medicine [13], the medical researcher aims at understanding the working mechanism of the human body in order to predict the effects of various procedures and drugs and to provide empirical knowledge about human health and well-being. The medical practitioner applies the empirical knowledge for the purpose of curing the illness. However, the application of this method is rather unexplored in software engineering. “*Most of the empirical studies in software engineering so far, are used to demonstrate the feasibility or compare the performance of tools*” [83]. They are rarely used to validate or quantify the fundamental assumption underlying research. Recent research suggests that the use of empirical knowledge in program analysis might be able to draw inferences about properties of software systems in cases in which more traditional analyses have not succeeded [180].

We realize that many characteristics of scientific experimentation could be utilized to discover, refine and validate special properties of software systems. These characteristics include the formation and testing of hypotheses on the properties of the object under analysis, the iterative process of exploring and adjusting these hypotheses in response to findings and the use of sampling to cost-effectively explore effects relative to large populations in generalizable

manners. We refer to the properties which are established and validated experimentally as **empirical properties**. By applying traditional program analysis techniques augmented with these empirical properties, targeted characteristics can be extracted from software systems to support various software engineering activities. Our preliminary research suggests that this approach might create scalable, feasible and effective solutions for many software engineering problems.

Motivated by these initial finding and the importance of empirical knowledge in software engineering [12, 13, 163, 192, 201, 226], we have conducted large-scale empirical studies to explore the use of program analysis augmented with empirical studies to solve problems in software analysis, testing and maintenance. More specifically, through applying this approach, this thesis addresses the following problems: (1) infeasible path detection, (2) recovery, testing and maintenance of input error correction for database applications and (3) automated extraction of database interactions for web applications. Next, we discuss the challenges and explain our motivations for solving each of these problems.

1.1.1 Infeasible path detection

Static program analysis is undoubtedly an integral part of many software engineering activities [22, 64, 74, 139]. However, one of the biggest challenges for static analysis is the existence of infeasible program paths in that there is no input for which the paths will be executed. This is because static analysis is done before execution; one commonly made assumption during static analysis is that every program path is executable. Such conservative analysis too often yields imprecise results. According to Hedly et al. [89] 12.5% of all the paths are infeasible. If a majority of infeasible paths can be detected during static analysis, it will greatly improve the performance of many software engineering tasks, especially structural testing and coverage analysis.

In general, detecting infeasible paths is theoretically unsolvable. Many techniques rely on symbolic evaluation [23, 74, 225], thus trading expensive computation for sharper analytical results. Moreover, due to the limitation of symbolic evaluation in handling pointers, arrays and function calls, only a small percentage of infeasible paths can be detected. Another class of approaches is based on properties of infeasible paths which are empirically discovered [27, 64, 139]. These approaches are fast and effective. However, to the best of our knowledge, none of the existing heuristics is capable of detecting a large number of infeasible paths.

1.1.2 Recovery, testing and maintenance of input error correction

In a database application, a major and important component is processing input submitted from users to update its databases and deliver information through printing reports or displaying on interactive media; these are referred to as the external effects raised by the application. It should come to no surprise that people make errors in the use of computer systems [120, 177]. User input errors, however, can be serious which might lead to severe consequences after the execution of the system such as financial uncertainty, disruption to communication and corporate instability.

Currently, input validation [88] is the most popular means to enforce the accuracy of the user inputs to a system. Unfortunately, many user input errors can only be detected after the completion of execution. Take for example the entering of an extra valid product in a customer order, there is no means to detect and reject the valid extra item if it was submitted by mistake.

Error correction may seem an important supplementary safety goal since total error prevention is difficult (if not possible) to achieve. **Input error correction** refers to a bundle of system functionalities which allow users to correct erroneous effects raised due to input errors. The provision of input error correction is extremely important in any database application since input errors are unavoidable. Consequently, thorough testing and maintenance of input error correction become essential in ensuring the reliability of a system.

However, recovery of the input error correction feature is a complex, time consuming and error-prone task as there are many possible input errors during user interaction with the system; each input error, in turn, might result in several erroneous effects.

To test the provision of input error correction for a type of input error, test cases must cover the relationship between an input error and its correction feature. Test cases must also cover both code and specification characteristics. Unfortunately, most of the existing techniques are either solely code-based or specification-based through adopting structural or functional testing strategies, respectively.

Finally, maintaining input error correction for a database application requires a deep understanding on how the feature is implemented in a system. Unfortunately, most of the systems have a large number of components containing hundred thousand lines of code. Therefore, it is not obvious which component implements a given feature. In addition, the nature of software

development implies that many new or modified input error correction features are only recognized during maintenance. At this stage, the maintainers are facing the pressure to modify the system to include new feature as quick as possible without introducing any adverse impact to the existing code.

1.1.3 Automated extraction of database interactions

Nowadays, web applications have become fundamental parts, tightly integrated into many *e*-systems. A large number of web applications utilize a Database Management System (DBMS) and one or more databases. A program in a web application normally interacts with databases through statements which execute the SQL data manipulation language (DML) operations such as select, insert, update or delete. For web database applications, it is clear that database interactions are among the most essential functional features. As such, information on database interactions is useful in testing, understanding and maintenance of web applications.

Current industrial approaches to understanding and maintenance of the database interaction features include using tools scanning through the code, consulting documentations or asking system experts. This task is tedious and should be automated to increase the productivity and reliability of the maintenance process of web applications.

The difficulties in extracting database interactions from source code mainly result from the dynamic nature of web applications which are caused by statements generated only at run-time and dependent on user input. As such, the database interactions of a statement or a program vary from execution to execution of the web application. This problem, however, can neither be solved completely by static nor dynamic program analysis.

1.2 Objectives

The overall objective of this thesis is to explore the application of program analysis techniques augmented with empirical properties in software analysis, testing and maintenance. More specifically, the objectives of this thesis are as follows:

1. To propose a set of commonly seen patterns of infeasible program paths. Based on these patterns, we further aim to establish an approach to automatically detect a large number of infeasible program paths.
2. To formulate a theory for inferring the correctness of the input error correction feature implemented by a database application. The theory

will then be applied to recover, test and aid in the maintenance of this feature.

3. To investigate the combination of symbolic evaluation and empirical properties to solve the problem of database interaction extraction for web application. We aim to develop a systematic approach to infer all the possible database interactions from source code with high accuracy.

1.3 Major contributions of the thesis

The major contributions of this thesis are:

1. **A novel approach for infeasible path detection:** Although it is impossible to solve the general problem of identifying all infeasible paths, we have discovered that many infeasible paths exhibit some common properties. As a result, we propose in this thesis the key properties of infeasible paths and four code patterns that can cause these infeasible paths including conflicting-decision pattern, mutually-exclusive-decision pattern, check-then-do pattern and looping-by-flag pattern. We provide significant statistical evidences to support the validity of each property. Based on these properties, we further establish a static approach to detect infeasible program paths in the four code patterns.

We also analyze, demonstrate and evaluate the precision of the proposed approach through an experiment. A preliminary version of this work has been published in [153]. The approach presented in this thesis has been published in [151].

2. **A novel approach for recovery, testing and maintenance of input error correction:** We have formulated a theory, which integrates both invariant and empirical properties of input error correction, for inferring the implementation of this feature from source code. Based on this theory, we propose an approach to automatically recover the provision of input error correction in database applications.

The recovered information can be used to form a code-based equivalence partition with respect to input error correction. As such, we further introduce the Black&White approach which reconciles the code-based equivalence partition with a specification-based equivalence partition; thus, test cases generated from the reconciled implementation cover both code and specification characteristics.

We also make use of the recovered input error correction to aid software engineers in maintaining this feature. We present the effect-oriented

decomposition slicing technique to decompose a program with respect to the input error correction feature implemented in the program. As a result, changes can be made to this feature without introducing ripple effects.

We statistically validate our theory. We further conduct two case studies: one compares the performance of the Black&White approach with two traditional approaches in term of fault detection capability; one investigates the usefulness of the recovered information in program understanding and maintenance. This work has been published in [152, 150].

3. **A novel approach for database interaction extraction in web application:** We have presented an approach using symbolic execution and empirical inference rules to deduce all the possible database interactions automatically. The basic idea is that for each path which contains database interaction points, we apply symbolic execution rules to derive a symbolic expression representing the path. Empirical inference rules are then used to deduce the possible types of interactions.

We improve the performance of symbolic execution by following three strategies: (1) demand-driven of symbolic execution of function call, (2) hybrid mix of symbolic and concrete execution and (3) pre-execution of frequently used functions. We also measure and evaluate the performance of the proposed approach in term of precision and recall. This work has been published in [154, 155].

1.4 Organization of the thesis

This thesis is organized as follows. Chapter 2 provides background on program analysis techniques, gives an overview of empirical methods and reviews the importance of empirical methods in software engineering, particularly program analysis. Chapter 3 starts introducing related works on infeasible path detections, input error correction recovery, testing and maintenance, and reverse engineering of web applications.

Chapter 4 presents an infrastructure to support our research in the use of traditional program analysis techniques augmented with empirical properties. The infrastructure is the integration of various existing tools in software engineering. We describe the architecture of the two components of the infrastructure including EmAnalyzer, which provides program analysis functions, and EmValidator, which supports the statistical validation of empirical properties.

Chapter 5, Chapter 6 and Chapter 7, each demonstrates the application of program analysis augmented with empirical properties to solve a specific

problem in software engineering. In each of these chapters, we present the statistical validation to verify the correctness of each of the proposed empirical properties. Moreover, we also describe a prototype tool to demonstrate the feasibility of the proposed solution that addresses the specific problem in the chapter.

Chapter 5 presents four common patterns of infeasible paths including conflicting-decision, mutually-exclusive-decision, check-then-do and looping-by-flag patterns. Based on these patterns, we further propose a static approach to detect infeasible paths in a program. We also conduct an experiment to evaluate the effectiveness of the proposed approach in detecting infeasible paths in a large number of programs.

In Chapter 6, we first present a theory for inferring input error correction. The theory is an integration of both invariant and empirical properties. We provide a classification of user input errors and how they can be represented through program paths. We also present a mechanism for deriving erroneous effects resulting from each type of input errors. Based on the theory, we further propose an approach to automatically recover input error correction implemented by a database application. This chapter also discusses and evaluates the usefulness of the recovered information in testing and maintenance of input error correction.

Chapter 7 discusses the difficulties in extracting database interactions from web applications and presents our empirical approach to this problem by using symbolic evaluation and inference rules. In this chapter, we provide a set of symbolic evaluation rules for symbolically executing program paths and a set of inference rules for deducing all the possible types of interactions from symbolic expressions. An experiment has been designed to measure the precision and recall of the proposed approach and to compare the performance of our approach with a conservative approach. The time complexity of the proposed approach is also discussed.

Chapter 8 reviews the contributions of this research and identifies promising areas for future work.

All the notation conventions used throughout this thesis is summarized in Appendix A.

Chapter 2

BACKGROUND

As this research explores the use of empirical properties in program analysis, this chapter reviews relevant concepts in program analysis and empirical methods. This chapter is divided into two main sections. Section 2.1 provides an overview of fundamental program analysis techniques including control flow analysis, control dependence analysis, data flow analysis and program slicing. In Section 2.2, we first review various empirical methods with focus on experiments. We then discuss the importance of experimentation in software engineering. Finally, we review the use of empirical methods in program analysis.

2.1 Traditional program analysis

Program analyses¹ are techniques that statically compute program properties at each program point by estimating an abstraction of the possible program states when the execution of the program reaches the program point.

An intraprocedural program analysis technique computes program properties by examining a single procedure while an interprocedural technique examines a set of procedures that interact with one another. Many software engineering tasks, such as coverage analysis [41, 179, 218], test data generation [153, 208, 228], slicing [102, 103, 175], dynamic execution profiling [24] and impact analysis [9, 159] require information about the control flow and data flow, control dependence and data dependence among statements in a program. For instance, during compilation of source code, it is helpful to identify dependencies among program statements so that various optimization code transformations can be made without affecting the functionality of the program. During debugging, it is useful to isolate all the code on which an error-producing statement depends for calculating its result in order to determine the source of the error more easily.

¹ To distinguish from *dynamic* program analyses that compute program properties based on program execution, the program analyses defined here are often referred to as *static* program analyses. We drop “static” in our terminology because we do not consider dynamic program analyses in this thesis.

In our research, we use information on control and data dependences to identify highly potential infeasible program paths, to recover input error correction implemented in a system and to extract all the database interactions from the source code of a web application. We also use program slicing to maintain input error correction in database applications and to facilitate the validation of empirical properties and other program understanding tasks. Therefore, this section provides background necessary to understand fundamental dependence analysis techniques and program slicing.

2.1.1 Control flow analysis

Control flow analysis [4] determines for each program statement n_s , those statements in the program that could follow n_s in some executions of the program. Control flow analysis is the fundamental aspect of program analysis. Faithful representation of potential flows among program statements is essential to the identification of which variable definitions can reach variable uses and which statements control the execution of other statements.

Control flow relationship among statements in a procedure P_T in a program P can be represented by a directed graph called **Control Flow Graph (CFG)** $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \text{'begin'}, \text{'end'})$ where $\mathcal{N}$ is a set of nodes, $\mathcal{E}$ is a set of edges, 'begin' is the unique entry node and 'end' is the unique exit node. Each node represents a statement in P_T . Each edge (n_s, n_t) represents the flow of control from statement n_s to statement n_t [189]. For a procedure with multiple exit points, it is always possible to introduce an artificial 'end' node and insert an edge from each real 'end' node to the artificial 'end' node.

In this thesis, we do not distinguish a node in a CFG and the statement it represents. Nodes in a CFG are represented by circles. **Predicate nodes**, from which two edges may originate, are presented by diamonds. The **predicate** at each predicate node will be evaluated during the program execution to determine which out-going edge of the predicate node will be traversed. There are two type of predicate nodes namely **predicate nodes of selection construct** (if statements) and **predicate nodes of iteration construct** (for, while statements). Each out-going edge of a predicate node is called a **branch**. Each branch in the CFG is also attached with a predicate, referred to as **branch predicate**, describing the conditions under which the branch will be traversed. Figure 2-1 gives one example procedure and its CFG.

A **path** $p_k = (n_i, \dots, n_j)$ in a CFG represents one way to traverse the CFG from arbitrary node n_i to node n_j . A **complete path** c_k [112] is a sequence of node in a CFG that starts at the 'begin' node and ends at the 'end' node. A complete path in a procedure represents one way that the procedure can be executed.

A program that consists of multiple procedures is often represented as an **interprocedural CFG (ICFG)**, which consists of a set of inter-connected CFGs, one for each procedure. However, in an ICFG, each procedure call statement in a procedure is represented with two nodes: a **call node** and a **return node**. A call node represents the point where program control transfers to the entry of the called procedure. A return node represents the point where program control transfers from the exit of the called procedure back. A **call edge** connects a call node in a procedure to the 'begin' of the called procedure. A **return edge** connects the 'end' node of the called procedure to the return node corresponding to the procedure call.

When a program is executed, its execution trace consists of the sequence of statements that have been executed. An execution trace of a program can be mapped to a path starting from the 'begin' node of the main procedure in the ICFG of the program. Such paths are called executable or **feasible** paths. To compute precise program information, a program analysis technique should consider only feasible paths and compute information by propagating the information along only these paths.

2.1.2 Control dependence analysis

Control dependence analysis [190] determines for each program statement, the predicates that control the execution of that statement. The most common use of control dependence in software engineering is in determining whether a change to the semantics of a program statement affects the execution of another program statement. Such information may be used, for example, to locate the cause of a software failure, evaluate the impact of modification or determine the parts of a program that should be retested in response to a modification.

Let n_x and n_y be nodes in the CFG of a procedure such that $n_x \neq n_y$. If n_x appears on every paths from the 'begin' node to n_y , then n_x **dominates** n_y . Domination is both reflexive and transitive. The **immediate dominator** of a node n_y is the closest dominator of n_y on any path from the 'begin' node to n_y . In a **dominator tree**, the children of a node n_x are all immediately dominated by n_x . The **dominance frontier** of a CFG node n_x is the set of all CFG nodes n_y such that n_x dominates the predecessor of n_y but does not dominate n_y . In the CFG in Figure 2-1, node 1 dominates all other nodes in the CFG because it appears on every paths from the 'begin' node to the 'end' nodes. Node 6 does not dominate node 7 because it does not appear on this path (begin, 1, 2, 3, 4, 5, 7).

Let n_x and n_y be two nodes in a CFG such that $n_x \neq n_y$. If n_x appears on every path from n_y to the 'end' node then n_x **post-dominates** n_y . Like the dominator relation, the post-dominator relation is reflexive and transitive. The **immediate**

post-dominator of n_y is the closest post-dominator of n_y on any path from n_y to the 'end' node. In a **post-dominator tree**, the children of a node n_x are all immediately post-dominated by n_x . In Figure 2-1, node 8 post-dominates all other nodes in the CFG because every path from the 'begin' node to the 'end' node must pass through node 8. Node 6 does not post-dominate node 5 because node 6 does not appear in the following path from node 5 to the 'end' node: (5, 7, 3, 8, end).

```

begin
1.  total = 0
2.  value = read_input()
3.  while (value != null)
    {
4.      total = total + value
5.      if (total mod 100 == 0)
6.          print(total)
7.      value = read_input()
    }
8.  print (total)
end

```

(a) Pseudocode

(b) CFG

Figure 2-1. An example procedure

A CFG node n_y is **control-dependent** [63] on a CFG node n_x if both of the following hold:

1. There exists a path p_k from n_x to n_y such that n_y post-dominates every node after n_x on p_k
2. n_y does not post-dominate n_x

Informally, n_y is control-dependent on n_x if, in the CFG, there are two edges out of n_x such that following one edge causes node n_y to be reached definitely, whereas following the other edge may cause that node not to be reached [48]. According to this definition, in Figure 2-1, nodes 4, 5, 6, 7 are control-dependent on node 3. However, node 8 does not control-dependent on node 3 because node 8 post-dominates node 3.

Many algorithms have been proposed for computing control dependence at the intraprocedural level [18, 47, 48, 63, 189]. In intraprocedural control

dependence analysis, each procedure is treated in isolation by ignoring the transfer of control due to function calls and returns.

A popular method to perform intraprocedural control dependence analysis is proposed by Ferrante et al. [63]. The approach starts by augmenting the CFG with a special predicate node ‘entry’ that has one edge labeled ‘True’ going to the ‘begin’ node and another edge labeled ‘False’ going to the ‘end’ node. The next step is to compute the post-dominator tree by computing the dominator tree in the reverse CFG.

The **reverse CFG** [63] of a procedure has the same nodes as the CFG of the procedure but it has an edge (n_y, n_x) for each edge (n_x, n_y) in the CFG. The roles of ‘begin’ and ‘end’ nodes are also reversed. Ferrante et al. [63] has proved that the post-dominator relation on CFG is the dominator relation on reverse CFG. Dominator in the reverse graph can be computed quickly by using the algorithm proposed by Lengauer and Tarjan [134]. Once the dominator relation has been computed, the dominance frontier of each node can also be computed in linear time [47]. Cytron et al. [47, 48] has proved that a CFG node n_y is control dependent on a CFG node n_x if n_x is in the dominance frontier of n_y in the reverse CFG. As such, control dependence can be derived from the dominance frontier.

To function effectively on whole programs, however, program analysis techniques must account for interprocedural dependences which can be computed only by analyzing the interactions among procedures. Various definitions of, and methods for computing and utilizing interprocedural control dependences have been presented [84, 137, 189, 190].

Sinha et al. [190] propose two approaches to computing interprocedural control dependences. In the first approach, interprocedural control dependence is computed by applying an existing intraprocedural control dependence algorithm to an interprocedural inline flow graph (IIFG) of a program. Unlike an ICFG, at each call site, a distinct copy of the CFG of the called method is inline. Thus, an IIFG can be exponential in size of a program and is infinite for recursive programs. This approach distinguishes each context in which a procedure can be called and computes *context-based* interprocedural control dependences. However, this method may not be practical and unordinary expensive.

The second approach ignores the context-based distinction and computes *statement-based* control dependences: control dependences that exist in at least one context of execution of a statement. However, statement-based control dependences are not as precise as context-based control dependences because

the computation summarizes the control dependences that exist in different contexts.

2.1.3 Data flow analysis

Data flow analysis [5, 65, 85] is the process of identifying the potential for a value in one statement to affect the computation of another statement. Data flow information is used in activities such as program slicing [81], data flow testing [57, 65, 111, 211] and compiler optimization [142, 162]. Given a definition of a data item in a program, one would be interested in the uses which might be affected by the particular definition. The inverse is also true: for a given use, the definitions of data items which can potentially supply values to it are of interest. An other useful information is: given a program statement, what data definition are “live” at that statement, that is, what data definition given before this point are used after this point. All these kinds of information can be obtained through data flow analysis.

A data **definition** is an expression or that part of an expression which modifies a data item. A data **use** is an expression or that part of an expression which references a data item without modifying it. A data definition potentially **affects** a use if the data items are the same and the result of the definition is available to the use. A **definition-use pair (du-pair)** with respect to a variable v_k is a pair (n_d, n_u) , where n_d is a statement that defines the variable v_k and n_u is a statement that uses v_k , and there is a path p_x in the program from n_d to n_u along which v_k not redefined. Path p_x is referred to as a **definition-clear (def-clear) path** with respect to variable v_k . For example, in Figure 2-1, the path (7, 3) is a def-clear path with respect to variable value. However, the path (7, 3, 4, 5, 7, 3) is not a def-clear path with respect to variable value because value is redefined in the second execution of node 7.

A data flow relation can be represented as a **data dependence graph** in which nodes represent program statements and edges represent the data dependence between program statements. As such, if (n_d, n_u) is a du-pair then a data dependence edge exists between n_d and n_u .

Two basic data flow problems are the “reaching definition” and “live variable”. The goal of the reaching definition problem is to identify all the locations of definitions for all variables that may reach a particular statement in a program. The solution to a reaching definition problem for a particular statement n_u is a set of tuples where each tuple (v_k, n_d) represents the fact that statement n_d defines variable v_k and there is a def-clear path from n_d to n_u with respect to v_k . Solution to the live variable problem provides the set of similar tuples for all variable values that may be used after a given statement.

Many approaches have been proposed for the intraprocedural dataflow analysis [2, 5]. Intraprocedural data flow analysis considers the flow of data within a procedure, while assuming some approximations about definitions and about uses of reference parameters and global variables at call sites. Basically, a data-flow problem can be formulated as a set of equations that compute data flow facts and those data flow facts are computed and propagated iteratively throughout the program, using CFG representation [2].

Interprocedural data flow analysis [11, 58, 85, 111] computes information about the flow of data across procedure boundaries caused by referenced parameters and global variables. Some techniques have been proposed to provide summary data flow information for determining the local effects of called procedures at call site [10, 43]. However, these techniques do not provide information about the locations of interprocedural definitions and uses in other procedures in the program. Allen et al. [3] propose an interprocedural data flow analysis technique for non-recursive procedures. This technique processes procedures in reverse invocation order to incorporate the abstracted information about called procedures at call sites to obtain the local reaching information; that is, the technique requires that a procedure be processed only after those that it calls have been processed. This reverse order imposes a penalty when changes are made in a procedure, for it causes the reanalysis of those procedures directly or indirectly dependent on the changed procedure. Myers [147] introduces super graph or in-line substitution to compute the interprocedural data flow relation. However, the use of in-line substitution is impractical for large programs. In addition, recursive procedures cannot be analyzed. A class of demand-driven approaches [57, 56] [94, 104] has been proposed to reduce the time/space overhead of conventional approaches. Reps et al. [172] have shown that a large class of interprocedural, finite, distributive data flow analysis problems can be solved precisely in polynomial time by transforming them into a special kind of graph reachability problem.

2.1.4 Program slicing

Program slicing [203] is a well-known method which can be used to extract from a program those statements which are relevant to a particular computation. Program slicing was first introduced by Weiser [209], who discovers that the mental process made by programmers when debugging their code was slicing and tried to formally define this process and its output. Since then, program slicing has grown as a field and an amazing number of papers have been published that present different forms of program slicing, algorithms to compute them and applications to software engineering. Program slicing has been used extensively in many software engineering fields such as debugging,

code understanding, testing, reverse engineering, software maintenance, reuse and metrics [14, 21, 68, 111, 128, 132].

A **slice** $S(v_s, n_s)$ of program P on variable v_s at statement n_s yields the portions of the program that contribute to the value of v_s just before statement n_s is executed; (v_s, n_s) is called a **slicing criterion**.

The program slice concept introduced by Weiser is often referred to as *executable backwards static slice*. “Executable” because the slice is required to be an executable program. “Backwards” because the direction edges are traversed when the slice is computed using a dependence graph. Finally, “static” because they are computed as the solution to a static analysis problem without considering runtime information. Weiser has shown that an approximation of a slice can be found by computing the least solution to a set of data flow equations relating a CFG node to the variables which are relevant at the node with respect to the slicing criterion. This can be done in two steps: (1) requisite data flow information is computed and then (2) this information is used to extract the slice. The first step identifies data flow information which basically is the set of relevant variables at each node. For the slice with respect to the criterion (v_s, n_s) , the relevant set for each node contains the variables whose values (transitively) affect the computation of v_s at n_s . The second identifies the statements of the slice. These include all statements that assign to a variable relevant at n_s and the slice taken with respect to any predicate node that directly controls n_s ’s execution. Table 2-1 gives one example on the computation of relevant sets for the slicing criterion (8, a). In this example, the slice with respect to (8, a) includes lines 7, 6, 2 and 1.

Table 2-1. Relevant sets for (8, a)

N_s	Statement	$\text{refs}(n_s)$	$\text{defs}(n_s)$	$\text{relevant}(n_s)$
1	$b = 1$	-	b	-
2	$c = 2$	-	c	b
3	$d = 3$	-	d	b, c
4	$a = d$	d	a	b, c
5	$d = b + d$	b, d	d	b, c
6	$b = b + 1$	b	b	b, c
7	$a = b + c$	b, c	a	b, c
8	print a	a	-	a

Legend: $\text{refs}(n_s)$ – variables referenced at n_s
 $\text{defs}(n_s)$ – variables defined at n_s
 $\text{relevant}(n_s)$ – variables relevant at n_s

Ottenstein et al. [161] later note that static slicing can be restated in terms of a reachability problem in a **Program Dependence Graph** (PDG). A PDG is a

directed graph with vertices corresponding to statements and edges corresponding to data and control dependences. As such, backward slices can be efficiently computed using the PDG by traversing the dependence edges backwards. Horwitz et al [93] extends the PDG based algorithm to compute interprocedural slices on the **System Dependence Graph (SDG)**. The authors demonstrate that their algorithm is more accurate than the algorithm introduced by Weiser because they consider procedure calling contexts. Various algorithms have been proposed for interprocedural slicing [19, 67, 116, 148].

Finally, Korel and Laski [123, 125, 126] introduce the notion of *dynamic slicing*: a slice computed for a particular fixed input. Dynamic slicing uses dynamic analysis to identify all and only the statements that affect the variables of interest in a particular execution of the program. The runtime information makes dynamic slices smaller than static slices, thus allowing an easier localization of program bugs. Another advantage of dynamic slicing is the runtime handling of pointers and arrays. However, dynamic slicing techniques limit their applicability to only a particular input. Korel and Laski's algorithm is similar to Weiser's in that it uses CFG as an intermediate representation. The algorithm is based on dataflow equations to iteratively compute a dynamic slice. To ensure that the resultant slice is executable, the algorithm requires that if any occurrence of a statement within a loop in the execution trace is included in the slice, then all the other occurrences of that statement be included in the slice.

2.1.5 Challenges

Program analysis is undeniably useful which provide numerous benefits to a large variety of tasks in the software engineering process including code optimization, debugging, software testing, maintenance and evolution. However, many program analysis techniques are rarely applied in practice for their inability to scale to large system [83]. For example, classic program analyses such as point-to analysis, typically run out of memory space before running out of time [20] due to its computational complexity. For another example, data extraction techniques to recover software properties from source code also require significant computing resources.

Another difficulty encountered when applying program analysis techniques to real-world program is to efficiently compute sufficiently precise information for programs in the presence of complex language construct such as pointer. Analyzing programs that use pointer can be very challenging because aliasing can occur in such programs [204, 207]. Aliasing describes a situation in which a data location in memory can be accessed through different symbolic names in the program. Precisely computing alias information is undecidable in the

presence of general purpose pointers, dynamically allocated memory and loop statements. Many researchers have proposed alias analysis algorithms that compute approximate analysis information for programs that use pointers. Many of these algorithms compute highly accurate alias information [37, 61, 129]. However, empirical studies suggest that these algorithms may be too expensive for analyzing large programs. Many of these algorithms may compute very imprecise alias information with a lower cost [197]. As such, to support efficient and accurate analysis of programs with pointers, new alias analysis algorithms that provide a better trade-off between efficiency and precision must be developed.

The treatment of industrial programming languages is also a challenge. Program analysis techniques must apply to the languages that are widely used today or are expected in the future. It seems that most of these languages are object-oriented or distributed. It is quite unfortunate that very few contributions in the field of program analysis have tackled the specific features of these languages.

2.2 Empirical approaches to program analysis

This section first looks at various empirical methods and then discusses the application of traditional experimentation and empirical knowledge into software engineering in general and program analysis in particular.

2.2.1 Empirical methods

Empirical methods play an important role in testing the validity of research hypotheses. There is an iterative nature of the method in the construction of hypotheses, deduction of possible consequences of the hypotheses and control of the predicted consequences through observations. There are various ways of carrying out empirical studies, which are suitable for different purposes. Thus prior to selecting an empirical approach, one needs to consider what is the purpose of the study, what observations will be made and what will the observations be used for.

2.2.1.1 Experiments

The purpose of an experiment is to test a hypothesis, or to make observations that are otherwise hard to obtain. In our research, we use experiments for the purpose of validating empirical properties of the object under analysis. The basic principles behind an experiment are illustrated in **Figure 2-2**, which is adapted from the one introduced by Wohlin et al. [214].

Figure 2-2. Experiment principles (adapted from [214])

The starting point is that we have an idea of a cause and effect relationship. For example, we believe that if two predicate nodes which use the same set of variables and these variables are not redefined in any path between the two nodes, then they will lead to conflicting branches which in turn cause infeasible paths. We then formulate a theory which consists of a set of hypotheses. A **hypothesis** is a general statement which is tested by deduction, i.e., a prediction is made which is testable in the experimental environment. For example, a hypothesis can state assumptions about the observable behavior of infeasible paths in a program or input error correction feature in a database application. To test a hypothesis, we need to design an experiment. In the design of the experiment, we have a number of **treatments** (values that the studied variable can take) over which we have control [214]. When the experiment is performed, observations are collected and analyzed by means of statistical techniques in order to test whether the predictions were actually true. If so the experiment can be taken as evidence in favor of the hypothesis.

When conducting a formal experiment, we want to study the **outcome** when we vary some of the input variables. Wohlin et al. [214] defines two kinds of variables in a experiment including independent and dependent variables. An **independent variable** can be manipulated and controlled during the experimental process. A **dependent variable** is the one that we want to study to see the effect of the changes in the independent variables.

The first step toward analyzing data collected during the experimental process is to understand the data by using descriptive statistics. These provide a visualization of the data which can be used to interpret the data informally. After this step, the data set should be reduced, either by removing data points or reducing number of variables if some of the variables provide the same information. The last step is hypothesis testing, where a particular type of test (for e.g. binomial test) is chosen based on the input data, the outcome and the type of results that we are looking for. One important part of hypothesis testing is interpretation in which one need to determine whether the hypothesis was accepted or rejected. This forms the basis for decision-making and conclusions concerning how to use the results from the experiment, which include motivation for further studies.

In this thesis, **binomial tests** [77] are conducted to validate the empirical properties. Binomial test is used when there are two possible outcomes (“success” and “fail”) for each case in the sample. In our approach, an empirical property either holds (“success”) or does not hold (“fail”) for each case in the sample for testing the property. For each empirical property $\mathcal{R}$, the alternate hypothesis states what the probability of $\mathcal{R}$ holds for all the cases is equal or more than p (for e.g. $p = 99\%$) and the null hypothesis states what the probability of $\mathcal{R}$ holds for all the cases is less than p . Binomial test can be used to reject the null hypothesis. To conduct the test, we first compute the z-score as follows:

$$z = \frac{X/n - p}{\sqrt{(p(1-p)/n)}}$$

where n is the sample size for testing the property, X is the number of cases that support the alternate hypothesis. We then choose a significant level, which is also referred to as Type I error. The significance level is a criterion for rejecting the null hypothesis. It is defined as the probability of making a decision to reject the null hypothesis when the null hypothesis is actually true. Depending on the value of p and the significant level chosen, we can derive the minimum z-score value z_0 . If the computed z-score value is greater than z_0 , we reject the null hypothesis.

We believe that a significance level between 0.001 and 0.005 and a p value from 0.95 are considered acceptable. However, in hypothesis testing, it is always desirable if one could maximize the p value and minimize the Type I error. This can only be done by increasing the sample size and conducting very large scale experiments. Unfortunately, large scale empirical studies face serious obstacle in term of time and resources. As such, researchers always have to compromise between the significance of the empirical results and cost, time required to conduct the experiment. For this reason, in all statistical

validation presented in this thesis, 95% is set as the lowest p value. Whenever time and resource permitted, a higher p value is chosen.

A **controlled experiment** is an experiment in which the researcher can manipulate the conditions under which the observations are made, i.e., that the context of the events are controlled. In the context of software engineering, as most the experiments are highly controlled and hence often referred to as controlled experiment. Controlled experiment is the only empirical approach which is characterized by this property. The rationale for this control is to limit the number of variables in the study. The relation between a controlled experiment and the real world is that: When a piece of reality is investigated, the presumably relevant variables are identified and reproduced and controlled in the artificial environment. However, some variables present in the real environment have to be left out. After all, the simplification through reduction of the number of variables is a major benefit of an experiment. The question is whether the variables left out are important with respect to the phenomenon observed. Another problem of restricted environment is that new variables may be added. If these variables have an influence on the phenomenon observed, they may compromise the validity of the experiment.

Threats to the validity [42] refers to various factors and conditions concerning the experiment and the way it is carried out and interpreted which may compromise the results. Threats to validity are relevant in any kind of empirical study, not only experiments. Internal validity means that changes in the dependent variables can be safely attributed to changes in the independent variables. External validity means that the study's results generalize to settings outside the study.

2.2.1.2 Surveys

A survey is a retrospective study of a situation that investigates relationships and outcomes. It is useful for studying a large number of variables using a large sample size and rigorous statistical analysis. This empirical method is especially well-suited for answering questions about what, how much, and how many as well as questions about how any why [167].

The primary means of gathering qualitative or quantitative data are interviews or questionnaires. These are done through taking a sample which is representative from the population to be studied. This approach is very common in disciplines such as marketing, medicine, psychology and sociology where control of the independent and dependent variables is not possible or not desirable or when the phenomena of interest must be studied in their natural setting at a specific time.

2.2.1.3 Case studies

A case study is an observational study while an experiment is a controlled study [226]. Therefore, a case study investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evidence. As such, “*while an experiment deliberately divorces a phenomenon from its context and a survey’s ability to investigate the context is limited, the case study aims deliberately at covering the contextual conditions*” [192]. However, Yin [224] notes that since the phenomena are studied in their real context, the number of variables in the study will be huge. Therefore a case study must deal with this situation, in terms of data collection as well as data analysis.

In software engineering, case studies are particularly important for the industrial evaluation of methods and tools.

2.2.2 The role of experiments in software engineering

An empirical study is just a test that compares what we believe to what we observe. Nevertheless, such tests, when wisely constructed and executed, help in understanding how and why things work [163]. Empirical verification of knowledge is one of the foundations for developing any discipline. This approach has been used in many fields such as physics, medicine and manufacturing. Software engineering is empirical in nature. For example, software development has been characterized by a serious need of empirical facts tested against reality that provide evidence of the advantages and disadvantages of using different methods, techniques and tools to build software systems [110]. However, as far as software engineering is concerned, the empirically verified knowledge is not only sparse but also not very widely disseminated among developers and researchers [109]. Various surveys in the area of empirical software engineering research have been conducted which conclude that the majority of published articles in computer science and software engineering provide little or no empirical validation and the proportion of controlled experiments is particularly low.

Tichy et al. [201] conduct a quantitative study surveying over 400 research articles and look for the application of experimental methods. Interestingly, they find that 40% of computer science papers about design and modeling completely exclude experimentation, in software engineering papers; the number is even higher at 50%. The computer science papers contain a significantly lower amount of purely empirical papers than in areas like Neural

Computing or Optical Engineering. Hypothesis testing articles are rare in all samples (1%).

Zelkowitz and Wallace [226] conduct a similar study like Tichy and analyze 612 software engineering papers and 137 papers from other sciences including physics, management science and behavior theory. The results obtained are similar to Tichy's in which about one-third of the software engineering articles have no experimental validation at all. However, the percentage was decreasing over the years (1985: 36.4%, 1990: 29.2%, and 1995: 19.4%). About one-third of the articles contain only assertions as experimental validation. However, assertions are preliminary form of experimental validation and are potentially biased. This means that researchers are starting to realize the need for experimental validation, yet still do not do it with strong methods.

Sjoberg et al. [193] conduct a survey on the use of controlled experiments in software engineering. Analysis is done on 5453 scientific articles published in 12 leading software engineering journals and conferences from 1993 to 2002. Among these articles, only 103 articles (1.9%) report the use of 113 controlled experiments, given that controlled experiment is the classical scientific method for identifying cause-effect relationship. One reason may be the large effort and resources needed to run well-designed experiments.

Since the work of Basili et al. [12, 13], there has been an increased focus on the need for, and approaches to, applying empirical methods in software engineering research. Basili et al. [12] discuss the developing bodies of knowledge via an iterative model building, prediction, hypothesis testing, observation and analysis. The paper argues that common families of studies contribute to important and relevant hypotheses that may not be suggested by individual experiments. Therefore, a framework for organizing common families of studies is necessary. Moreover, such a framework facilitates building knowledge in an incremental manner through the replication of experiments within families of studies. The paper thus emphasizes the importance of replicated experiments. Too many software engineering experiments stay isolated which do not lead to a larger body of knowledge.

Perry et al. [163] believe that the problem with empirical research is not in the details, but rather in the goals of those studies. The authors then identify the most important components of an empirical study including research context, hypotheses, experimental design, and threats to validity, data analysis and conclusions. The authors also emphasize on the importance of replicated experiments. They highlight that it is often challenging to design studies for complex issues and difficult questions considering the effort and cost needed for empirical studies. In this case, it is necessary to focus on smaller problems

and to create multiples studies, which results possibly can be combined to answer deeper questions. The credibility of empirical studies will be improved drastically if other researchers are provided with enough information to reproduce the results.

Kitchenham et al. [117] present several guidelines for conducting empirical studies in software engineering which hopefully will improve the quality and quantity of performing and evaluating empirical research in software engineering. The paper provides the following explicit steps for the improvement of individual experiment, which are based on a review of guidelines for medical researchers:

1. **Experimental context:** Information about the industrial context in which the experiment is conducted, must be included. The research hypotheses have to be discussed and also how they have been derived.
2. **Experimental design:** The population under analysis and the sampling technique used for it must be described. It has to be documented what intervention have been conducted and what method has been used to reduce bias to determine the sample size.
3. **Conduct of the experiment and data collection:** The measures for the outcome of a study must be documented in sufficient detail.
4. **Analysis:** Procedures used to control for multiple testing should be specified. In addition, the data must not violate the assumptions of the tests used on them.
5. **Presentation of results:** A reference should be provided for all statistical procedures used and the statistical package used should be reported. The magnitude of effects and the confidence limits of quantitative results should be represented as well as confidence levels.
6. **Interpretation of results:** Researchers should differentiate between statistical significance and practical importance. The limitations of the study should be discussed.

2.2.3 The use of empirical methods in program analysis

Despite the general trend of applying empirical methods in software engineering, there has been little consideration given to the use of traditional experimentation in program analysis. Experimentation is a potential avenue for addressing many program analysis questions in which more traditional analyses have not succeeded.

According to Zeller [227], program analysis can be classified into the following four classes defined by the number of program executions considered:

1. **Deductive program analysis** generates findings *without executing* the program (static analysis).
2. **Observational program analysis** generates findings from a *single execution* of the program.
3. **Inductive program analysis** generates findings from *multiple executions* of the program.
4. **Experimental program analysis** generates findings from *multiple executions* of the program where the executions are *controlled* by the tool.

The subsume relationship of these four classes form a hierarchy as shown in Figure 2-3. In deductive program analysis, deduction is used for reasoning from the program code to concrete runs. Deduction does not require knowledge of the concrete. Therefore, the program in question does not need to be executed – the program analysis is static.

Figure 2-3. A hierarchy for program analysis (from Zeller [227])

Observational program analysis, inductive program analysis and experimental program analysis are runtime analysis techniques because at least one actual program run is required to perform the analysis. Observational program analysis allows the programmer to inspect arbitrary aspect of a single program run, whereas inductive program analysis is used to summarize multiples program runs – e.g. a test suite or random testing – to some abstraction that holds for all considered program runs. Typical program analysis tools that use induction are coverage tools that summarize the statement and branch coverage of multiple runs.

The most advanced program analysis technique is experimental program analysis. All other program analysis techniques are not able to prove that some aspect of a program is actually the cause for a specific behavior. Experimental program analysis uses multiple program runs to search for the actual cause and required a series of experiments that are controlled by the tool.

Zeller does not precisely define experimental program analysis. Moreover, when considering the principles of experimentation, there are several drawbacks in Zeller's view about experimental program analysis. Zeller suggests that the experimental approach can be used to "prove actual causality". However, experimentation can be used not just to establish causality but, more broadly, to establish relationships and characterize a population. Further, experiment in general does not provide "proofs"; rather, it provides probabilistic answers. Zeller's definition of experimental program analysis misses several concepts that are integral to experimentation including the roles of population, sampling methods, selection of independent and dependent variables, experiment design and statistical analysis. Finally, experimental program analysis is not constrained to the traditional static or dynamic classification. It can overlap with both static and dynamic analysis techniques.

To fill in these gaps, Ruthruff et al. [180] attempt to formalize experimental program analysis (EPA). They define experimental program analysis as "*the evolving process of manipulating a program or factors related to its executing, under controlled conditions in order to characterize or explain the effect of one or more independent variables on an aspect of the program*".

According to this definition, EPA techniques manipulates either concrete representation of a program such as source code, or factors related to its execution such as input or program state, in order to learn about the effects of those manipulations on an aspect of the program under analysis. It is important to note that EPA is an "evolving process", which means that multiple experiments should be conducted in sequence with the design of later experiments changed by leveraging findings from previous experiments. In this way, EPA techniques are able to build a body of knowledge regarding the program under analysis. Ruthruff also provides the operational steps towards applying experimental methods into program analysis, which are presented in Table 2-2.

Table 2-2. Tasks in experimental program analysis (from Ruthruff et al. [180])

Guideline	Task Identifier	Task description
RECOGNITION AND STATEMENT OF THE PROBLEM	Formulation of research question	Questions about specific aspects of a program.
	Identification of population	The aspect of the program that the experiment will draw conclusion about
SELECTION OF INDEPENDENT AND DEPENDENT VARIABLES	Factors	The internal or external aspects of the program that could impact the effect of the measured manipulation.
	Independent variables	The factors that are manipulated in order to impact the program aspect of interest.
	Fixed variables	The factors that are set or assumed to be constant.
	Nuisance variables	Uncontrolled factors that can affect the measured observations on the program.
	Dependent variables	The constructs used to quantify the effect of treatments on the target program aspect
CHOICE OF EXPERIMENT	Treatments	The specific instantiations of levels from the independent variables that are tested
	Hypothesis statement	Statements about the effect of treatments on an aspect of the program
	Sample	A set of elements from the population
	Treatment assignment	Assignment of levels of independent variable to the sampled units.
PERFORMING THE EXPERIMENT	Experiment procedures	An automated process using algorithm to assign treatment to units in the sample and capturing observations measuring the isolated effects of those changes.
ANALYSIS AND INTERPRETATION OF DATA	Data analysis	Analyzing the observations to discover or assess the effect of the treatments on the aspect of the program of interest.
	Hypothesis testing	Using the analysis from the observation to confirm or reject the previously stated hypothesis regarding treatment effects
	Interim analysis	Making a decision regarding whether further tests are needed based on the results of the current and previous tests
CONCLUSIONS AND RECOMMENDATION	Final conclusions	The conclusions drawn from the application of experimental program analysis

This is a very general framework which follows the traditional experimentation procedure presented in Section 2.2.1.1. In terms of methodology to discover empirical properties, our research is most closely related to Ruthruff's work.

The experimental nature of this approach has been reflected in many of the existing approaches [135, 136, 138, 139]. Liu et al. [136] introduce a hypothesis testing-based method called SOBER which automatically localizes software faults without any prior knowledge of the program semantics. This model qualifies the fault-relevance of each predicate in a program according to a ranking algorithm that ranks predicates according to how abnormally each predicate evaluates in incorrect execution; that means, the more abnormal the evaluations, the more likely the predicate is fault-relevant. Marlevis et al. [139] propose a predictive metric for likely feasible program paths. The metric states that if a path p_x contains less predicates than a path p_y then p_x is more likely to be feasible than p_y . This metric is then statistically validated through hypothesis testing. χ^2 test is performed at 0.5% level of significance which shows that there must be some forms of dependency between a path's feasibility and the number of predicates involved.

2.3 Conclusion

Control/data flow analysis, control dependence analysis and program slicing are among the fundamental aspects of program analysis which are required by many software engineering tasks including software analysis, testing and maintenance. Control flow analysis determines for a program statement those statements which could follow that statement in some execution of the program. Control dependence analysis determines for each program statement the predicates that control the execution of that statement. Data flow analysis identifies the potential for a value in a statement to affect the computation of another statement. Finally, program slicing extracts from a program those statements which are relevant to a particular computation.

Empirical approaches, especially experimentation is central to any scientific process. However, the use of experimentation and empirical knowledge is not so common in software engineering, especially program analysis. Ruthruff et al. [180] are the first to formalized the experimental program analysis approach following the general principles of scientific experimentation. Our approach to discover empirical properties is quite similar to the experimental procedure presented by Ruthruff where multiple program runs are conducted to discover, learn and validate special properties of the program under analysis. Once we have a set of stable properties, we use these properties to design algorithms. These algorithms augment program analysis techniques with the discovered empirical properties to detect instances of the properties in the program so that valuable information can be recovered which can be used for various software engineering tasks.

Chapter 3

RELATED WORK

The goal of our research is to explore the application of program analysis techniques augmented with empirical properties to solve the following problems in software engineering: (1) Infeasible path detection, (2) automated recovery, testing and maintenance of input error correction, and (3) automated extraction of database interactions in web applications. Therefore, this chapter reviews relevant concepts and related research in the following areas

1. Infeasible path detection
2. Input error correction recovery, testing and maintenance:
 - Input error correction
 - Design recovery
 - Integrated functional and structural testing
 - Using slicing in software maintenance
3. Reverse engineering of web applications

Our work on input error correction recovery, testing and maintenance is related to several research fields. First of all, it focuses on *input error correction*, one of the most important functional features in any database application. Various techniques and approaches have been proposed in the area of human computer interaction regarding the design, implementation and different strategies for input error correction. Second, we propose an approach to recover the provision of input error correction from source code. This work is related to *design recovery* technique in the field of reverse engineering. Third, our work introduces a new approach to test input error correction which incorporates both code and implementation characteristics. Therefore, basically, the approach *integrates both functional and structural testing* for input error correction. As such, we provide a brief introduction on fundamental structural and functional testing strategies. We then review related work in integrated functional and structural testing. Finally, our work is based on program slicing and the recovered information on input error correction to aid the maintenance of the feature. Thus, in this chapter, we also review the use of slicing in software maintenance.

Our work on extracting database interactions for web applications, on one hand, it can be viewed as a traditional design recovery approach; on the other hand, it can also be viewed as a reverse engineering approach for web applications, which is quite different from traditional reverse engineering due to the dynamic nature of web applications. Therefore, in this chapter, we also provide an overview of research in the area of reverse engineering for web applications.

3.1 Infeasible path detection

A great majority of program paths are found to be infeasible, which in turn make static analysis overly conservative. As static analysis plays a central part in many software engineering activities [23, 64, 74], knowledge about infeasible program paths can be used to greatly improve the performance of these activities, especially structural testing and coverage analysis.

An **input variable** v_i of a program P is a variable which appears in an input statement or an input parameter. The **domain** $D(v_i)$ of the input variable v_i is the set of values that v_i can hold. The input domain $\mathcal{D}$ of the program P is the product

$$\mathcal{D} = D(v_1) \times D(v_2) \times \dots \times D(v_n)$$

where $D(v_i)$ is the domain of the input variable v_i . A point x_k in the n -dimensional input space $\mathcal{D}$, $x_k \in \mathcal{D}$, is referred to as an **input**. A path is **feasible** (or executable) if there exists an input $x_k \in \mathcal{D}$ for which the path is traversed during the program execution; otherwise, the path is **infeasible** (or unexecutable).

Hedly et al. [89] have discussed the cause and effect of infeasible paths in computer programs. According to them, about 12.5% of all the paths are infeasible which can be approximately classified into the following categories:

1. 52% of the infeasible paths are due solely to the number of times that loops must be executed.
2. 12.7% are due to loops which contain or associate with conditions on the loop variables.
3. 23.1% are due to testing a value immediately after it has been set.
4. 9.2% are due to consecutive conditions where the path taken according to the first condition controls the path through the second.

Bodik et al. [22, 23] also report that the main cause of infeasible paths is the correlation between some predicate nodes along the path. Two predicate nodes are correlated if along some paths, the outcome of the later can be implied from the outcome of the earlier [23].

```

procedure compute_grade(int mark, int studentID)
begin
1.   grade = ""
2.   if (mark ≤ 100) {
3.     if (mark < 50)
4.       grade = "Fail"
5.     else
6.       grade = "Pass"
7.   }
8.   if (grade != "")
9.     update the student with 'studentID' record in the database
end

```

Figure 3-1. The compute_grade procedure

Figure 3-2. CFG of the compute_grade procedure

Take for example predicate nodes 2 and 6 in the procedure compute_grade in Figure 3-1. They are correlated because the outcome of the predicate s_p : (grade != "") in statement 6 can be implied from the outcome of predicate s_q : (mark ≤ 100) in statement 2. More specifically, when s_p is true, s_q is true and when s_p is false, s_q is false.

In the presence of correlation, some paths are not executable [23]. In the previous example, nodes 2 and 6 are correlated. As such, the following complete path through the CFG in Figure 3-2 is infeasible:

$$c_x = (\text{begin}, 1, 2, 3, 4, 6, \text{end})$$

If following path c_x , when control reaches node 6, the outcome of the predicate s_q is always true because the outcome of predicate s_p is true (the 'true' out-coming edge of node 4 is not followed). Path c_x takes the 'false' out-coming edge of 6. In this case, there is no input on which c_x is traversed; thus, according to the definition, path c_x is infeasible.

Improving the precision of static analysis by detecting infeasible program paths has been tackled by many researchers. In general, detecting infeasible paths is theoretically unsolvable. Most of the existing techniques can be roughly classified into two types: *precise methods* [23, 74, 229] and *approximated methods* [27, 64, 138, 139, 222]. Precise methods generally rely on symbolic evaluation or constraint solving techniques, thus trading expensive computation for sharper analytical results. Moreover, due to the limitation of symbolic evaluation in handling pointers, arrays and function calls, only a small

percentage of infeasible paths can be detected. Approximated methods are based on heuristic rules which are discovered through empirical study to detect infeasible paths. These approaches are fast and effective. However, to the best of our knowledge, none of the existing heuristics is capable of detecting a large number of infeasible paths.

3.1.1 Precise methods

Clarke [39] presents an approach to infeasible path detection for test data generation based on symbolic evaluation. Each path is symbolically evaluated to generate the constraints representing the path. If a constraint is found to be inconsistent with some others, then the path is shown to be infeasible.

Goldberg et al. [74] determine the feasibility of a path by symbolically executing the path to derive a symbolic expression, which is then solved by a theorem prover. If there are input values which drive the execution of the program down to the path then it is feasible. However, the approach can only be applied to a subset of Ada language. Moreover, the performance of the method is limited by the type of symbolic expressions that the theorem prover can solve. The use of symbolic evaluation imposes restriction on the treatments of loops and function calls.

Bodik et al. [23] propose an approach to detect infeasible paths in a program using branch correlation to improve the accuracy of traditional data flow analysis. The work is motivated by the experimental results from the authors' earlier work [22], which shows that 9 to 40% of predicate nodes in large programs exhibit correlation which can be detected statically during compile time. The proposed approach is based on four main sources of branch correlation, as follow:

1. constant assignment
2. a prior conditional branch may subsume the branch predicate s_p
3. type conversion
4. pointer dereferencing

For each branch predicate s_p , the approach tries to solve the outcome of the predicate using backwards symbolic substitution. If the predicate cannot be solved, then it is not sure whether the predicate is correlated with any other branch predicate during run time. However, if the outcome of the predicate can be identified to true/false at compile time then the paths which start at the node where the outcome was resolved and end at the false/true branch of the analyzed predicate, respectively, correspond to infeasible paths. However, it is reported by Bodik et al. [23] that only about 45% of the predicate nodes are analyzable due to some limitations of symbolic evaluation and the compiler.

Moreover, only about 13% of the analyzable predicates show some correlations during compile time.

Zhang et al. [229] propose a constraint solver to detect the feasibility of program paths. Basically, the constraint solver is an extension of a Boolean satisfiability checker which accepts constraint expressions in a natural form involving variable of primitive data types such as integers, Booleans, real numbers and fixed size arrays. The main drawback of this approach is the complexity of the constraint solving algorithm. As such, the approach cannot scale well for large size programs. Moreover, the constraint solver is limited to only primitive data types.

Kountoris et al. [127] present a method to annotate feasible path analysis based on control flow graph. The authors define Conditional Control Flow Graph (CCFG) which is a CFG with additional information called clocks and control hierarchy. A *clock* annotates an edge, e_k , such that $\text{clock}(e_k)$ denotes the condition under which the execution flow passes through e_k . The control hierarchy is the complete set of clocks. Therefore, each path has characteristics that can be defined by its set of clocks. The control hierarchy is quite similar to the path constraints. After constructing the CCFG for a program, a path through the CCFG is feasible if the Boolean product of its edge clocks is not false. This approach assumes that all predicates can be symbolically evaluated. However, in practice, many predicates cannot be evaluated because inputs are taken at runtime. Moreover, analyzing predicates that include indirect memory references add to the complexity of the algorithm.

Jasper [99] describes techniques used by Test Specification and Determination Tool (TSDT), a prototype for analysis and testing of critical applications written in Ada. This tool determines feasibility of program paths by using the Kestrel Institute Theorem Prover [114], which mathematically determines feasibility by proof or disproof. It uses hierarchical deduction, propagates backwards to determine the outcome of predicates. The theorem prover also looks forward to find consequential outcomes. To facilitate the proofs, it maintains a knowledge base, which stores deduced results, as well as axioms and inference rules.

3.1.2 Approximate methods

Woodward et al. [217] highlight the difficulties in performing a path testing strategy for computer programs, one of those is the existence of infeasible paths. The paper classifies infeasible paths into two types:

1. Those which it would be possible to detect by more sophisticated static analysis. For example, if two predicate nodes are known to use the same variable, which has not been redefined in between the two uses,

then the information concerning the infeasibility of certain paths could be deduced.

2. Those which it would only be possible to determine by consideration of values of control variables. Hence symbolic execution and algebraic manipulation or something similar is implied.

The authors also discuss an informal method of reducing infeasible paths. If some infeasible feature of a sub-path is discovered, the information is stored to prevent the future enumeration of longer paths which also contain that feature. However, since infeasible paths complicate path testing, the authors suggest tackling the problem of reducing a large number of infeasible paths early from the software design and production stages by applying the following techniques:

1. Language design
2. Programming standards
3. Program transformation

A metric to predict the feasibility of a program paths is introduced by Malevris [138, 139]. The metric is based on the observation that the greater number of predicates contained within a path, the greater the probability of it to be infeasible. χ^2 tests have been performed which show strong confidence the dependency between the number of predicates and the infeasibility of a path. It is also concluded that since the problem of identifying the infeasibility of a path is undecidable, any metric which is capable of distinguishing between infeasible and feasible paths should be categorized as a heuristic.

Forgacs et al. [64] believe that the main goal of testers is generally not to execute a specific path but rather to reach one (or more selected program points where they think that it must be tested). Therefore, instead of minimizing the number of predicates on a path as having been suggested by Malevris [139], the number of predicates which are actually determinant to reach that point should be minimized. They propose using dataflow analysis and program slicing to select a set of potentially feasible paths to reach a given point in the program. However, no statistic has been given to show the effectiveness of the proposed heuristic.

Bueno et al. [27] propose an approach to the identification of likely infeasible program paths. This is a dynamic approach, which is based on program execution. The approach uses genetic algorithms to search for input data which drives the program execution to an intended path. The fitness function for genetic algorithms combines information both on control and data flow. A likely infeasible path can be detected by monitoring the search process of the genetic algorithms. A heuristic has been proposed to conclude on the feasibility

of a path. The heuristic is based on the fact that when generating test data for a feasible path, “a continual population’s best fitness improvements can be observed as the path predicates are reached and solved by the genetic algorithms”. Therefore by monitoring the search process, an infeasible path can be identified as soon as possible. However, this approach, as other approaches based on genetic algorithm, depends very much on the fitness function.

We have also proposed an infeasible path detection strategy [153] which can be integrated with any dynamic test data generation technique to enhance their performance in the presence of infeasible paths. Our approach combines the use of empirical properties of infeasible paths with dynamic information collected during test data generation to detect the infeasibility of the path under consideration. During the test data generation process for a path, the program execution is monitored and checked against the empirical properties. If a match occurs, the path is highly infeasible and the test data generation process is terminated immediately without any wasted effort. However, this approach can only be used with dynamic test data generation. Moreover, the approach results in some false positive cases which make it not suitable for use to generate test cases for critical systems.

3.2 Input error correction recovery, testing and maintenance

This section reviews related work in the area of input error correction, design recovery, integrated functional and structural testing, and slicing-based software maintenance.

3.2.1 Input error correction

User input error is one of the most insidious sources of failure and data loss in today’s IT environments [26]. User error happens for many reasons which might lead to severe impacts. Even in advanced technology systems, malfunctions of automated safety systems and mal-adaptation of user interactions often result in serious accidents [216]. Total elimination of human errors, therefore, may be difficult. A complementary approach to error elimination could be through error detection and correction. Kontogiannis [120] presents a three-phase error correction process as follow:

1. **Error detection** – realizing that an error is about to occur or suspecting that an error has occur.
2. **Error localization** – explaining why an error occurred.
3. **Error correction** – modifying an existing plan or developing a new one to compensate. There are three possible correction strategies:

- a. *Backward correction* in which the system is brought back to the original state occupied before the commission of the error. This means that users can reverse their actions, for example the 'undo' command' in text editing.
- b. *Forward correction* in which users may seek to bring the system to an intermediate stable state in order to delay the resultant erroneous effects and to find a better solution later on. This strategy is often used when critical equipment has been damaged and the available response time is limited.
- c. *Compensatory correction* in which users may activate redundant equipment and bring the system to the desired state that was originally intended.

An important step in error detection and correction is the identification of potential human errors. Wright et al. [219] state that users' behaviors should be analyzed so that possible errors associated with such behaviors and the consequences can be identified. This can be done through using detailed description of the human-computer interactions. Errors are then classified as either non-accomplishment of required action (errors of omission), incorrect accomplishment of required action (value errors) or extraneous action (error of commission).

Undo is the most widely supported feature in software systems especially systems which require a graphical user interface. Undo enables the system to recover from erroneous user operations, learn new system features easily and explore alternative solutions. The ability to undo any operation at any time is, therefore, especially important. Undo error correction strategy can be classified into single-step undo, chronological undo and selective undo. **Single-step undo** [221] is the most common undo model in single user applications which only allows undoing the last operation. **Chronological undo** [221] allows user to undo a sequence of operations in the chronological order from the last executed operation to the desired one. Basically, chronological undo can be achieved by repeatedly executing the single-step undo until all operations are undone. The basic idea of **selective undo** [16] is to let the user select a previous operation and append a new operation to the history that reverses the effect of the operation to undo. In case that the undo command does not make sense, the user is not able to select it. Among the three types of undo, only selective undo enables users to undo any operation at any time.

Input error correction should be provided in a database application as a major and important component of any database application is processing input submitted by users. However, existing work related to user error correction only deals with human-computer interaction errors in general. To the best of our

knowledge, none of the existing approaches covers the characteristic of programs which implement input error correction for database applications, one of the most important software domains.

3.2.2 Design recovery

Reverse engineering is *“the process of analyzing a subject system to identify the system’s components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction”* [35]. Reverse engineering helps in understanding the software architecture, recovering or extracting design and features, given the source code.

Reverse engineering covers a broad range of sub-areas. Our approaches to input error correction recovery is closely related to the design recovery sub-area. In design recovery, domain knowledge, external information and deduction or reasoning are added to the observations of the subject system to *“identify meaningful high level abstractions beyond those obtained directly by examining the system itself”*[35]. In this way, *“design recovery recreates design abstraction from a combination of code, existing documentation, personal experience and general knowledge about problem and application domains”* [35]. We approximately subdivide design recovery techniques into the following categories: (1) plan-based, (2) design-pattern-based, (3) slice-based.

3.2.2.1 Plan-based approaches

Plan-based techniques rely primary on using graph or pattern matching to identify clichés, plans or patterns within source code. A **cliché**, **plan** or **pattern** is a description of a functional feature contained within a program. The description can be specified using a formal language or expressed through a graphical model.

Different pattern matching algorithms namely code-to-code, concept-to-code have been introduced [118, 119] for design concept detection. Code-to-code matching is used to detect redundant code within a program and to compute the similarity between two code fragments. It is based on a dynamic program pattern matcher that computes the best alignment between two code fragments in term of insertion, deletion and substitution costs between statements of a model code fragment and an input code fragment. The concept-to-code matching algorithm first represents the design concept in an abstract language then computes the probability that an abstract statement can generate a particular code fragment. A language for the abstract representation of design concepts is also introduced. However, the methods are programming language-dependent.

Sartipi et al. [182, 183] present a graph matching model for the software architecture recovery problem. This approach attempts to find a mapping from a pattern graph, which represents the high-level architecture of the software, onto a graph representation of the source code entities. An environment for the recovery process is also presented. The environment consists of an off-line pre-process phase and an online analysis phase. During offline information extraction, the graph representation of the software system is formed and divided into a collection of sub-graphs. During the on-line analysis, the user defines the architectural graph and the tool will identify correspondences between sub-graphs and modules in the architectural graph. The approach imposes some constraints on what source-code entities can be mapped to high-level architectural graph.

Zhang et al. [230] introduce an approach which can overcome limitations of the approach proposed by Sartipi et al. [182, 183]. This approach solves the problem of refining and completing a partially defined mapping from source code to the design model by the software engineer. It extends the partial mapping to a more complete one by solving a system of simultaneous flow set equations. However, this approach requires an initial model and a partial mapping by users. Thus, it is vulnerable both to errors in the model and to errors in the implementation. Moreover, the effectiveness of flow analysis becomes a liability when mistakes in the initial model or mapping results in an inconsistent system. Moreover, the paper cannot conclude whether the method works for large systems.

3.2.2.2 Design-pattern-based approaches

Design patterns [69] are descriptions of successful solutions to common software problems including program functional features. They have the ability to capture the rationale behind design solutions and discuss the trade-off among their alternatives. They are the root of many key elements of large-scale software systems. In order to comprehend these systems, we need to first locate and then understand patterns on which they were built. In reverse engineering, the usefulness of design patterns encounters strong resistance because one pattern can be implemented in many different ways without ever being the same twice. Nevertheless, many ongoing researches have been attempting to recover design patterns from source code. Most approaches express programs and patterns using a structural representation and perform the matching between the pattern and source code using either meta-programming [90, 113], graph re-writing system [156, 164, 184] or logic programming [91, 165]. As long as the same amount of information is extracted and the same constraints are used, the matching technique does not affect accuracy, only efficiency.

Keller et al. [113] introduce a prototype system for the reverse engineering of design components based on the structural description of design patterns. The basic idea of the approach is that they use a representation of the design pattern, which they call Abstract Design component, to detect the implementation of the design pattern from source code. However, there are some parts, which are not clearly specified. For example, it is not clear how to specify an abstract design component to represent a particular design pattern. The algorithm to match between the abstract components to code is not presented in the paper. Moreover, the applicability of the method to automatic recovery is still a question mark.

A semi-automatic method for recognizing instances of a design pattern is proposed by Niere et al. [156]. Each design pattern is represented by a graph transformation rule. By aligning code with rules, implementation is actually mapped to patterns. After the source files are parsed, bottom-up matching will be applied to identify correspondences between code and rules. When a rule that depends on other rules and cannot be applied in bottom-up mode, the algorithm switches to top-down mode. The author believes that it is more effective to combine human domain knowledge to refine the analysis. Each successful mapping will be useful information for engineers to revise the rules accordingly. The approach does not solve the problem of implementation variant since not all the variants of a pattern are defined.

Heuzeroth et al. [90] present an approach to support software understanding by detecting design pattern automatically. The approach combines both static and dynamic analysis since the authors believe that neither static nor dynamic analysis themselves is adequate to find patterns in software systems. The approach first uses static pattern detection on code to obtain some information on the pattern. This information is then filtered using dynamic analysis. This approach can improve the quality of detecting the pattern in the sense that protocol conformance of a pattern can be checked. However, if pattern candidates are not executed during dynamic analysis, this approach cannot conclude anything about whether the pattern conforms or violate the protocol. The paper does not mention about the language used to define static and dynamic patterns.

Another different approach which is based on the reclassification of design patterns is proposed by Shi and Olsson [187]. They propose reclassification of design pattern by their intention which is more suitable for reverse engineering and are driven by code structure or system behaviors. Based on this reclassification, various patterns can be recognized more accurately than existing tools through using data flow diagram. Their tool is able to detect most of the patterns defined by Gamma et al. [69].

3.2.2.3 Slice-based approaches

Program slicing is a method for restricting a program to a specified subset of interest. It is a decomposition technique, which extracts program statements relevant to a particular computation. Therefore, it is very interesting and has been used widely in numerous fields in software engineering including program debugging [111], testing [124], software metrics [160], parallelization [210], software maintenance [68, 132] and reverse engineering [14, 130].

Lanubile et al. [130] realize that it is important to recover reusable functions from program source code. They introduce transform-slicing techniques for this purpose. The approach requires the user to identify a criterion that involves a set of input variables, a set of output variables and an initial statement. Thus, this technique faces difficulties in identifying the statements to slice upon to recover the functionality from program source code.

An other approach to extracting functionalities from source code based on static slicing techniques augmented for handling input/output statements is proposed by Tan et al. [199]. This approach is developed based on the hypothesis that a program delivers its functionalities through its output-statements. As such, it is not useful for programs that do not satisfy the hypothesis such as non-database or non-data-intensive programs. Moreover, it is limited by the existing techniques that deal with input/output statements in slicing. Thus in some cases, the approach may yield a slice that is larger than what is required.

Wong et al. [215] report a study applying an execution slice-based technique to identify the code which is unique to a feature or is common to a group of features. They believe that static slicing is less effective in identifying code segment that is uniquely related to a given feature because it produces a larger segment of code which includes many common utility codes. For this reason, they use execution slice, which is the set of program code executed by a set of test inputs. They find code unique to a feature by identifying code that is executed by any test that shows functionality of a feature but not by any that does not show the functionality of the feature. Their approach depends very much on the selection of test cases. Poorly selected test cases will lead to inaccurate identification of code.

3.2.3 Integrated functional and structural testing

Software testing is an active research area in software engineering. Many testing techniques have been proposed [6, 25, 40, 46, 49, 115, 144, 146, 181, 186, 206, 220]. Software testing techniques can be broadly classified into structural and functional testing techniques [108].

Functional testing (Black-box testing) is based on the idea that any program can be considered to be a function that maps values from its input domain to values in its output range. In functional testing, the program is considered to be a black-box. Thus, functional testing does not focus on the internal structure and the implementation of the program. The only information used is the specification of the software. According to Jorgensen [108], functional test cases have two distinct advantages:

1. They are independent of how the software is implemented, so if the implementation changes, the test cases are still useful.
2. Test case development can occur in parallel with the implementation, thereby reducing overall project development time.

However, a black-box test suite might contain significant redundancies. Moreover, black-box test cases might fail to test those behaviors of the program which are not specified.

Best known among functional testing strategies are: boundary value testing, equivalence class testing and decision-table based testing. The rationale behind **boundary value testing** is that errors tend to occur near the extreme values of an input variable. Therefore, boundary value testing uses input variables values at their minimum, just above their minimum, a nominal value, just below their maximum and at their maximum. In **equivalence class testing**, the input domain is partitioned into a set of equivalence classes based on an equivalence relation. Consequently, elements in each equivalence class have something in common. The idea of equivalence class testing is to identify test cases by using one element from each equivalence class. If the equivalence relation is wisely formulated, this will greatly reduce the potential redundancy among test cases. Jorgensen [108] believes that the choice of the equivalence relation very much follows a craftsman's approach where one needs to second-guess the likely implementation and thinking about the functional manipulations that must somehow be present in the implementation. Of all the functional testing methods, **decision table-based testing** is the most rigorous because decision tables enforce logical rigor. This technique works well for situations in which a lot of decision making takes place and those in which important logical relationships exist among input variables. A decision table has two parts: the input conditions part and the output actions part. The decision table specifies under what conditions a test action must be performed. All the possible combinations of input conditions define a set of rules. For each rule, a test action should be considered. The rules are then interpreted as test cases. However, decision table-based testing does not scale up very well as the number of rules increases exponentially with the number of input conditions.

Structural testing (White-box testing) techniques design test cases based on code characteristics. Structural testing has been the subject of very strong theory that it lends itself to the definition and use of test coverage criteria [107]. A test coverage criterion provides a way to “*explicitly state the extent to which a software item has been tested, and this in turn makes testing management more meaningful*”[108]. As white-box test cases are based on source code, they are not able to recognize those program behaviors which are specified but are not programmed.

Best known among structural testing strategies are statement, branch and path testing. **Statement testing** requires each statement in the program to be executed by at least one test case. **Branch testing** requires that all the control transfers in the program under test are exercised during testing. Path testing requires that all the complete paths through the program are executed during testing. **Path testing** was first introduced by [96]. This testing technique, however, is problematical because of the huge (possible infinite) number of paths. Since then, many path based coverage criteria have been proposed which aim to bridge the gap between branch and path testing and make path-based testing more effective and feasible. The most commonly used path-based testing strategy is **basis path testing** [141].

A **basis set** is a set of complete paths through a CFG that are linearly independent and the paths in the basis set can be used to construct any path through the CFG.; a path in the basis set is thus called a basis path. This coverage criterion requires that all the basis paths in a program must be executed during testing. This implies directly from the fact that if a path is a linear combination of tested paths, it can be considered as redundant. McCabe [141] has proposed a cyclomatic complexity measurement, which is in fact the number of basis paths in a program, and an algorithm to generate a basis set from any given flow graph. Loop construct is the most complicated control element which makes the number of path in a program infinite. There are a number of approaches which are concerned with the testing of loops. The **Ct testing** strategy proposed by Bently et al. [15] limits the number of test cases by specifying a minimum iteration count k . This testing criterion requires that every loop in the program under testing should be executed zero times, once, twice and so on up to k .

Most of the existing software testing techniques adopt either a structural or a functional strategy that solely bases on code or specification respectively. To test a software system more accurately, both code and specification behaviors must be covered [17, 108, 176]. Unfortunately, very few approaches have been proposed to address this problem.

Richardson and Clark [176] have introduced the partition analysis technique to verify a procedure's implementation against its specification. The **partition analysis technique** is composed of three steps. In the first step symbolic evaluation is applied to translate both the specification and the implementation into a functional representation called specification-based partition and implementation-based partition. Each partition contains a set of sub-domains. By forming the intersection of the specification sub-domains and the implementation sub-domains, the sub-domains of the procedure partition are formed. In the second step, for each sub-domain in the procedure partition, verification techniques are used to determine the equality between the description of the elements in the specification sub-domain and the computational descriptions of those elements in the implementation sub-domain. Any inconsistency should be reconciled before moving to the last step. Finally, the computational descriptions are used to generate test data that extensively exercises the functional behaviors of the sub-domain. We believe that the use of symbolic execution is a limitation of the approach as complicated functional features such as input error correction cannot be represented using symbolic expressions.

Chen et al. [32] propose a solution to the problem of black-box test suite reduction through the use of path-based (white-box) information. A black-box test suite covers all compatible combination of classes of inputs to ensure that all aspects identified from the specification will be thoroughly tested. However, a black-box test suite often contains too many test cases to be practically tested in its entirety. To reduce the number of test cases, the authors suggest partitioning existing black-box test cases into equivalent classes with respect to a given white-box criterion. Given a white-box coverage criterion W , two test cases are said to be W -equivalent if they are considered to be "processed similarly" according to the coverage criterion W . A limited number of test cases are chosen from each equivalent class; thus reducing the test suite size. Based on this idea, the authors have also proposed an algorithm for white-box oriented selection of black-box generated test cases.

Baydeda et al [17] briefly discuss the benefit of integrating black-box and white-box testing. The authors then propose a new graphical representation of classes, called Class Specification and Implementation Graph (CSIG), for integrating black-box and white-box testing. In this representation, each method of a class is represented by two control flow graphs. One of them is a control flow graph constructed on the basis of method specification (method specification graph) whereas the other is control flow graph determined using the method implementation (method implementation graph). To construct the CSIG, first the method implementation graph is constructed. This can easily be

done by through control flow analysis. The next step is to generate a prototype for each specified method. The prototype contains necessary information of the corresponding method's specification. As such, it is used as a basis for test case identification. As the third step, the method specification graph is generated for each prototype constructed in the second step. After that, all the method specification graphs and the method implementation graphs for all the methods are combined in a class control flow graph frame (CCFG frame), which acts as an abstract test driver. Method graphs are corrected by control and data flow edges. Control flow edges indicate that a method can be invoked by another method. Data flow edges show that a method assigns values to attributes used by another method. Once the CSIG has been constructed, well-known structural testing techniques can be used to obtain test cases which cover both specification and implementation characteristics. However, no experiment on the fault detection capability of the proposed presentation has been done.

3.2.4 Using slicing in software maintenance

Program slicing is a method for restricting a program to a specified subset of interest, which has been used widely in numerous fields in software engineering including debugging, program understanding, testing, reverse engineering and software maintenance [21, 103, 111, 124, 128, 132].

Software maintainers are faced with the problem of understanding the existing software and making changes without having negative impacts on the unchanged parts. Gallagher et al. [68] propose using the decomposition slicing technique to aid in making changes to a piece of software without unwanted side effects. While a program slice is dependent on a variable and a statement number. A decomposition slice does not depend on statement numbers. A **decomposition slice** with respect to a variable v_k captures all the computations of v_k and is defined as follows:

Let O_{v_k} be the set of statements in program P that output variable v_k , let n_l be the last statement of P and let $\Phi = O_{v_k} \cup \{n_l\}$. The statements in $\mathcal{DS}(v_k, P) = \bigcup_{n_k \in \Phi} S(v_k, n_k)$ forms the decomposition slice on v_k .

Two decomposition slices are **independent** if they have no statements in common. A slice is **strongly dependent** on another slice if it is a subset of the latter. A slice which is not strongly dependent on any other slice is **maximal**. A statement which is shared by more than one slice is **dependent**; otherwise it is independent. A variable which is defined by an independent statement is independent. The **complement** of a decomposition slice is defined as the

original program minus all the independent statements in the decomposition slice. The complement is also a program slice.

Intuitively, a decomposition slice on a variable is the union of certain slices taken at certain line numbers on the given variable. It is used to guide the removal of statements in a systematic fashion to construct the complement by removing only independent statements. Since it is required that a slice be executable, dependent statements are crucial on both the slice and its complement. Based on this decomposition slicing concept, program can be decomposed into manageable pieces to automatically assist the maintainer in guaranteeing that there are no ripple effects introduced by modifications in a component. The complement provides a semantic context for modifications in the decomposition slice; the complement must remain fixed after any change.

According to Gallagher, modifications can take three forms: additions, deletions and changes. A change can actually be considered as a deletion followed by an addition. As such, Gallagher determines only those statements in a decomposition slice that can be deleted and the forms of statements that can be added. The following set of guidelines has been proposed by the authors to ensure that modifications to the decomposition slice should not affect the complement; thus, only testing of the modified slice is necessary.

1. **Rule 1** – *Independent statements may be deleted from a decomposition slice*: independent statements do not affect data flow or control flow in the complement; thus they can be removed from the decomposition slice without affecting the complement.
2. **Rule 2** – *Assignments statements that target independent variables can be added anywhere in a decomposition slice*: independent variables are unknown to the complement. Therefore, changes to them cannot affect the computation of the complement.
3. **Rule 3** – *Logical expression and output statements maybe added anywhere in a decomposition slice*: Evaluation of logical expression or the inclusion of output statements will not even affect the computation of the slice. Thus, the complement remains intact. However, care must be taken then adding statements which are control dependent on the logically expression to make sure that they do not interfere with the complement.
4. **Rule 4** – *New control statements that surround any dependent statement will affect the complement's behavior*: Newly added control statements will be included in the complement due to the fact that the dependent statements are in both the slice and the complement. Thus any control statements which control the dependent statements will also be in the slice and the complement.

The decomposition slicing technique gives the maintainer a view of the program with the unnecessary statements removed and dependent statements restricted from modification. The decomposition slice zooms in a smaller piece of code for the maintainer to focus on, while the rules provide the means by which the deleted and restricted parts cannot be changed accidentally. Gallagher also proposes an algorithm to merge the modified slice back into the complement in four steps:

1. Order the statements in the original program (i.e., line numbering). The program slice and the complement can be identified with the line numbers from the original program. The sequence numbering of the slice and its complement are referred to as slice sequence and complement sequence, respectively.
2. For deleted statements, delete the sequence number from the slice sequence.
3. The statements inserted into the slice, a new sequence number needs to be generated. Let p be the sequence number of the statement preceding the statement to be inserted. Let m be the least value in the slice sequence greater than p . Let $f = \min(\lceil p \rceil, m)$. Insert the new statement at sequence number $(f + p)/2$.
4. The merged program is obtained by merging the modified slice sequence values into the complement sequence.

In this algorithm, the unchanged dependent statements are used to guide the reconstruction of the modified program. The basic idea behind this algorithm comes from the observation that through the technique the maintainer is editing the entire program.

However, researchers [97, 205] soon realize that the “interference” dependency between two variables is missing in the decomposition slice graph. Tonella [205] proposes a new program representation called a **concept lattice of decomposition slices** which is an extension of decomposition slice graph with additional nodes associated with weak interference between computations. Basically, a decomposition slice graph is not a lattice, i.e. infimum and supremum are not always unique; therefore, the decomposition graph cannot capture all the relationships between computations. Based on this idea, Tonella forms a concept analysis problem in which there exists a binary relationship between a variable and a statement if the statement belongs to the decomposition slice of the variable. As such, the resulting concept “*contains a set of variables which share some common computations*”. This new program representation is more useful compared to decomposition slicing in term of software maintenance, especially impact change analysis as all the dependencies between computations are explicitly represented. Consequently,

the problem of impact change analysis is equivalent to identifying a set of reachable nodes in the concept lattice.

Surgeon's Assistant [66] is a CASE tool which also uses decomposition slicing to analyze and limit the scope of changes to the program. The tool also incorporates a decomposition slicing system that uses an acyclic graph to display slices as nodes to the maintainers. This graph is also referred to as a decomposition slice graph which shows computation inclusion relationship between different variables in a program. Ricca et al. [175] mention the use of decomposition slicing in understanding web applications especially its organization, main computations and sub computations. The use of a decomposition slice graph helps in evaluating the impact of a modification on web pages in the web application.

3.3 Reverse engineering of web applications

Web applications are one of the fastest growing classes of software systems in use today. These applications support a wide range of activities including business functions, scientific activities as well as medical activities [6, 169, 173]. Hassan and Holt [87] provide a structure for a web application which consist of a set of components that interacts to form a system. The structure is depicted in Figure 3-3.

Figure 3-3. Structure of a Web Application (Adapted from [87])

Components of a web application might be written in various programming language and run on multiple machines distributed over the network. Hassan and Holt believe that there is a set of recognizable components which comprise a web application. This set consists of *web browsers*, *web servers*, *application servers* and the following components:

1. **Static pages** contain only HTML code and executable code that runs on the web browser.
2. **Active pages**, for example Active Server Pages (ASP) and Java Server Pages (JSP), contain a mixture of HTML tags and executable code. When users request for an active page, the application server preprocesses it and integrates data from various resources such as web

objects or databases to generate a final HTML web page and send to the web browser.

3. **Web objects** are pieces of compiled code which provide a service to the rest of the software system through a defined interface.
4. **Multimedia objects** such as images and videos.
5. **Databases** are used to store data that is shared among the various components.

Most of the web applications utilize a Database Management System (DBMS) and one or more databases. A program in a web application normally interacts with databases through SQL statements; these are referred to as **database interactions** in web applications. SQL statements can be classified into two categories: **Data Definition Language** (DDL) and **Data Manipulation Language** (DML) statements [143]. Within the program analysis point of view, our interest concerns DML statements. SQL DML statements are targeted towards manipulating the data stored in the database such as insert, update and delete.

- **Insert** statements insert data into a database table. Regardless of the specific syntactical variation used, if an insert statement executes successfully, all the columns of the references table will be defined. Therefore, even if the statement refers to only a subset of the table columns, the remaining columns will also be defined either according to their default value or the NULL value.
- **Update** statements modify the values of one or more columns in one or more rows, according to the specified search conditions.
- **Delete** statements delete those rows of the target table that meet the search criteria specified in the statements. If there is no search criterion then all the records of the table will be deleted.

Clearly, database interactions are among the most important functional features in web applications. Our approach to extract database interactions from source code is most closely related to work in the area of reverse engineering of web applications, especially design recovery. A brief literature review of general approaches to design recovery has been provided in Section 3.2.2. In this section, we only focus on reverse engineering for web applications.

Reverse engineering processes have proved to contribute an important role in the maintenance of traditional software systems by providing powerful techniques for recovering high-level abstraction and documentation from undocumented or poorly documented software systems. However, reverse

engineering a web application encounters several challenges, one of which is the dynamic software components in a web application. Such components are created on-the-fly depending on the user interaction with the application. Therefore, to be successful, processes for reverse engineering a web application might require expensive human expertise.

Several approaches and tools for the reverse engineering of web applications have been proposed in the literature. Some of them aim at obtaining an architectural view of the web application that depicts web application components and their relationship at different level of details [38, 52, 87, 174]. Some others [51] allow abstracting a description of the functional requirements implemented by the web application which are then represented using UML [158] use case diagram. Some others else [54, 55] recover UML class diagram of the application business objects and the logical relationship between them or model the business processes implemented by the web application [202].

The first contribution towards understanding web applications is given by Ricca and Tonella [173, 174]. They develop a tool called ReWeb for performing analysis on web pages. A graphical representation of the web site is introduced, which is an extension to traditional static flow analysis. Dynamic information, in the form of web server logs, is also mentioned in this paper to enhance the analysis. However, the logs do not allow fine-grained analysis.

A tool, named WARE, is developed by Di Lucca et al. [52] to recover UML documentations from web applications. WARE performs static analysis first and then dynamic analysis to complement static information to detect page clusters. In a later paper [53], Lucca recommends a reverse engineering process for web applications, which includes five steps: (1) static analysis of the web application, (2) dynamic analysis of the web application, (3) automatic clustering of the web application into clusters of meaningful and independent components, (4) validation of the clustering and (5) abstractions of UML diagrams. There are on-going experiments to validate and explore further into this framework.

Antoniol et al. [6] introduce another approach to understanding web applications using code instrumentation. A tool named WANDA is developed which instruments the web application and combines static information with dynamic information to recover the architecture of the web application and its documentation. Most of the dynamic information is collected from markers instrumented in the original source code during the web application execution. However, it is not clear that how the web application should be invoked to ensure sufficient dynamic information. Currently, this is done by letting a user

interacting with the web application for hours, trying to discover all the web application features.

Hassan and Holt [87] present an approach to recover the architecture of a web application to facilitate web application maintenance. The approach is capable of extracting the dynamic structure of a web application as well as presenting the interactions between various components of a web application. The approach uses a set of specialized parsers and extractors to examine not only source code but also binaries of a web application. The extractors generate facts about components, relations and attributes of the software system. These facts are then combined with human expertise to produce an architecture diagram.

Lucca et al. [51] discuss the use of clustering for decomposing a web application into a group of functionally related components. Basically, to do clustering, three issues must be considered including (1) build a model in which components to be clustered are adequately described; (2) define a criterion under which component should be clustered into a cohesive unit and (3) select a clustering algorithm. The authors introduce a conceptual model for web applications in which components are related through the following types of relationships: *link*, *submit*, *redirect*, *build*, *load-in-frame* and *includes* one. As for the second choice, the coupling between components on the basis of relationship between them is used as the criterion to group conceptually related components. The more link between components, the stronger the coupling between them. For the third consideration, hierarchical clustering is used to obtain different partitioning of a system at different levels of abstraction. The approach has been experimented with medium size web applications and produces interesting and encouraging results.

3.4 Summary

In this chapter, we have reviewed work related to (1) infeasible path detection, (2) input error correction recovery, testing and maintenance and (3) database interactions extraction for web applications.

Existing approaches to infeasible path detection can be classified into precise and approximated methods. Precise methods cannot scale well and existing approximated methods are not capable of detecting a large number of infeasible paths. In Chapter 5, we propose an approximate approach which can detect majority of infeasible paths with high accuracy.

Input error correction has been studied in depth by human-computer interaction researchers. However, they only deal with human-computer interaction errors in general. None of them cover the characteristic of programs which implement

input error correction for database applications. Existing approaches to design recovery of software systems are very general which are hard to apply to recover input error correction as they do not provide any mechanism to derive possible input errors from the source code of a system. To the best of our knowledge, we have not seen any approach that uses any of these techniques for the automated recovery of input error correction. In Chapter 6, we propose an empirical recovery approach which specifically target input error correction feature. Though several techniques have been proposed for integrating functional and structural testing, these techniques are not applicable for the testing a complex functional feature like input error correction. In Chapter 6, we use recovered information on input error correction to combine both code and specification characteristics in testing this feature. Decomposition slicing technique has shown its usefulness in maintaining software systems. Adopting the idea of decomposition slicing, we propose in Chapter 6 the effect-oriented technique to aid software engineers in maintaining input error correction without introducing any ripple effect.

Database interactions are among the most important functional features in web application. Recent research work on reverse reengineering of web application using static program analysis cannot be applied directly to extract database interaction because these approaches do not differentiate between different types of interactions that can be executed by a program statement. On the other hand, dynamic approaches, which are based on executing the application on particular inputs, cannot be generalized to all executions. In Chapter 7, we propose an approach which combines symbolic execution with empirical properties to extract all database interactions automatically.

Chapter 4

A GENERIC INFRASTRUCTURE TO SUPPORT EMPIRICAL PROGRAM ANALYSIS

While doing research in applying program analysis augmented with empirical properties in software analysis, testing and maintenance, we realize the need for a generic infrastructure which supports program analysis and facilitates the discovery and validation of new empirical properties. This chapter presents such an infrastructure that we have developed to support our research work. All the prototype tools presented in Chapter 5, Chapter 6 and Chapter 7 are based on this infrastructure. The infrastructure is composed of two main parts: **EmAnalyzer** and **EmValidator**. **EmAnalyzer** provides program analysis functions for programs written in Java while **EmValidator** assists in the validation of new empirical properties. The infrastructure is implemented in Java under the Eclipse [59] Integrated Development Environment (IDE). It is the integration and extension of various existing software engineering tools.

4.1 EmAnalyzer

EmAnalyzer provides tools to perform program analysis on Java programs and to visualize analysis information. The architecture of **EmAnalyzer** is shown in Figure 4-1. **EmAnalyzer** consists of three modules: **Soot** – a Java bytecode analysis and optimization framework, **PathGen** – a path generation tool – and **EmViewer** – a collection of viewing tools. In this section, we discuss each module in detail.

Figure 4-1. EmAnalyzer architecture

4.1.1 Soot

Soot is developed by the Sable research group from McGill University, whose objective is to provide tools leading to the better understanding and faster execution of Java programs. The Soot website is at <http://www.sable.mcgill.ca/soot/>. Since the first release of Soot, it has been used in various research projects as an efficient program analysis tool for Java programs.

One of the main benefits of Soot is that it provides four different Intermediate Representations (IRs) for different analysis purposes. Each of the IRs has different levels of abstraction that give different benefits when doing program analysis, they are: Baf, Jimple, Shimple and Grimp.

1. **Baf** is a bytecode representation resembling the Java bytecode but abstracts away the constant pool and abstracts type dependent variations of instructions to a single instruction. For example, in Java byte code, there is a number of instructions for adding integers, longs, etc., in Baf, they have all been abstracted into a single instruction for addition.
2. **Jimple** is a stackless typed 3-address, statement based representation suitable for most analysis; resulting in a very convenient representation for doing program analysis. In Jimple, an analysis only has to handle 15 types of statements compared to the more than 200 possible types of instructions in Java bytecode.
3. **Shimple** is a Static Single Assignment (SSA) version of the Jimple representation. SSA-form guarantees that each local variable has a single static point of definition which significantly simplifies a number of program analyses.
4. **Grimp** is similar to Jimple but allows tree expressions. In this respect, Grimp is closer to resembling Java source code than Jimple is and so is easier to read; and hence the best intermediate representation of inspecting disassembled code by a human reader.

Figure 4-2 shows part of the class diagram of Soot containing main data structures of Soot.

- **Scene.** The Scene class represents the complete environment the analysis take place in. Through this class, we can set various parameters such as the application classes (the classes supplied to Soot for analysis), the main class (the one that contains the main method) and access information regarding interprocedural analysis (e.g., call graphs).
- **SootClass.** Represents a single class loaded into Soot or created using Soot.
- **SootMethod.** Represents a single method of a class.
- **SootField.** Represents a single member field of a class.
- **Body.** Represents a method body and containing four child classes corresponding to the four IRs including BafBody, JimpleBody, ShimpleBody and GrimpBody.
- **Stmt.** Represent a statement in the method body. For each statement, Soot provides the following methods to obtain dataflow and control flow information:
 - **getUseBoxes():** get values used at a node
 - **getDefBoxes():** get values defined at a node
 - **getUseAndDefBoxed():** get both values used and defined at a node

- **getBoxesPointingToThis():** get nodes jumping to this node
- **getUnitBoxes():** get nodes to which this node is jumping to
- **fallsThrough():** returns true if execution after this node may continue at the following node
- **branches():** returns true if execution after this statement does not necessarily continue at the following node

These data structures are implemented using Object-Oriented techniques and designed to be easy to use and generic where possible.

Figure 4-2. Part of the class diagram of Soot

Soot provides several different implementation of CFG. All these implementations are based on the interface `DirectedGraph` which provides methods for getting:

- The “begin” and “end” node of the graph
- The successors and predecessors if a given node
- The graph size

The base class of all types of CFGs is the `UnitGraph` class which provides facilities to build CFGs. There are three different implementations of this class including `BriefUnitGraph`, `ExceptionalUnitGraph` and `TrapUnitGraph`. `BriefUnitGraph` is very simple in the sense that it does not have edges representing control flow due to exceptions being thrown while the other two implementations include edges from throw clauses to their handlers to the

method body. Figure 4-3 presents the class diagram of different types of CFGs supported by Soot.

Figure 4-3. Different types of CFGs in Soot

The Soot Driver component of EmAnalyzer contains various drivers to provide basic supports to process a Java system represented in Soot's IRs. Figure 4-4 shows the type hierarchy of all the Soot drivers implemented. All the Soot drivers handle calling the appropriate Soot's APIs with the respective arguments input by EmAnalyzer. SootBasedDriver is a generic driver which provides functions to determine the location of the input files and libraries and the correct location for the output files and handle all the messages generated by Soot. SootBasedDriver also set the following Soot options to provide information required by our analysis:

- -whole-program: run Soot in whole program mode, taking into consideration the whole program when performing analyses and transformations.
- -allow-phantom-ref: allow Soot to process a class even if it cannot find all classes referenced by that class.
- -trim-cfgs: when constructing CFGs which include exceptional edges, Soot minimizes the number of edges leading to exception handler by analyzing which instructions might actually be executed before an exception is thrown, instead of assuming that every instruction protected by a handler has the potential to throw an execution the handler catches.
- -keep-line-number: preserve liner number tables for class files throughout the transformation so that analysis information can be mapped back to source code.

Figure 4-4. Soot drivers

CFGGenerator first runs Soot to transform class files into Jimple IR and then invokes the build graph APIs of Soot (class CFGGraphType) to construct a CFG for each instance of the SootMethod class. Control dependence analysis is not explicitly available in Soot. Thus, the ControlDependenceAnalysis class makes use of the dominator tree construction function provided by Soot to build a dominator tree in the reverse CFG of each CFG and then computes control dependence in the reversed dominator tree. The technique for computing control dependence in the reversed dominator tree has been discussed in Section 2.1.2. DataDependenceAnalysis combines several data dependency functions of Soot to provide variable definition and use analysis. It also provides some extra functions which are not supported by Soot, such as returning all the statements which define a local variable and checking whether a path is a def-clear path with respect to a set of variables.

4.1.2 PathGen

For each CFG produced by Soot, PathGen can be used to generate a set of paths through the CFG. PathGen supports the generation for the following sets of complete paths through a CFG:

1. A set of linearly independent paths (basis set)
2. A set of representative paths

A basis set is a set of complete paths through a CFG that are linearly independent and the paths in the basis set can be used to construct any path through the CFG. The number of paths in the basis set can be identified by the McCabe's complexity measure [141], which is equal to $|\mathcal{E}| - |\mathcal{N}| + 2$, where $|\mathcal{E}|$ is the number of edges and $|\mathcal{N}|$ is the number of nodes in the CFG.

The algorithm implemented in PathGen for constructing a basis set for a CFG is proposed by Poole [168], which actually is a modified Depth First Search (DFS) algorithm [106]. The search starts at the 'begin' node and recursively descends down all possible outgoing edges. If the node visited has never been visited before, a default outgoing edge is picked, and then the current path is split into new paths that traverse each outgoing edge, going down the default

edge first. A default edge can be chosen as any edge which is not a back edge or which later causes a node to have two incoming edges. If the node visited is the 'end' node, then a path in the basis set has been found. Otherwise, the path traverses the default edge. The pseudo-code of this algorithm is given in Figure 4-5. The algorithm is first called with the 'begin' node of a CFG. Poole has proved that this algorithm outputs a set of linearly independent paths in a CFG.

```

Algorithm findBasisPaths( $n_k$ )
begin
1. if ( $n_k$  is the 'end' node of the CFG) then
2.   add this path to the basis set)
   else
3.   if ( $n_k$  has not been visited before) then {
4.     mark  $n_k$  as visited
5.     label a default outgoing edge
6.     findBasisPaths(destination of the default outgoing edge)
7.     for (all other outgoing edges) do
8.       findBasisPaths (destination of the edge)
   }
9.   else
10.    findBasisPaths (destination of the default outgoing edge)
end

```

Figure 4-5. Algorithm findBasisPaths

```

Algorithm findRPaths( $n_k$ )
begin
1. if ( $n_k$  is the 'end' node of the CFG) then
2.   add this path the set of representative paths)
   else {
3.   if ( $n_k$  is the predicate node of an iteration construct) then {
4.     Mark the true branch of  $n_k$  as 'visited'
5.     findRPaths(destination of the true branch of  $n_k$ )
6.     for (all other outgoing edges) do
7.       findRPaths(destination of the edge)
8.     Mark the true branch of  $n_k$  as 'unvisited'
   }
   else {
9.     for (all outgoing edges of  $n_k$ ) do
10.      findRPaths(destination of the edge)
   }
   }
end

```

Figure 4-6. Algorithm findRPaths

In Chapter 6 and Chapter 7 we need to compute a set of representative sets in a CFG. A representative path (r-path) in a CFG is a complete path in the CFG,

such that any loop included is iterated with exactly one time. The algorithm for computing a set of representative paths in a CFG is quite similar to the algorithm for computing a basis set. The algorithm **findRPaths** is given in Figure 4-6. The algorithm is also a modified version of the DFS algorithm. The algorithm is first called with the ‘begin’ node as the input parameter. For each predicate node of iteration construct (while, for statement), if the true branch of the predicate node has not been visited, the algorithm traverses the true branch, at the same time marks it as “visited”. However, this branch will be re-marked as “unvisited” once the algorithm backtracks from the recursive call. For all other nodes, all the outgoing edges will be visited. If the node visited is the “end” node, a representative path is found and the path is added to the set of representative paths in the CFG.

As the true branch of a predicate node of iteration construct is marked as visited during a recursive call following the true branch, the algorithm makes sure that the loop will only be iterated at most once. As the true branch of the predicate node will be re-marked as unvisited when the algorithm returns from the recursive call, the algorithm ensures that all the representative paths in the CFG will be computed.

4.1.3 EmViewer

EmViewer produces various visual representations of the source code and program analysis information. All the graphs constructed by Soot can be viewed using the Dot component of the Graphviz [75] software package. Graphviz is a collection of software for viewing and manipulating abstract graphs. It provides graph visualization for tools and web sites in not only software engineering but also networking, databases, knowledge representation and bio-informatics. Dot draws directed graph as hierarchies. It reads attributed graph text files and writes drawings either as graph files or in a common graphic format such as GIF, PNG, SVG or PostScript. Soot provides a Dot Driver which converts a graph into attributed graph text files. We use this file, input it into Dot and use GIF as the output format.

Figure 4-7 shows the “Control Flow Graph View” of EmViewer. Users can click on the method name in the “Class View” panel; the CFG of the method will be displayed in the “Control Flow Graph View” panel.

The source code view is convenient for referring to the source code while using EmAnalyzer without opening another Java editor. To display Java source code in EmAnalyzer, we use the Java2Html converter [101]. Java2Html converts Java (and other programming language) source code into HTML, RTF, Tex and XHTML with syntax highlighting, which can be easily displayed in EmViewer.

Figure 4-8 provides a screenshot showing a typical use of the source code view for viewing analysis information on infeasible program paths.

Figure 4-7. CFG display in EmAnalyzer

For each path computed by the PathGen component, they are listed in the “Path View” panel of EmViewer in the form of a sequence of line numbers. Users can click a path and the path will be highlighted in the “Source Code View” panel of EmViewer. Actually, the path computed by the PathGen component is represented as a sequence of Soot Grimp statements; we refer to this type of path as Grimp path. As such, to highlight a path in the source code of the respective method, we need to first convert a Grimp path into a source code path. We enable the option “—keep-line-number” of Soot, so that for each Grimp statement, the line number of the respective source code statement will be attached. Based on this, we convert a Grimp path into a sequence of line numbers. We then use this sequence to annotate the source code at the line numbers in the sequence. We extend the Java2Html converter to realize these annotations and display the source code with those annotated statements highlighted. Figure 4-8 shows the “Source Code View” in EmViewer with a path highlighted.

The analysis information view is reserved for any analysis tool built on top of EmAnalyzer to display the analysis information. In Figure 4-8, the analysis information view is combined with the path view to display information on infeasible paths. For each infeasible path, the pattern which was used to detect

the path is shown together with the line numbers of the statements which cause the infeasible path.

Figure 4-8. Source code view in EmAnalyzer

4.2 EmValidator

Validating an empirical property $\mathcal{R}$ against an individual case C in the sample for testing $\mathcal{R}$ involves the following two steps:

1. Applying $\mathcal{R}$ on the case and collecting the resulting information.
2. Verifying the information obtained in (1) to confirm whether $\mathcal{R}$ truly holds for the case.

In the first step, algorithms are often designed to automate the application of empirical property $\mathcal{R}$ on C . The second step is more difficult in which the following activities are often involved:

- Slice the program, to which the individual case C belongs, to get a smaller view of the program with focus on the empirical property being investigated. In this way, it is easier to understand the program and verify the correctness of the information obtained.

- Generate test cases to verify the correctness of the information obtained by applying $\mathcal{R}$ on C. Information captured during the execution of the program with these test cases gives exact information which reflects the runtime behavior of the program. If test cases can be wisely designed, necessary information for verifying $\mathcal{R}$ can be achieved. However, test data generation is an extremely complicated and time consuming task as it requires careful analysis and understanding of the source code.
- Capture runtime information during a program execution.

Figure 4-9. EmValidator Architecture

As the above mentioned activities are often carried out repeatedly during the statistical validation of an empirical property, we have developed the EmValidator tool to provide support to this validation process as much as possible. The architecture of EmValidator is given in Figure 4-9. EmValidator consists of three main modules: (1) **TestGen** – A test data generation assistant tool, (2) **JTracer** – a source code instrumentation and execution trace collection

tool and (3) **Kaveri** – a program slicing tool. In the subsequent sub-sections, we discuss each module in detailed.

4.2.1 TestGen

The TestGen component takes a complete path as input and tries to generate test data for the path either automatically or through interacting with the user. For each path c_p , TestGen first extracts a symbolic expression representing the path domain by taking the conjunction of the branch predicates of all the branches that c_p follows. The expression is simplified and input to a simple constraint solver to determine whether it can be solved. Choco [36] is used as the constraint solver for the TestGen component. Choco is a Java library for constrain satisfaction problems (CSP), constraint programming (CP) and explanation-based constraint solving (e-CP). Each CSP problem in Choco is defined by a triplet $(V, \mathcal{D}, C)$ such as:

- $V = \{v_1, \dots, v_n\}$ is the set of variables of the problem.
- $\mathcal{D}$ is the problem domain which associates to each variable v_i its domain $D(v_i)$.
- $C = \{C_1, \dots, C_k\}$ is the set of constraints. Each constraint C_j is a relation between a subset of variables which restricts the domain of each one.

The test data generation problem can be easily modeled as a CSP problem in Choco. In this case, V is the set of input variables, $\mathcal{D}$ is the program domain and C is a set of branch predicates. Currently, Choco can only solve constraints for integer variables, set variables and set variables.

For example, test data generation for the path (1, 2, 3, 4) through the program in Figure 4-10 can be modeled as an CSP problem in Choco with V , $\mathcal{D}$, C are defined as follow: $V = \{x, y\}$, $D(x) = \text{all integers}$, $D(y) = \text{all integers}$, $C = \{(x < 5), (y > 3), (x < y)\}$. A simple program for solving this test data generation problem using Choco is given in Appendix B.

<pre> 1. if (x < 5 && y > 3) 2. // do something 3. if (x > y) 4. // do something else </pre>

Figure 4-10. An example program for Choco

If Choco cannot make any conclusion, TestGen will suggest suitable input values by searching through the Test Data Pool. The test data pool is constructed as follow. JTracer is used to instrument input statements (statements which read user input values) or parameters to collect information

on input variable names, their data types and the input values submitted during an execution of the program. Prior to the use of the data pool, we manually input some sample data into the pool. Moreover, whenever a test case is successfully generated, the input values will be recorded in the data pool for later use. To generate input values for a path, TestGen first identifies the set of input variables for executing the path. For each input variable, TestGen searches in the data pools for data items with compatible data types with the input variable. If there exists such data items, the ones with the closest variable names will be recommended to the user. If the path cannot be executed with the recommended values, the symbolic expression representing the path domain is displayed for solving manually by the user.

4.2.2 JTracer

JTracer consists of a source code instrumentation tool and the Java Debugging Interface (JDI) for collecting execution traces.

Table 4-1. Probes for instrumentable entities

Instrumentable entities	Information available
Method entry	<ul style="list-style-type: none"> ▪ enclosing object ▪ argument objects
Method exit	<ul style="list-style-type: none"> ▪ return or exception object
Before method call	<ul style="list-style-type: none"> ▪ target object ▪ parameters objects
After method return	<ul style="list-style-type: none"> ▪ return or exception object
Field read	<ul style="list-style-type: none"> ▪ field object ▪ containing object
Field write	<ul style="list-style-type: none"> ▪ old field object ▪ new field object ▪ containing object
Throw or catch	<ul style="list-style-type: none"> ▪ exception object
Assignment	<ul style="list-style-type: none"> ▪ value being assigned
Extension of InsectJ	
Predicate node	<ul style="list-style-type: none"> ▪ predicate outcome ▪ containing object

The code instrumentation tool is used when information other than execution traces need to be collected during a program's execution. For this purpose, we use InsectJ [185], a generic instrumentation framework for collecting dynamic information. InsectJ enables users to collect various types of runtime information with limited effort because it lets them easily specify the following attributes: (1) which types of program entities should be monitored at runtime, (2) which parts of the code which entities should be monitored, (3) what kind of information should be collected for each entity and (4) how to process the

information collected. More importantly, the framework can be extended to collect new types of information.

The InsectJ framework provides a set of predefined **probes**. A probe is typically associated with a program construct (e.g., method call or a predicate node or a variable). The code construct that can be instrumented with probes are referred to as **instrumentable entities**. This predefined set of probes covers the basic and important constructs for a Java program. Table 4-1 lists all the instrumentable entities provided by InsectJ and the one that we have implemented. For each instrumentable entity, InsectJ can provide various types of information associated with that entity. For example, for a predicate node, InsectJ provides information on the predicate outcome and the conditional statement containing the predicate.

Users can specify how the collected information is processed through using a set of predefined *monitors* in InsectJ. A monitor is a concrete class that consumes information reported by one or more probes inserted into code and suitably handles it. Figure 4-11 shows a wizard for creating a new monitor object for collecting runtime information using InsectJ. Users can choose the type of information to be collected by selecting the respective monitor. For example, we can select definition and use monitors to collect information reported for both field writes and reads.

The collection of execution traces for a program can be done through the use of the Java Debugging Interface (JDI). The Java Platform Debugger Architecture provides the JDI implemented by each Java Virtual Machine (JVM). The JDI allows collecting debugging and tracing information from a running Java program without instrumenting the source code. We use JDI to launch the execution of a program to trace its execution in a debug JVM. To collection runtime information, the JDI offers methods to select events of interest and provides callbacks which are executed when selected events occur. For our purpose of collecting execution trace, we utilize the events for statement execution. An event handler will be notified whenever a statement is executed. The event handler checks if the statement is the “end” statement of the program and records the trace through the program that has been exercised; otherwise the line number of the statement will be recorded.

New Concrete Monitor Wizard

A monitor object is used by probe inserters to call back the information. Each probe inserter specifies its own monitor interface to implement to be able to be called by that

Source folder:

Package:

Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☒ final ☐ static

Superclass:

Interfaces:

- ☒ edu.gatech.cc.rtinsect.monitor.interfaces.UseMonitorInter...
- ☒ edu.gatech.cc.rtinsect.MonitorObject
- ☒ edu.gatech.cc.rtinsect.monitor.interfaces.DefMonitorInter...

Monitors:

	name	description
☐	BasicBlockMonitorInterface	Registers every entry in a basic block.
☐	MethodCallMonitorInterface	Inserts a probe before each method...
☒	DefMonitorInterface	Every time a field is defined a call is ...
☒	UseMonitorInterface	Every time a field is used, a call is m...
☐	MethodEntryMonitorInterface	At the beginning of a method a call i...
☐	MethodExitMonitorInterface	At the end of every method, a call is...
☐	CastMonitorInterface	Inserts probes at cast sites in progr...
☐	BranchMonitorInterface	Calculates the distance for each bra...
☐	ConstructorCallMonitorInterface	No description
☐	ClassResetInterface	Adds the possibility to reset the class ...

☐ Add monitor factory method

Do you want to add comments as configured in the [properties](#) of the current project?
☐ Generate comments

Figure 4-11. Creating a new monitor object using InsectJ

4.2.3 Kaveri

Kaveri is a program slicing tool which is built on top of Soot, thereby utilizing all the program analysis libraries of Soot. Kaveri is developed by Ranganath [171] and his research group at the Kansas State University. As a program comprehension aid, Kaveri contributes the following features:

- **Slice Java programs by choosing slice criteria:** The user can pick the criteria, generate the program slice and view the slice using the “Java source editor” feature.
- **View the slice in the Java editor:** The part of the source code included in the slice is highlighted in the editor. This aids slice-based program comprehension.
- **Perform additive slicing:** “What program points are common to slices based on criteria b and c ?” is a common question during program comprehension. In Kaveri, the user can achieve this by associating the different highlighting schemes to slices based on b and c and viewing both the slices in the editor at the same time.
- **Support for program comprehension:** Understanding dependence relations between various program points help understand the generated program slice. Kaveri provides the following features to facilitate this understanding
 - Provide the *slice comprehension view* to view which program points in a Java statement are included in the slice.
 - Provide the *dependence history view* to facilitate the user to navigate the dependence-based path taken by the user and backtrack on it.
 - Support *path queries* that can be used to find sequences of program points that are related via a pattern of dependencies and other relations specified by a language such as regular expression.
 - Provide the “*scoped sliced*” based on scope specifications to understand the relation between certain program points which are independent of external influences.

4.3 Summary

In this chapter, we have presented our infrastructure to support the application of program analysis techniques augmented with empirical properties into software analysis, testing and maintenance. The infrastructure is a combination of various research works and open source tools. It is divided into two parts: EmAnalyzer and EmValidator systems.

EmAnalyzer is composed of Soot, PathGen and EmViewer. Soot is a research tool developed by the Sable group which analyzes Java programs and produces CFGs and other control and data dependency information. PathGen is implemented based on the basis path algorithm proposed by Poole [168]. PathGen generates two types of paths through a CFG including linearly independent paths and representative paths. EmViewer is based on the two existing research tools including the Dot graph from Graphviz and the Dot Driver provided by Soot. EmViewer provides various views for easily

visualizing of source code and program analysis information including: CFG View, Source Code View, Path View and Analysis Information View.

EmValidator provides TestGen, JTracer and Kaveri to facilitate the validation of empirical properties. TestGen assists in the test data generation process by using the open-source Choco constrain solver. Jtracer makes use of the Java Debugging Interface to instrument the source code and collects various runtime information including execution traces. Kaveri is a research tool from Kansas State University which provides program comprehension facilities through program slicing.

Chapter 5

INFEASIBLE PATH DETECTION

5.1 Introduction

Detection of infeasible program paths has been recognized as an important problem in the use of static program analysis [23, 64, 74], it is particularly useful for structural testing and coverage analysis. Structural testing, both path-based and data-flow testing involves three steps: (1) Identification of a set of paths that will meet the selected criterion; (2) Generating a set of test data that will cause the paths to be executed; (3) Executing the program with the test data. The paths generated in the first step are referred to as test requirements. Infeasible path detection plays the following important roles in structural testing.

1. For path-based testing, information on infeasible paths can be used to guide the algorithm which selects program paths to be tested. As such, infeasible paths can be removed from the test requirements.
2. In dataflow testing, if infeasible paths are not detected during static analysis, definition-use (def-use) pairs which lie on infeasible paths might be selected for testing. Weyuker [211] describes the difficulties in def-use testing as follows:

The problem was determining which of the definition/use associations or du-paths were executable [feasible]. This problem is encountered when using many program-based criteria, including statement and branch coverage, but is particularly acute for all-du-paths criterion since there are frequently a large number of unexecutable du-paths. In fact, we found that the unexecutable path problem, not the large number of required test cases, was the primary practical difficulty in using the all du-paths criterion.

Therefore, in structural testing, the ability to detect infeasible paths can save considerable effort, both manual and automatic, that is taken to generate test data for these paths. Typically, existing test data generation algorithms [30, 79, 121] do not consider infeasible paths.

Coverage analysis is used to assure the quality of a test suit. Given a test coverage criterion and the test requirements with respect to the criterion,

coverage analysis computes the number of paths in the test requirements that have been exercised by a test suite. This is the coverage degree of the test suite with respect to the coverage criterion. 100% coverage can rarely be achieved on a real program due to the existence of infeasible paths. If the test requirements contain infeasible paths, it is not obvious what the achievable coverage degree is; thus we could not know how much more testing is needed for a particular situation.

Figure 5-1. Coverage rate vs. Effort (Taken from [44])

In a survey of test coverage, Zhu et al. [231] conclude that, none of the path-based coverage criteria is applicable due to the possible existence of infeasible paths in the test requirements. This is also reflected in the industry standard for path coverage which is in the range of 80-90% [44, 71]. According to Hedly et al. [89], about 12.5% of all the paths are infeasible. As such, in general, it is impossible to achieve 100% coverage of all the paths. Even in some programs where there is no infeasible path, Cornett [44] has pointed out that to attain coverage approaching 100%, a lot more effort is required whereas the same effort might find more bugs in a different testing activity such as formal technical review. The author also argues that the highest level of testing productivity occurs when one can find the most failures with the least effort where effort is measured in term of time required to create test cases, execute them and analyze the test run results. Therefore, it follows that in industry practice, the strategy is to increase coverage as fast as possible, not to achieve 100% coverage. This idea is further illustrated in the diagram in Figure 5-1.

Although it is impossible to provide a method that is both sound and theoretically completed to detect infeasible paths, we have discovered that most of the infeasible paths follow four common code patterns. In this chapter, we present the patterns. Based on the recognition of the patterns, we further propose a

method for detecting infeasible program paths. Interestingly, our experiment shows that a large number of infeasible paths can be detected by the proposed method.

5.2 Infeasible path patterns

In this section, we characterize four code patterns and the type of infeasible paths caused by each one of them. Our experimental results show that infeasible paths detected in these code patterns constitute a very large proportion of infeasible paths.

5.2.1 Conflicting-decision pattern

Figure 5-2. Conflicting-decision pattern

We have discovered that, sometimes, actions to be performed under the same or completely different conditions could be implemented by separate independent selection constructs (if statements) with identical or complement conditions. This is referred to as **conflicting-decision pattern**.

Definition 5-1 – Conflicting-decision pattern. Let n_p and n_q be predicate nodes of selection construct in a CFG. We say that n_p and n_q follow conflicting-decision pattern if and only if the following two conditions are satisfied:

1. All the basis paths from n_p to n_q are def-clear paths with respect to variables referenced at n_p and n_q .

2. There exist two nodes n_p^{succ} and n_q^{succ} , where n_p^{succ} and n_q^{succ} are successors of n_p and n_q respectively, such that the branch predicates of (n_p, n_p^{succ}) and (n_q, n_q^{succ}) are syntactically identical.

Figure 5-2 gives an example of the conflicting-decision pattern. This example is taken from Soot [194]. We use ‘...’ to represent statements which are not important to the example. In this figure, branch predicates of (n_1, n_2) and (n_4, n_5) are syntactically identical.

The implementation of this pattern leads to some infeasible paths. Figure 5-3 illustrates the infeasible paths that are caused by conflicting-decision pattern. In the first type of infeasible paths illustrated in Figure 5-3(a) the branch predicates of branches at n_1 and n_4 are “C” and “not C” respectively; thus there exists no input values which can satisfy both branch predicates. Similarly, the second type of paths in Figure 5-3(b) is also infeasible.

Next, we shall formalize the property of infeasible paths caused by conflicting-decision pattern.

Figure 5-3. Infeasible paths in conflicting-decision pattern

Property 5-1 (Invariant) – Infeasible paths caused by conflicting-decision pattern. Let n_p and n_q be predicate nodes of selection construct in a CFG. If n_p and n_q follow conflicting-decision pattern then any path through the CFG that contains branches of n_p and n_q with syntactically different branch predicates is infeasible.

Proof. Let n_p and n_q be two predicate nodes of selection construct in a CFG that follow conflicting-decision pattern. As such, all the basis paths from n_p to n_q are def-clear paths with respect to variables referenced by n_p and n_q . Moreover, there exist a branch of n_p and a branch of n_q whose branch predicates are syntactically identical. Assuming that these branches are $e_1 = (n_p, n_p^{\text{succ1}})$ and $e_2 = (n_q, n_q^{\text{succ1}})$. Let $e_3 = (n_p, n_p^{\text{succ2}})$ and $e_4 = (n_q, n_q^{\text{succ2}})$ be the other branches of n_p and n_q respectively. As branch predicates of e_1 and e_2 are identical, any path which follows e_2 must also follow e_1 and vice versa. Therefore, any path which contains e_1 and e_4 is infeasible. Similarly, any path which contains both e_2 and e_3 is also infeasible. This completes the proof.

5.2.2 Mutually-exclusive-decision pattern

Empirically, we have discovered that a decision based on the same set of factors for one purpose is often implemented in a program using two approaches.

One alternative for implementing this type of decision is to use nested-if structure: the next condition will only be checked if the previous condition fails. Figure 5-4 illustrates this nested-if code pattern through an example taken from Crimson [45].

The second alternative avoids the use of nested-if structure. It formulates a set of mutually exclusive conditions based on the set of factors for making the decision so that a separate selection construct for each condition with only one non-null branch is used to jointly implement the decision. This design pattern is referred to as **mutually-exclusive-decision pattern**.

```
package org.apache.crimson.parser;
final class ContentModel{...
public boolean first (String token) {...
  if (content instanceof String && content == token)
    retval = true;
  else if (((ContentModel)content).first (token))
    retval = true;
  else if (next != null)
    retval = ((ContentModel)next).first (token);
  else
    retval = false;
  ...
}
```

Figure 5-4. Nested-if pattern

The implementation of this pattern leads to some infeasible paths in a CFG. Figure 5-5 illustrates this pattern through an example taken from Soot [194]. In this example, any path which contains more than one node from the set of nodes $\{n_2, n_4, n_6, n_8\}$ is infeasible. This is because the set of conditions at

nodes n_1 , n_3 , n_5 and n_7 are mutually exclusive; each value of n only satisfies at most one condition.

Though from code optimization and infeasible path avoidance viewpoint, the first alternative is clearly better, from our empirical observation, the mutually-exclusive decision pattern (second alternative) is frequently used. We believe this could be due to the latter alternative provides a neater code structure.

Before proceeding to the formalization of infeasible paths property in mutually-exclusive-decision pattern, we shall introduce some terms. A **prime variable** is a variable which is not defined through any other variable in the program. In Figure 5-5, n is a prime variable because n is an input, it is not defined through any other variables in the method.

Figure 5-5. Mutually-exclusive-decision pattern example and its CFG

Given a set of predicate nodes of selection construct $\{n_{p_1}, \dots, n_{p_n}\}$, the **set of external variables** of n_{p_j} , $1 \leq j \leq n$, contains all the variables which are defined by nodes that are control dependent on n_{p_j} and are referenced by nodes that are not control dependent on n_{p_j} .

Definition 5-2 – Empirical mutually-exclusive-decision pattern. Let $\{n_{p_1}, \dots, n_{p_n}\}$ be a set of predicate nodes of selection constructs in a CFG. If these predicate nodes satisfy the following conditions, then we say that $n_{p_1}, \dots, n_{p_n}$ are empirically mutually exclusive (e-mutually-exclusive):

1. n_{p_j} dominates $n_{p_{j+1}}$ and $n_{p_{j+1}}$ post-dominates n_{p_j} , $1 \leq j \leq n-1$.
2. n_{p_j} has only one successor that is control-dependent on n_{p_j} , $1 \leq j \leq n-1$.
3. Any basis path from n_{p_1} to n_{p_j} , $1 \leq j \leq n$, is a def-clear path with respect to the variables referenced at n_{p_j} .
4. The sets of prime variables that are referenced at n_{p_j} , $1 \leq j \leq n$, are identical.
5. The sets of external variables of all n_{p_j} , $1 \leq j \leq n$, are identical.

The first condition reflects that these predicate nodes define separate selections structures, which means they are not nested-if structures. The second condition ensures that each predicate node has only one non-null branch. The fourth condition reflects the fact that the decision implemented by these predicate nodes is based on the same set of factors. The last condition is to ensure that these predicate nodes are implemented for one purpose.

Property 5-2 (Empirical) – Infeasible paths caused by mutually-exclusive decision pattern. If $n_{p_1}, \dots, n_{p_n}$ are e-mutually exclusive predicate nodes of selection construct in a CFG then any path that contains nodes n_u, n_v such that n_u and n_v are control-dependent on n_{p_i} and n_{p_j} respectively ($1 \leq j, i \leq n, i \neq j$), is infeasible

The rationale behind Property 5-2 is the *uniqueness of decision* which means at most one decision is made at one point in time.

5.2.3 Check-then-do pattern

In many situations, a set of actions is only performed upon the successful checking of a set of conditions. The commonly used design method to implement the above-mentioned requirement is to separate the checking of conditions from the actions to be performed. The checking of conditions reports the outcome through setting a ‘flag’ variable. Actions are performed in a branch of a predicate node which tests the value of the ‘flag’ variable. This code pattern is referred to as **check-then-do** pattern. Many instances of this code pattern can be found in Soot [194].

Check-then-do pattern always introduces some infeasible paths. Figure 5-6 illustrates the check-then-do pattern through an example taken from Soot. Variable `remove` serves as a flag variable which is assigned to `true` initially. This variable is set to `false` if `synchBody.contains(pAs)` is true. After the checking, if value of `remove` is true (node n_5), node n_6 will be executed which adds `synchAs` to `toRemove`.

Figure 5-6. Check-then-do pattern

In this code segment, any path which contains nodes n_1 , n_4 and follows the ‘true’ branch at node n_5 , (n_5 , n_6), is infeasible because the predicate at node n_5 is always evaluated to false along the path. Similarly, any path which contains node n_1 but does not contain node n_4 and does not follow the ‘true’ branch at node n_5 is also infeasible.

Definition 5-3 – Empirical check-then-do pattern. Let v_k be a variable and n_p be a predicate node of selection construct. Let n_u be a node in the CFG. We say that n_u and n_p follow empirical check-then-do pattern (e-check-then-do) with respect to v_k if they satisfy the following conditions:

1. n_p only references to v_k
2. n_u dominates n_p and n_p post-dominates n_u
3. n_u assigns v_k to a constant which always results to satisfaction of the branch predicate of one branch of n_p

Next, we present an empirical property to realize the infeasible paths introduced by check-then-do pattern.

Figure 5-7. Two types of infeasible paths in check-then-do pattern

Property 5-3 (Empirical) – Infeasible paths caused by check-then-do pattern. Let n_p be a predicate node of selection construct in a CFG and $n_p^{succ_1}$, $n_p^{succ_2}$ be the two successors of n_p . Let n_u be a node in the CFG such that n_u and n_p follow the e-check-then-do pattern with respect to a variable v_k . If the value of v_k assigned at n_u always results to the satisfaction of $(n_p, n_p^{succ_2})$, then the following paths are infeasible:

1. Any path that contains n_u , n_p , $n_p^{succ_2}$ and the sub-path from n_u to n_p contains a node n_v that redefines v_k to a value syntactically different from the value assigned by n_u and no redefinition of v_k after this node in the sub-path.
2. Any path that contains n_u , n_p , $n_p^{succ_1}$ and the sub-path from n_u to n_p does not contain any node that defines/refines v_k to a value syntactically different from the value assigned by n_u .

The rationale of Property 5-3 is as follows. The given condition implies that the value of n_k defined/redefined at n_u always results to the satisfaction of the branch predicate of one branch of n_p . As n_v redefines v_k to a value syntactically

different from the value assigned at n_u it is likely that the values of v_k defined/redefined at n_v always results to the satisfaction of the branch predicate of the other branch of n_p . Therefore, v_k serves as a flag variable to report the result of checking with one reports “success” and the other reports “fail”. Then, the selection construct defined by n_p checks the value of v_k to determine what the actions to be performed. Figure 5-7 illustrates the two types of infeasible paths introduced by check-then-do pattern.

5.2.4 Looping-by-flag pattern

Figure 5-8. Looping-by-flag pattern

Sometimes, programmers use a “flag” variable to control the termination of a loop. Normally, the flag variable is initialized to a value which enables the execution of the loop. Inside the body of the loop, the flag variable will be set to another value if some conditions are satisfied. This value of the flag variable will lead to the termination of the loop. This coding pattern is often referred to as **looping-by-flag pattern**. Figure 5-8(a) describes the pattern through an example from Soot [194] and Figure 5-8(b) gives the corresponding CFG. In Figure 5-8, variable `change` serves as a flag variable to control the loop at node n_2 . Initially, node n_1 defines `change` to `true` which enables the entering of the loop at n_2 . Inside the loop, node n_4 redefines the variable `change` to `false` which leads to the termination of the while-loop at n_2 . As such, any path which contains the following path (begin, n_1 , $n_2 \dots n_3$, $n_4 \dots n_2$, n_3) is infeasible. Below, we formalize looping-by-flag pattern.

Definition 5-4 – Empirical looping-by-flag pattern. Let n_u , n_v , n_p and n_q be four nodes in a CFG in which n_p is a predicate node of iteration construct and n_q is a predicate node of selection construct. We say that n_u , n_v , n_p , n_q follow empirical looping-by-flag pattern (e-looping-by-flag) with respect to a variable v_k if they satisfy the following conditions:

1. The predicate at n_p only references v_k .
2. n_q is transitively control-dependent on n_p .
3. n_u dominates n_p and n_p post-dominates n_u .
4. n_u assigns v_k to a constant V_k which always results to satisfaction of the branch predicate of the entry branch of n_p .
5. n_v is control dependent on n_q and n_v assigns v_k to a value that is syntactically different from V_k .

Below, we formalize properties of infeasible paths cause by looping-by-flag pattern.

Property 5-4 (Empirical) – Infeasible paths caused by looping-by-flag pattern. Let n_u , n_v , n_p and n_q be four nodes in the CFG of the program in which n_p is a predicate node of iteration construct and n_q is a predicate node of selection construct. Let (n_p, n_p^{entry}) be the entry branch at n_p . If n_u , n_v , n_p , n_q follow e-looping-flag pattern with respect to a variable v_k , then any path that satisfies the following conditions is infeasible:

1. contains n_u , n_v and n_q
2. contains more than one appearance of n_p^{entry}
3. At least one sub-path from n_v to an appearance of n_p^{entry} is a def-clear path with respect to v_k

The rationale behind Property 5-4 is that if n_v assigned v_k to a value syntactically different from n_u , it is likely that the value will lead to the termination of the loop controlled by n_p . As such, a path in which there is no redefinition of v_k along the sub-path from n_v to an appearance of n_p^{entry} is likely to be infeasible.

According to Property 5-4, the following paths in Figure 5-8(b) are infeasible

$$:\{(\dots, n_1, n_2, \dots, n_3, n_4, \underbrace{\dots, n_2, n_3, \dots}_{\text{no redefinition of 'change'}}) \}.$$

5.3 Statistical validation

In this section, we report the results of our binomial testing [76] to statistically validate the empirical properties presented in the previous section including Property 5-2, Property 5-3 and Property 5-4. As Property 5-1 is an invariant

property, it is excluded from our statistical validation. For each empirical property $\mathcal{R}$, the alternate hypothesis states that the property holds for equal or more than 97% of the cases. And, therefore, the null hypothesis states that the property holds for less than 97% of the cases. That is:

H_0 (null hypothesis): $p(\mathcal{R} \text{ holds}) < 0.97$

H_1 (alternate hypothesis): $p(\mathcal{R} \text{ holds}) \geq 0.97$

The binomial test statistics z is computed as

$$z = \frac{X/n - p}{\sqrt{(p(1-p)/n)}},$$

where n is the total sample size for the test and X is the number of cases that support the alternative hypothesis H_1 . Taking 0.001 as the type I error, if $z > 3.1$, we reject the null hypothesis, otherwise, we accept the hypothesis.

Our samples are randomly drawn from the following nine Java systems spread across a wide variety of application domains:

1. *AVIEC* is a research prototyping developed by graduate students. AVIEC provides program analysis functions and automated verification of functional features in software systems.
2. *Taxier* is a taxi dispatching system developed by final year students. Taxier combines fuzzy logic and the linear assignment algorithm to provide effective real-time dispatching of taxis within a city.
3. *Tomcat* [8] is an open source, Java-based web application container that runs servlet and Java Server Page (JSP) web applications. Tomcat server is the reference implementation for the servlet and JSP specifications.
4. *JasperReport* [100] is an open source project which provides the necessary features to generate dynamic report including data retrieval using the Java Database Connectivity (JDBC) as well as support for parameters, expressions, variables and groups.
5. *Robocode* [178] is an easy-to-use robotics battle simulator that runs across all platforms supporting Java 2. Users create a robot, put it onto a battlefield and let it battle against opponent robots created by other users.
6. *HtmlParser* [195] is a simple but powerful Java library allowing analysis and manipulation of HTML documents.
7. *JHotdraw* [105] is a framework for the creation of drawing editors. It provides support for a range of programs from simple paint package style editors to more complex programs. The framework also supports the creation of geometric and user defined shapes, editing those shapes and creating behavioral constraints in the editor and animation.

8. *Jlibsys* [195] is an open source library management system. *Jlibsys* is designed to help simplify the task of managing a physical library consisting of individual inventory items such as books, tapes, optical media, etc.
9. *NCS* is a J2EE application developed by the National Computer System of Singapore. For confidential reason, we do not disclose the system's name here.

Table 5-1. Samples for testing empirical properties

System	KLOC	Source	P ₂	P ₃	P ₄
AVIEC	31.2	Student	67	51	21
Taxier	67.5	Student	98	74	35
Tomcat	313.5	Open source	59	42	32
JasperReports	175.6	Open source	43	23	14
Robocode	44.0	Open source	27	19	23
HtmlParser	61.6	Open source	121	134	87
Jhotdraw	71.7	Open source	34	51	47
JlibSys	12.2	Open source	32	19	36
NCS	147.4	Industrial	148	174	81
Total			629	587	376

HtmlParser and Jlibsys can be downloaded from Sourceforge [195] by searching for the project's name. A procedure in a system forms a case (an individual in the sample) for the testing of Property 5-2, Property 5-3 and Property 5-4. For each empirical property, we randomly draw a set of procedures from each system for testing the property. Table 5-1 gives the sample size for the testing of each property. The first two columns give the name of each system and the source from where the system is taken respectively. Columns P₂, P₃ and P₄ give the number of cases drawn from each system for the testing of Property 5-2, Property 5-3 and Property 5-4 respectively.

To automate the application of the empirical properties, we have implemented the InfeasibleDetector tool to detect infeasible paths in a set of paths based on the properties proposed in the previous section. The implementation details of this tool will be discussed later in Section 5.5.1. The InfeasibleDetector tool is implemented by extending EmAnalyzer.

For each empirical property $\mathcal{R}$, for each procedure in the sample for testing $\mathcal{R}$, EmAnalyzer is used to compute a set of paths through the CFG of the procedure such that for each path any loop included is iterated at most twice. For the purpose of validating the empirical properties, the algorithm findRPaths (Figure

4-6) implemented for the component PathGen of EmAnalyzer is slightly modified by allowing each loop to be iterated twice. This can easily be done by attaching a counter at the true branch of each loop. The maximum count should be two.

For each path p_x , InfeasibleDetector is used to check whether the path p_x can be concluded as infeasible by property $\mathfrak{R}$. If so, EmValidator assists us to confirm the infeasibility of the path. First the TestGen component of EmValidator tries to generate a test case for p_x . If such a test case can be generated, p_x is feasible; otherwise, TestGen displays the symbolic expression representing the path domain of p_x for solving manually. If still no test case can be generated, p_x is infeasible; otherwise, it is feasible. If the results produced by InfeasibleDetector are confirmed correct by TestGen for all the paths in the procedure, then we conclude that empirical property $\mathfrak{R}$ holds for the case.

All the cases in the sample for testing each empirical property gave affirmative result. Therefore, the z-score of binomial test for Property 5-2, Property 5-3 and Property 5-4 are 4.41 ($X = 629, n = 629, p = 0.97$), 4.26 ($X = 587, n = 587, p = 0.97$) and 3.41 ($X = 376, n = 376, p = 0.97$) respectively. As all the z-scores are greater than 3.1, we conclude that Property 5-2, Property 5-3 and Property 5-4 hold for equal or more than 97% of all the cases at 0.1% level of significance.

In our statistical validation, to check the feasibility of a path, we use TestGen which is based on symbolic execution and evaluation. In case that TestGen cannot generate any test case, we base on manual evaluation. Another way to check the feasibility of a path is by using dynamic execution. Dynamic techniques are based on the program's real execution where actual values are assigned to input variables and the program's execution flow is monitored. If the program's execution deviated from the intended path, techniques such as, genetic algorithms [28, 27] or iterative relaxation [78, 121, 122] are used to determine new values for the input variables which make the right branch to be taken. When a path is feasible, dynamic approaches prove to be fast and efficient. However, in the case of infeasible paths, dynamic approaches might have to go through a large number of iterations before they can come up with a conclusion [23]. Some of them terminate after a timeout period with no conclusion at all. In the worst case, some of them might falsely conclude that a path is infeasible. For the validation of the empirical properties, we choose symbolic execution because they give precise results even though symbolic execution is expensive and it might not work in some cases such as function calls and pointers.

Similar to any empirical research, surely, the result of our experiments is also associated with some threats to validity. Though much effort has been made in

our experiment on the size of sample source codes and their coverage in terms of domains and sources (from student projects to open-source projects to industrial projects), it is still far from covering the complete application domains and programmer behaviors. The latter is impossible even there is no constraint on time and resources. This may poses some limitations on the results of our experiment in term of generality.

5.4 Detecting infeasible paths

Detection of infeasible paths plays an important role in program analysis and software testing. Existing methods detect infeasible paths by attempting to solve the conjunction of branch predicates along a path. To achieve this, they propagate these nodes to related nodes along the path. The process will usually involve solving branch predicates that reference or transitively reference to input variables. It will also propagate through many other nodes along the path. As results, it leads to extensive symbolic computation and evaluation. Our empirical study discovers that most of infeasible paths follow the four patterns discussed in Section 5.2. Based on the recognition of the patterns through Property 5-1, Property 5-2, Property 5-3 and Property 5-4, we proposed in this section a method to detect infeasible paths in a very different way from existing methods.

5.4.1 The algorithms

Algorithm `detect_infeasible_paths` (Figure 5-9) is the main algorithm which detects infeasible paths in a set Π of paths in a procedure P_S . The algorithm sequentially calls `detect_CD`, `detect_e-MED`, `detect_e-CTD` and `detect_e-LBF` algorithms to detect infeasible paths introduced by conflicting-decision pattern, mutually-exclusive-decision pattern, check-then-do pattern and looping-by-flag pattern respectively.

Algorithm `detect_infeasible_paths`(procedure P_S , set of path Π)
Output: a set of infeasible paths in Π .
Begin
 1. Construct control flow graph $\mathcal{G}$ of P_S
 2. $I_1 = \text{detect_CD}(\mathcal{G}, \Pi)$
 3. $I_2 = \text{detect_e-MED}(\mathcal{G}, \Pi)$
 4. $I_3 = \text{detect_e-CTD}(\mathcal{G}, \Pi)$
 5. $I_4 = \text{detect_e-LBF}(\mathcal{G}, \Pi)$
 6. **return** $(I_1 \cup I_2 \cup I_3 \cup I_4)$
End

Figure 5-9. Detect infeasible paths in a given set of paths

Each of the algorithms in Figure 5-10, Figure 5-11, Figure 5-12 and Figure 5-13 consists of two steps. In the first step, the pattern is realized based on the definition using purely static analysis. In the second step, infeasible paths in the given set of paths are detected by using the properties of infeasible paths caused by the corresponding pattern. Below we describe the main steps of each algorithm.

```

Algorithm detect_CD ( $\mathcal{G}, \Pi$ )
Output: a set of infeasible paths in  $\Pi$  detected by identical-decision pattern.
Begin
//Step1
1.  $\Psi = \{n_p \mid n_p \text{ is a predicate node of selection construct}\}$ 
2.  $\mathfrak{I}_{CD} = \emptyset, \Pi_{\text{infeasible}} = \emptyset$ 
3. for (each pair  $(n_p, n_q), n_p \in \Psi, n_q \in \Psi$ ) do {
4.   if (all the basis paths from  $n_p$  to  $n_q$  are def-clear paths
       w.r.t variables referenced at  $n_p, n_q$ ) then{
5.     if (branch predicates of  $(n_p, n_p^{\text{true}}), (n_q, n_q^{\text{true}})$ 
         are syntactically identical) then
6.        $\mathfrak{I}_{CD} = \mathfrak{I}_{CD} \cup \{(n_p^{\text{true}}, n_q^{\text{false}}), (n_p^{\text{false}}, n_q^{\text{true}})\}$ 
7.     else if (branch predicates of  $(n_p, n_p^{\text{true}}), (n_q, n_q^{\text{false}})$ 
         are syntactically identical) then
8.        $\mathfrak{I}_{CD} = \mathfrak{I}_{CD} \cup \{(n_p^{\text{true}}, n_q^{\text{true}}), (n_p^{\text{false}}, n_q^{\text{false}})\}$ 
     }
   }
}
//Step2
9. for (each path  $p_x$  in  $\Pi$ ) do
10.  for (each pair  $(n_p, n_q) \in \mathfrak{I}_{CD}$ ) do
11.   if ( $p_x$  contains both  $n_p$  and  $n_q$ ) then
12.     $\Pi_{\text{infeasible}} = \Pi_{\text{infeasible}} \cup p_x$ 
13. return  $\Pi_{\text{infeasible}}$ 
End

```

Figure 5-10. Detect infeasible paths in conflicting-decision pattern

The basic idea of algorithm detect_CD (Figure 5-10) is to form in the first step the set $\mathfrak{I}_{CD}$ of pairs of nodes $(n_p^{\text{succ}}, n_q^{\text{succ}})$ such that if a path contains both n_p^{succ} and n_q^{succ} , it is infeasible. Once this has been done, infeasible paths can be easily detected in the second step.

This algorithm bases mainly on Property 5-1. For each pair (n_p, n_q) of predicate nodes that satisfy the first condition of conflicting-decision pattern, the algorithm determines the branches of n_p and n_q which are identical. If branch predicates of the true branch of n_p , (n_p, n_p^{true}) , and the true branch of n_q , (n_q, n_q^{true}) , are

identical, then according to Property 5-1, any path which contains (n_p, n_p^{true}) and (n_q, n_q^{true}) and any path which contains $(n_p, n_p^{\text{false}})$ and (n_q, n_q^{true}) are infeasible. As such, $(n_p^{\text{true}}, n_q^{\text{false}})$ and $(n_p^{\text{false}}, n_q^{\text{true}})$ are added to $\mathfrak{I}_{\text{CD}}$ (line 6). If the true branch of n_p and the true branch of n_q are not identical, similar checking is carried out for the true branch of n_p and the false branch of n_q to infer on the infeasibility of n_p and n_q (line 7, 8).

Algorithm detect_e-MED ($\mathcal{G}, \Pi$)

Output: a set of infeasible path in Π detected by mutually-exclusive-decision pattern.

Begin

//Step 1

1. $\Psi = \{n_{p_i} \mid 1 \leq i \leq n, n_{p_i} \text{ is a predicate node of selection construct, } n_{p_i} \text{ has only one successor which is control dependent on } n_{p_i}, n_{p_{i-1}} \text{ dominates } n_{p_i}, n_{p_i} \text{ post-dominates } n_{p_{i-1}}\}, \Pi_{\text{infeasible}} = \emptyset$
2. **for** (each predicate $n_{p_i} \in \Psi, 1 \leq i \leq n$) **do** {
3. **if** (all the basis paths from n_{p_i} to n_{p_i} are def-clear path w.r.t variables referenced at n_{p_i}) **then**
4. remove n_{p_i} from Ψ and **continue**,
5. $\Omega_{p_i} = \{\text{prime variables referenced at } n_{p_i}\}$
6. $\Phi_{p_i} = \{\text{external variables of } n_{p_i}\}$
7. **Partition** = $\{\Psi_k \mid \Psi_k \subseteq \Psi, n_{p_i} \in \Psi_k \text{ and } n_{p_j} \in \Psi_k \text{ iff } \Omega_{p_i} = \Omega_{p_j} \ \& \ \Phi_{p_i} = \Phi_{p_j}\}$

//Step 2

8. **for** (each path p_x in Π) **do** {
9. **for** (each $\Psi_k \in \text{Partition}$ which $|\Psi_k| > 1$) **do** {
10. $\mathfrak{I}_{\text{MED}} = \{n_p^{\text{succ}} \mid n_p^{\text{succ}} \text{ is control dependent on } n_p, n_p \in \Psi_k\}$
11. **if not** (p_x contains more than one element in $\mathfrak{I}_{\text{MED}}$) **then**
12. $\Pi_{\text{infeasible}} = \Pi_{\text{infeasible}} \cup p_x$
13. **return** $\Pi_{\text{infeasible}}$
- End**

Figure 5-11. Detect infeasible paths in mutually-exclusive-decision pattern

Algorithm detect_e-MED (Figure 5-11) first forms the set Ψ of predicate nodes of selection construct which satisfy the first three conditions of e-mutually-exclusive-decision pattern (line 1 to 6). The objective is that Ψ will then be partitioned into a set of subsets such that all the predicate nodes in each subset satisfy the last two conditions of e-mutually-exclusive-decision pattern (line 7). Therefore, each subset in the partition contains a set of e-mutually-

exclusive predicate nodes. This is the most important step of this algorithm. For each set of e-mutually-exclusive predicate node, Step 2 just applies Property 5-2 to each path in Π to determine its feasibility.

```

Algorithm detect_e-CTD ( $G, \Pi$ )
Begin
//Step 1
1.  $\Psi = \{n_p \mid n_p \text{ is a predicate node of selection construct, } n_p \text{ references only a}$ 
    $\text{single variable}\}, \mathfrak{I}_{\text{CTD}} = \emptyset, \Pi_{\text{infeasible}} = \emptyset$ 
2. for (each predicate  $n_p$  in  $\Psi$ ) do {
3.    $s_p = \text{the predicate at } n_p, v_p = \text{the variable referenced by } n_p$ 
4.   if (there exists a node  $n_u$  such that  $n_u$  defines  $v_p$  to a constant  $V_p$  and
    $n_u$  dominates  $n_p$  and  $n_p$  post-dominates  $n_u$ ) then {
5.      $value = \text{evaluate } s_p \text{ using } V_p$ 
6.     if ( $value$ )  $\mathfrak{I}_{\text{CTD}} = \mathfrak{I}_{\text{CTD}} \cup \{(v_p, n_u, n_p, n_p^{\text{true}})\}$ 
7.     else  $\mathfrak{I}_{\text{CTD}} = \mathfrak{I}_{\text{CTD}} \cup \{(v_p, n_u, n_p, n_p^{\text{false}})\}$ 
   }
}
//Step 2
8. for (each path  $p_x$  in  $\Pi$ ) do {
9.   for (each tuple  $(v_p, n_u, n_p, n_p^{\text{succ}}) \in \mathfrak{I}_{\text{CTD}}$ ) do {
10.    if ( $p_x$  contains  $n_u, n_p, n_p^{\text{succ}}$  and the sub-path from  $n_u$  to  $n_p$  is a
    def-clear path w.r.t  $v_p$ ) then  $\Pi_{\text{infeasible}} = \Pi_{\text{infeasible}} \cup p_x$ 
11.    if ( $p_x$  contains  $n_u, n_p, n_p^{\text{succ'}}$  and the sub-path from  $n_u$  to  $n_p$  contains a
    node  $n_v$  which redefines  $v_p$  and the sub-path from  $n_v$  to  $n_p$  is a
    def-clear path w.r.t  $v_p$ ) then  $\Pi_{\text{infeasible}} = \Pi_{\text{infeasible}} \cup p_x$ 
  }
}
12. return  $\Pi$ 
End

```

Figure 5-12. Detect infeasible paths in check-then-do pattern

Algorithm detect_e-CTD (Figure 5-12) first constructs the set $\mathfrak{I}_{\text{CTD}}$ of all tuples $(v_p, n_u, n_p, n_p^{\text{succ}})$ such that n_u, n_p follow the e-check-then-do pattern with respect to v_p and the value of v_p defined by n_u always lead to the execution of the branch (n_p, n_p^{succ}) (line 1 to 7). This set is used in Step 2 to detect infeasible paths as any path which satisfies Property 5-3 with respect a tuple in $\mathfrak{I}_{\text{CTD}}$ is infeasible. Basically the algorithm forms set $\mathfrak{I}_{\text{CTD}}$ as follow. For each predicate node n_p , line 5 performs constant substitution for predicate s_p at n_p by using V_p , which is the value assigned to v_p at node n_u . If v_p is evaluated to true, which means the value V_p assigned to v_p at n_u always lead to the satisfaction of the

branch (n_p, n_p^{true}) ; thus the tuple $(v_p, n_u, n_p, n_p^{\text{true}})$ is added to $\mathfrak{I}_{\text{CTD}}$ (line 6). Otherwise, the tuple $(v_p, n_u, n_p, n_p^{\text{false}})$, where $(n_p, n_p^{\text{false}})$ is the false branch of n_p , is added to $\mathfrak{I}_{\text{CTD}}$ (line 7).

Algorithm detect_e-LBF (Figure 5-13) constructs in the first step the set $\mathfrak{I}_{\text{LBF}}$ of tuples $(n_u, n_v, n_p, n_q, v_p)$ such that n_u, n_v, n_p and n_q follow e-looping-by-flag pattern with respect to v_p and the value V_p assigned to v_p at n_u always leads to the execution of the entry branch of the predicate node n_p . In the second step, for each tuple $(n_u, n_v, n_p, n_q, v_p)$ in $\mathfrak{I}_{\text{LBF}}$, the algorithm checks whether there is any path p_x which satisfies three conditions of Property 5-4 with respect to n_u, n_v, n_p, n_q and v_p (line 12). If so, p_x is infeasible; thus p_x is added to the set of infeasible paths detected $\Pi_{\text{infeasible}}$.

Algorithm detect_e-LBF (G, Π)

Begin

//Step 1

1. $\Psi = \{n_p \mid n_p \text{ is a predicate node of iteration construct, } n_p \text{ references only a single variable}\}$, $\mathfrak{I}_{\text{LBF}} = \emptyset$, $\Pi_{\text{infeasible}} = \emptyset$
2. **for** (each predicate n_p in Ψ) **do** {
3. $s_p =$ the predicate at n_p , $c_p =$ the variable referenced by n_p
4. **if** (there exists a node n_u such that n_u defines c_p to a constant V_p and n_u dominates n_p and n_p post-dominates n_u) {
5. **if** (there exist a node n_v which defines v_p to a value syntactically different from V_p) **then** {
6. $n_q =$ a node on which v_p is control-dependent
7. **if** (n_q is transitively control-dependent on n_p) **then** {
8. value = evaluate s_p using V_p
9. **if** (value) **then** $\mathfrak{I}_{\text{LBF}} = \mathfrak{I}_{\text{LBF}} \cup (n_u, n_v, n_p, n_q, v_p)$
- }
- }
- }
- }

//Step 2

10. **for** (each path p_x in Π) **do**
11. **for** (each tuple $(n_u, n_v, n_p, n_q, v_p)$ in $\mathfrak{I}_{\text{LBF}}$) **do**
12. **if** (p_x contains n_u, n_v, n_q and at least two appearances of n_p such that the sub-path from n_v to one appearance of n_p is a def-clear path w.r.t v_p) **then** $\Pi_{\text{infeasible}} = \Pi_{\text{infeasible}} \cup p_x$

13. return $\Pi_{\text{infeasible}}$

End

Figure 5-13. Detect infeasible paths in looping-by-flag pattern

5.4.2 Compare with existing approaches

Our approach consists of two steps, both of them are based purely on static program analysis techniques specifically control and data flow analysis. Only constant substitution is needed for algorithms detect_e-CTD and detect_e-LBF. Empirical properties of infeasible paths are the key features of our approach.

Many approaches rely on symbolic evaluation [23, 74, 229]. To check whether a path is infeasible, the path is symbolically executed to generate a symbolic expression representing the path. This expression is then solved by a constraint solver to determine the infeasibility of the path. Symbolic execution is expensive in both speed and space. In opposite to symbolic approach, we do not need to symbolically execute the path. Only simple constant substitution is needed. Basically, our approach checks whether the path falls into any type of infeasible paths caused by the four code patterns. As the proposed property of infeasible paths relies mainly on control and data dependency information, the checking can easily be done by going through all nodes in the path to make sure that they follow certain control and data dependency relationships.

The performance of our approach is comparable to those approaches which are based on heuristics that have been empirically validated [64, 222]. However, we improve on the earlier approaches in that we provide four empirical properties (heuristics) which cover a large portion of infeasible paths.

5.4.3 Time complexity

The cost of our technique can be divided between the code pattern recognition (Step 1) and infeasible paths detection in each code pattern (Step 2). Algorithm detect_CD takes $O(N_p^2 B)$ computation steps to find the set $\mathfrak{I}_{CD}$, where N_p is the number of predicate nodes of selection construct in the CFG and B is the number of basis paths through the CFG. According to the McCabe [141] complexity number, B is equal to $(|E| - |\mathcal{N}| + 2)$ where $|\mathcal{N}|$ is the number of nodes and $|E|$ is the number of edges of the CFG. Therefore, the complexity of the first step is $O(N_p^2 |E|)$. In Step 2 of the algorithm, all the infeasible paths in a given set Π can be detected in $O(|\Pi| |\mathfrak{I}_{CD}|)$, where $|\Pi|$ is the number of paths in Π and $|\mathfrak{I}_{CD}|$ is the number of pairs in $\mathfrak{I}_{CD}$. The number of elements in $\mathfrak{I}_{CD}$ is generally less than N_p^2 ; As such, the overall complexity of algorithm detect_CD is $O(N_p^2 * \max(|E|, |\Pi|))$.

Similarly, the cost of the first step and the second step of algorithm detect_e-MED is $O(N_p |E|)$ and $O(N_p |\Pi|)$ respectively. The overall complexity of the algorithm is $O(N_p * \max(|E|, |\Pi|))$. The cost of finding infeasible paths in algorithm detect_e-CTD is $O(N_p |\mathcal{N}|)$ and $O(N_p |\Pi|)$ for the first step and the

second step respectively; thus the overall complexity is $O(N_p * \max(|\mathcal{M}|, |\Pi|))$. Algorithm `detect_e-LBF` takes $O(N_p |\mathcal{M}|^2)$ for the first step and $O(N_p |\Pi|)$ for the second step. Overall, the algorithm detects all infeasible paths in loop-by-flag pattern in $O(N_p |\mathcal{M}|^2)$ computation steps.

5.5 Evaluation

5.5.1 The prototype system

We have implemented a prototype tool name `InfeasibleDetector` following the approach proposed in Section 5.4 to detect the infeasible paths in a set of paths extracted from Java programs. `InfeasibleDetector` uses various program analysis and path generation functions provided by `EmAnalyzer`. As having mentioned in Section 4.1, `EmAnalyzer` integrates various existing research work and open-source tools. The architecture of `InfeasibleDetector` is depicted in Figure 5-14. The `InfeasibleDetector` consists of four components implementing the four algorithms given in Figure 5-10, Figure 5-11, Figure 5-12 and Figure 5-13, respectively. The implementations of these algorithms are quite straightforward which are based mainly on the control dependence and dataflow information provided by `EmAnalyzer`.

In algorithms `detect_e-CTD` and `detect_e-LBF`, we need to perform constant substitution for predicates which reference to only a single variable (line 5 in Figure 5-12 and line 8 in Figure 5-13). Currently, only simple constant substitution algorithms are implemented in the tool including evaluation for arithmetic operations and bitwise operations. However, we emphasize that our infeasible path detection technique supports the analysis of arbitrary predicates.

Figure 5-14. `InfeasibleDetector` tool

5.5.2 Experiment

According to the Tiobe programming index [1], Java is the most popular programming language. As such, to evaluate the effectiveness of the proposed approach we have applied the approach to detect infeasible paths in many programs written in Java. There are two main benefits of choosing Java as the main programming language for our experiments. Firstly, we can collect a richer set of sample Java programs from various application domains. Secondly, we have found many program analysis tools for Java programs. These tools provide strong supports which further facilitate our experiments. Supplementary experiments on C programs are provided in Section 5.6.

Table 5-2. Descriptions of target systems

System	kLOC	Domain	Source	Packages
GraphAlgo	6	Algorithm	NTU Msc project	add; cmst; drop
JTrade	8	Trading	Open-source	sfljtse sfljtse.quotes sfljtse.stats
Crimson	30	Text editor	Open-source	org.apache.crimson.*
InsectJ	40	Program instrumentation	GIT research group	edu.gatech.cc.rtinsect.*
DFNET	48.5	Software Design	NTU PhD project	ntu.dfnet ntu.dfnet.graph
JGEditor	122	Graph editor	Industrial project	ncs ncs.graph ncs.plaf.basic
Soot	220	Software Analysis	McGill University research group	soot.coffi ; soot.dava.internal.AST soot.dava.toolkits.base* soot.javaToJimple.* soot.jimple soot.jimple.parser soot.jimple.spark.solver soot.jimple.toolkits.*

Legends: NTU – Nanyang Technological University
GIT – Georgia Institute of Technology

The sample programs for our experiments are real programs which have been carefully chosen to minimize all the threats to validity and to maximize the coverage of the experiment. More specifically, our sample programs are taken from three main sources including (1) academic/research institute, (2) industrial project and (3) open-source. This ensures that the sample programs are written

by programmers of different expertise levels. Software taken from the first source is normally developed by undergraduate students, master/PhD students and researchers. Software taken from the industrial projects is generally developed by junior and senior software engineers of two to three years of experience. For open-source software, it is however impossible to access the expertise level of the developers as this information is not available. In this case, we ensure that no two sample programs are taken from the same group of developers. Furthermore, all the sample programs are selected from different application domains with reasonable size.

Table 5-2 lists all the systems in the increasing order of their size (kLOC). Columns “Domain” and “Source” give the application domain and source from where the sample programs were taken. DFNET is developed by PhD students for the graphical dataflow analysis. GraphAlgo is a tool written by a group of Msc students to aid in the teaching of data structures and algorithm. InsectJ [185] is a java byte code instrumentation framework implemented by researchers from Georgia Institute of Technology. JTrade [195] and Crimson [45] are both open source systems. JTrade is trade list management system while Crimson is a Java XML parser. JGEditor is a graph editor provided by our industrial partner which has been developed by a group of senior programmers. Soot [170] is a well-known Java bytecode analysis and optimization tool developed and maintained by researchers from the Sable group, McGill University.

For each system, we randomly pick some packages for the experiment. After that, we carefully choose in each package only methods which contain infeasible paths. For each such method, the PathGen is used to generate a set Π of complete paths such that for each path, any loop included is iterated at most twice. We then invoke the proposed algorithms to detect all the infeasible paths in Π .

Table 5-3. False-positive and true-negative identification

EmValidator & Manual	InfeasibleDetector	Conclusion
Feasible	Infeasible	False-positive
Infeasible	Feasible	True-negative

For each path $p_x \in \Pi$, we determine the feasibility of the path with the assistance of the EmValidator tool. First, the path is input into the TestGen component of EmValidator. TestGen tries to generate a test case for p_x . If such a test case can be generated, EmValidator concludes that the path is feasible; otherwise, TestGen displays a symbolic expression representing p_x for manual solving. If still no test case can be generated, p_x is infeasible; otherwise, it is feasible. The false-positive and true-negative cases are then identified according

to Table 5-3. If p_x is feasible and the InfeasibleDetector tool concludes that it is infeasible, p_x is counted as a *false-positive case*. If p_x is infeasible and the prototype tool could not detect it, p_x is counted as a *true-negative cases*.

5.5.3 Results

Table 5-4 shows the results of our experiments on seven systems. The first column gives the name of each system. Columns Σ , Σ_{truly} , Σ_{detected} give the total number of paths taken into consideration, the number of truly infeasible basis paths and the number of infeasible basis paths detected by the InfeasibleDetector respectively. Columns Σ_{FP} and Σ_{TN} give the number of false positive cases and the number of true negative cases respectively. Finally, columns Σ_1 , Σ_2 , Σ_3 and Σ_4 give the number of infeasible paths detected by Property 5-1, Property 5-2, Property 5-3 and Property 5-4 respectively.

According to Table 5-4, InfeasibleDetector detects 1873 infeasible paths. There is no false positive case. As such the proposed approach successfully detects 82.3% of all the infeasible paths. The breakdown of percentage of infeasible paths detected by each property over all the infeasible paths detected is as follows: 33.2% are detected by Property 5-1, 21% are detected by Property 5-2, 44.6% are detected by Property 5-3 and only 1.2% is detected by Property 5-4.

Table 5-4. Experimental results

System	Σ	Σ_{truly}	Σ_{detected}	Σ_{FP}	Σ_{TN}	Σ_1	Σ_2	Σ_3	Σ_4
Soot	3240	1161	989	0	172	326	139	509	15
DFNET	438	222	198	0	24	132	32	30	4
InsectJ	275	76	76	0	0	12	14	50	0
Jtrade	216	112	77	0	35	65	0	12	0
GraphAlgo	405	196	146	0	50	0	41	102	3
Crimson	918	330	223	0	107	51	62	110	0
JGEditor	471	179	164	0	15	36	105	23	0
TOTAL	5963	2276	1873	0	403	622	393	836	22

Also from Table 5-4, there are 403 true negative cases, which accounts for 17.7% of all the truly infeasible paths. We investigate all the true negative cases and found that 95 cases, which accounts for 4.2% of all the truly infeasible paths, are not detected by the proposed approach at all. By “not detected by the proposed approach at all” we mean that those cases do not follow any of the four proposed code patterns. These code patterns are very rare. Some of them are highlighted in Figure 5-15 and Figure 5-16. Currently, we are working on explaining the intention of the programmers in these code patterns. We are also conducting more experiments to search for the instances of these code patterns.

The code pattern in Figure 5-15 leads to some infeasible paths. For example, line 3 and line 12 assign sRes and actual to the same value, which is Public. Consequently, predicate !sRes.equals(actual) at line 19 is always evaluated to false along any path which contains lines 3, 12 and 19; thus any path which contains lines 3, 12 and follows the true branch at line 19, (19, 20), is infeasible.

At a glance, predicates at lines 1, 3, 7 in Figure 5-16 look similar to *e*-mutually-exclusive-decision pattern. However, since the set of prime variables referenced in lines 1, 3, 7 are {dest, this, other}, {des, this}, {dest, other} respectively, these predicate does not follow *e*-mutually-exclusive-decision pattern. However, this code pattern also leads to some infeasible paths. Line 2 is only executed only if both predicates dest != this and dest != other are evaluated to true. Therefore, if line 2 is executed, the predicates at lines 3 and 7 are also true. As such, any path which contains line 2 but does not contain line 4 or/and line 8 is infeasible.

```

package soot.jimple.toolkits.annotation.qualifiers;...
public class TightestQualifiersTagger
    extends SceneTransformer {...
private void handleFields(){...
1.   String sRes = "Public";
2.   if (result == RESULT_PUBLIC){
3.       sRes = "Public";}
4.   else if (result == RESULT_PROTECTED){
5.       sRes = "Protected";}
6.   else if (result == RESULT_PACKAGE){
7.       sRes = "Package";}
8.   else if (result == RESULT_PRIVATE){
9.       sRes = "Private";}

10.  String actual = null;
11.  if (Modifier.isPublic(f.getModifiers())){
12.      actual = "Public";}
13.  else if (Modifier.isProtected(f.getModifiers())){
14.      actual = "Protected";}
15.  else if (Modifier.isPrivate(f.getModifiers())){
16.      actual = "Private";}
17.  else {
18.      actual = "Package";}

19.  if (!sRes.equals(actual)){
20.      ...
    }...
}...

```

Figure 5-15. A case not detected by the proposed approach (1)

The rest 308 true negative cases are not detected because of the limitations of the current InfeasibleDetector tool. In the current tool, for algorithms detect_e-CTD and detect_e-LBF, we only implemented simple constant substitution for

predicates with arithmetic and bitwise operations. As such, code segment like the one in **Figure 5-17** cannot be detected.

```

package soot.toolkits.scalar;...
public abstract class AbstractFlowSet
    implements FlowSet {...
public void union(FlowSet other, FlowSet dest) {
1.   if (dest != this && dest != other)
2.       dest.clear();

3.   if (dest != this) {
4.       Iterator thisIt = toList().iterator();
5.       while (thisIt.hasNext())
6.           dest.add(thisIt.next());
    }

7.   if (dest != other) {
8.       Iterator otherIt = other.toList().iterator();
9.       while (otherIt.hasNext())
10.          dest.add(otherIt.next());
    }
}...

```

Figure 5-16. A case not detected by the proposed approach (2)

```

package soot.dava.toolkits.base.AST.transformations;...
public class ForLoopCreationHelper{...
private List createNewStmtSeqNodeAndGetInit(List commonVars){
1.   List stmts = new ArrayList();
    ...
2.   int stmtNum=0;
3.   while(stmtNum<currentLowestPosition && stmtIt.hasNext()){
4.       stmts.add(stmtIt.next());
5.       stmtNum++;
    }
6.   if(stmts.size()>0){
7.       newStmtSeqNode = new ASTStatementSequenceNode(stmts);
    }
    else{
8.       newStmtSeqNode = null;
    }...
}...

```

Figure 5-17. A case not detected by the prototype tool

Nodes 1, 4 and 6 in the code segment actually follow e-check-then-do pattern with respect to `stmts.size`. Indeed, when `stmts` is initialized to a new `ArrayList` at node 1, variable `stmts.size` is assigned to zero. When a new element is added to change at node 4, `stmts.size` increases by one. Later on, node 6 checks `stmts.size` against zero. However, in this case, the tool does not possess such knowledge. In addition, the tool was not able to evaluate the predicate at 6; as such, it failed to detect infeasible paths in the code segment.

Currently, we are working on extending the tool to improve these weaknesses. Knowledge about predefined functions can be stored in a knowledge base and updated by users.

5.6 ArisPath: An infeasible path detector for C

In this section, we describe a tool that has been implemented to detect infeasible paths for programs written in C language. We also report the results of our experiments done on C programs to access the usefulness of the proposed approach.

We have developed ArisPath based on the proposed approach to infeasible path detection presented in Section 5.4. ArisPath makes use of the Aristotle tool [82], which is a static program analysis tool developed by researchers at Georgia Tech University. The tool is freely available at <http://www.cc.gatech.edu/aristotle/>.

Aristotle can extract CFG information and definition-use information for a given C program. Our tool, ArisPath, reads the information generated by Aristotle, and outputs infeasible paths according to four proposed patterns. ArisPath is written in Java and consists of the following classes:

- **ArisPath**: the entry point of the tool.
- **Program**: the class for modeling a C program. It manages a list of functions and provides methods for creating control flow table and definition-use table and for checking infeasible paths in a program.
- **Function**: the class for modeling a C function, which manages attributes such as the control flow of the function and the variables within the function. This class also provides methods for detecting infeasible path patterns within the function.
- **Node**: the class for modeling a single statement in a C program. It maintains statement related information such as statement type, neighbors of the statement, variables defined and used by the statement.
- **SourceCode**: this class maintains the source code of a C program.
- **Variable**: this class maintains the definition-use information of a variable.
- **Pair**: this is a supporting class that models a pair of nodes that have control or data dependency relationships.

The current version of the tool outputs infeasible paths among the *basis paths*. We repeat the experiment presented in the previous section with the use of

ArisPath on the four C programs listed in Table 5-5, all the programs are taken from the Software-artifact Infrastructure Repository (SIR [191], available at <http://sir.unl.edu>):

Table 5-5. Sample program taken from SIR

Program	kLOC	Description
gzip	5	A UNIX file compression tool
grep	10	A UNIX tool for searching a file for a pattern
flex	12	A UNIX scanner generator
sed	18	A UNIX stream editor

Table 5-6 shows the results of our experiments on four popular C programs. ArisPath detects 1678 infeasible paths out of 1799 truly infeasible paths in the four programs. There is no false positive case. As such, the proposed approach successfully detects 93.3% of all the infeasible paths. The breakdown of percentage of infeasible paths detected by each pattern over all the infeasible paths detected is as follows: 37.7 % are detected by Property 5-1, 25.0% are detected by Property 5-2, 25.3% are detected by Property 5-3 and 12% are detected by Property 5-4.

Table 5-6. Experiment results with C programs

Program	Σ	Σ_{truly}	Σ_{detected}	Σ_{FP}	Σ_{TN}	Σ_1	Σ_2	Σ_3	Σ_4
flex	1463	715	609	0	106	359	58	81	111
grep	1862	534	519	0	15	149	193	172	5
gzip	859	285	285	0	0	27	129	44	85
sed	1353	265	265	0	0	98	40	127	0
TOTAL	5537	1799	1678	0	121	633	420	424	201

Table 5-6 also shows that there are 121 true negative cases which account for 6.7% of all the truly infeasible paths. We have also investigated all the true negative cases and found that in this experiment, all the cases are not detected because of the limitations of the ArisPath tool. Currently, ArisPath only supports branch predicates of the form $(v_1 \text{ relop } v_2)$ where v_1 and v_2 are scalar variables or constants of primitive data types and *relop* must be arithmetic or bitwise operations. Our infeasible path detection technique, however, supports the analysis of arbitrary predicate expression. Despite the limitation of the ArisPath tool, the results obtained are very promising which prove that the proposed approach is applicable in the industrial testing practice.

Table 5-7 compares the experiment results obtained with Java programs in Section 5.5.3 and the ones obtained with C programs in this section. According

to Table 5-7, experiment results obtained with Java and C are quite consistent. More specifically, there is no false positive case detected in both experiments. They both show that Property 5-1 detects the most number of infeasible paths, followed by Property 5-3, then Property 5-2. Property 5-4 always detects the least number of infeasible paths. More importantly, both experiments give evidence that the proposed approach can be applied in both programming languages to detect more than 80% of all the infeasible paths. As such, the proposed approach is well suited in the industrial testing practice which requires only 80% to 90% coverage of program paths.

Table 5-7. Comparison of experiment results with Java and C

Language	$\sum_{FP}$	%TN	%detected				
			Total	%P1	%P2	%P3	%P4
Java	0	17.7	82.3	33.2	21	44.6	1.2
C	0	6.7	93.3	37.7	25	25.3	12

Legends: $\sum_{FP}$ - number of false positive cases

%TN – percentage of true negative cases

%detected – percentage of truly infeasible paths detected

%P1, %P2, %P3, %P4 – percentage of infeasible paths detected by Property 5-1, Property 5-2, Property 5-3 and Property 5-4 respectively.

5.7 Summary

In this chapter, we have proposed an empirical approach to identify infeasible paths in four common code patterns through realizing some common properties of these paths from source code. We have formulated four properties of infeasible program paths in four code patterns. Our binomial testing shows that all the proposed empirical properties of infeasible paths hold for 97% of all the cases at 0.1% level of significance. Based on these patterns, we have developed five algorithms and the InfeasibleDetector tool for infeasible path detection in Java programs and ArisPath for infeasible path detection in C programs. We have also conducted experiments to evaluate the effectiveness of the proposed approach. Even with some limitations in the current prototype tools, the proposed approach accurately detected 82.3% of all the infeasible paths in seven Java systems and 93.3% of all the infeasible paths in four C programs. Comparisons of the experiment results obtained with C and Java programs show the consistent of the proposed approach across different languages.

Our investigation into the infeasible paths undetected during our experiment reveals that most of the cases are due to the limitation of the prototype systems (InfeasibleDetector and ArisPath). During the implementation of these tools, we only implement constant substitution for simple branch predicates of the form ($v_1 \text{ relop } v_2$) where v_1 and v_2 are scalar variables or constants of primitive data types: integer, real, string and Boolean and *relop* must be arithmetic or bitwise

operations. As such, the performance of these tools can be significantly improved by enhancing the implementation of constant substitution. More specifically, complex expressions should be supported, especially expressions with function calls. The handling of function call can be quite challenging. In general, there are two approaches to handle function call: the **macro-expansion** approach and the **lemma** approach [133]. In the macro expansion approach, the source code of the called method is inlined within the caller program. However, this approach requires access to source code of the called function. The lemma approach requires that the called function must already be symbolically executed, which means all the possible return values together with their constraints must be known. This obviates the need to symbolically execute all paths in the called function and does not require the access to the called method's source code. In the future, we plan to implement a *demand-driven* approach to solving the function call problem for constant substitution. That means, if the function is called the first time, the function is symbolically executed to determine all the possible return values. After the first time, the function will be marked as "*analyzed*". Any further call made to the function, the lemma approach is applied. We also explore the use of a third approach which allows methods to be classified as "trusted": a trusted function $f(x)$, will be handled without expansion. The constant substitute engine obtains a value for $f(x)$ for a particular x by calling $f()$ directly. This is the implicit approach for operator such as '+'. It is also appropriate approach for calls to 'standard' library functions and functions whose source code is not available.

Another way to solve the function call problem is to treat the path as an interprocedural path. Indeed, when a statement contains a function call, the control and data flow have actually crossed the border of the calling function. Currently, the prototype system only processes intraprocedural paths; thus, any function call statement is ignored. To support interprocedural program analysis, new patterns should be discovered, examined and implemented so that the system can detect also interprocedural infeasible paths. We are investigating the relationship between control-coupling and interprocedural infeasible path. In software design, coupling is a measure of interprocedural connectivity. Two procedures are control-coupled if (1) one passes data to the other that is used in the latter procedure to determine the actions to be performed or (2) they use shared data to determine actions to be performed. This control-coupling clearly suggests some common properties of interprocedural infeasible paths.

This study, like any other empirical study, has some limitations. The binomial tests presented in Section 5.5.2 for validating the empirical properties are influenced by our choice of sample sets and the subject applications. Even though we have randomly chosen systems from various application domains

which are developed by software engineers of different level the chosen systems might not be the representatives of the population. This means there might be a chance that the proposed technique could then classify some paths as infeasible while in fact they are not. However, we surmise that our conclusions will generalize since they are based on results that are consistent over a population of sample sets that are greatly varied in size.

Moreover, as our statistical validation gives evidence that all the empirical properties hold for 97% of all the cases at 0.1% level of significance, the proposed approach is certainly applicable in industrial testing practice which requires only 80% to 90% coverage of program paths [44, 71]. The proposed approach can also be used to improve the accuracy of coverage analysis of a test suite to a very high level of accuracy as current path-based coverage analysis ignores infeasible paths totally.

Chapter 6

AUTOMATED RECOVERY, TESTING AND MAINTENANCE OF INPUT ERROR CORRECTION

6.1 Introduction

Database applications constitute one of the largest and most important software domains in the world. For example, database applications have been implemented to create electronics journals [140], scientific data repositories [145, 198], electronic commerce applications [80] and national science digital libraries [98]. In a database application, a major and important component is processing input submitted from users to update its databases and deliver information through printing reports or displaying on interactive media; these are referred to as the external effects raised by the application. It should come to no surprise that people make errors in the use of computer systems [120, 177]. User input errors, however, can be serious which might lead to severe consequences after the execution of the system such as financial uncertainty, disruption to communication and corporate instability. A study done by Brown [26] reports that “in early 2001, Microsoft suffered a nearly 24-hour outage in its Web properties as a result of human error made while configuring a name resolution system. Later that year, an hour of trading on the NASDAQ stock exchange was disrupted because of a technician’s mistake while testing a development system”. Although it is impossible to design systems in which people do not make errors, much can be done to minimize the effect of error, to maximize error detection and to make easier error correction or error recovery.

User inputs errors are very complicated which can take several forms, not only incorrect values submitted by users. According to Wright et al.[219], user input errors can be classified into: error of omission, error of commission and value error. **Error of omission** occurs when an input item is missed out during input submission. For example, a customer intended to purchase ten different products. However, when submitting the order, he missed out one product. **Error of commission** occurs when an additional input item is submitted by

mistake. For example, the entering of an extra valid product in a customer order is such a user input error if the item was entered by mistake. **Value error** occurs when incorrect values have been submitted. For example, instead of submitting the shipping address *X*, the customer erroneously submitted shipping address *Y*.

Currently, input validation [88] is the most popular means to enforce the accuracy of the user inputs to a system. Unfortunately, many user input errors can only be detected after the completion of execution. Take for example the entering of an extra valid product in a customer order, there is no means to detect and reject the valid extra item if it was submitted by mistake. We call the user input errors which are only discovered after the execution of a system as **post-execution input errors**. Within the scope of this thesis, we only consider post-execution input error; thus, we will use the term “user input error” to refer to a post-execution user input error.

Error correction may seem an important supplementary safety goal since total error prevention is difficult (if not possible) to achieve. **Input error correction** refers to a bundle of system functionalities which allow users to correct erroneous effects raised due to input errors. The provision of input error correction is extremely important in any software system since input errors are unavoidable. Recovery, testing and maintenance of input error correction are complex, time-consuming and error-prone tasks as there are many types of input errors and each of which may cause a number of erroneous effects, yet they are important tasks to assure the reliability of the feature.

To test the provision of input error correction for a type of input error, test cases must be generated to enforce the input error, test cases must also be generated to exercise the error correction implemented. Moreover, to test software more accurately, both code and specification behaviors must be covered during testing. Unfortunately, most existing techniques are either solely code-based or specification-based through adopting structural or functional testing strategies, respectively.

The first step towards maintaining a certain feature is to understand how the feature is implemented in a system. This is a critical problem in program understanding, especially when the understanding is directed to a certain goal like maintaining the feature. Before being able to understand a feature, one has to locate the code which implements the feature. Unfortunately, most of the systems have a large number of components containing hundred thousand lines of code. Therefore, it is not obvious which component implements a given feature. In addition, the nature of software development implies that many new or modified input error correction features to be incorporated are only

recognized during maintenance. At this stage, the maintainers are facing the pressure to modify the system to include new feature as quick as possible without introducing any adverse impact to the existing code.

We observed through empirical study that input error correction is often implemented through a few methods. The use of these methods results in some common properties of programs which implement the input error correction. By realizing these properties from source code, input error correction implemented by the system can be recovered. In this chapter, we propose an approach to automated recovery of input error correction implemented by a database application. We also explore the use of recovered information in testing and maintenance of input error correction. More specifically, we propose an approach to test input error correction which combines both code and specification characteristics. We also introduce the effect-oriented decomposition slicing technique to assist software engineers in maintaining input error correction feature for database applications without introducing any ripple effect.

The rest of this chapter is organized as follows. Section 6.2 describes an example application which will be used through out this chapter. Section 6.3 forms a theory for inferring input error correction implemented in a database application. Based on this theory, Section 6.4 proposes an approach to the automated recovery of input error correction. Section 6.5 describes the integration of functional and structural testing for input error correction. Section 6.6 discusses the use of the recovered information to aid in the maintenance of this feature. Section 6.7 presents a tool for the automated recovery, testing and maintenance of input error correction features. Section 6.8 reports the statistical validation results and two case studies to evaluate the usefulness of the proposed testing and maintenance approaches. Section 6.9 gives a comparison with related work. Section 6.10 concludes the chapter.

6.2 An example application

In this section, we describe a simplified real web application that will be used in all the examples throughout this chapter. All the procedures are presented in pseudo-code. E-Order is an online order processing system which consists of four procedures: `process_order`, `cancel_item`, `cancel_order` and `change_header`.

The four procedures interact with a database which consists of the following three tables:

Product = (Product-No, Product-Description, Qty-Sold)
 Order = (Order-No, Customer-No, Address, Date)

Order-Item = (Order-No, Product-No, Qty-Order, Qty-Delivered)

Figure 6-1. Procedure process_order and its CFG

Figure 6-2. Procedure cancel_item and its CFG

The process_order procedure (**Figure 6-1**) processes an order header and a list of order items submitted by user. Line 1 queries the database to check whether the order header submitted is already in the database. If the order header submitted is new (Line 2), Line 3 inserts the submitted order header into the Order table with all the required attributes. Line 4 iterates through all the order items submitted. For each order item, line 5 updates the Qty-Sold attribute in the Product table and line 6 inserts the order item into the Order-Item table.

The cancel_item procedure (

Figure 6-2) takes in an order number (inp_orderNo) and a product number (inp_productNo). Line 1 selects from the database an order item whose order number and product number is equal to inp_orderNo and inp_productNo, respectively. The procedure then checks whether such an order item exists and that no product has been delivered (line 2). If so, it updates the Product table on the Qty-sold of the order item in line 3 and removes the order item from the Order-Item table. Note that an ordered item can only be deleted if no product has been delivered.

Figure 6-3. Procedure cancel_order and its CFG

The cancel_order procedure (Figure 6-3) takes in an order number (inp_orderNo) as input. The procedure first selects the respective order from the database (line 1) and its belonging order items (line 2). If there exists such an order and the list of order items is empty (line 3), line 4 removes the order from the Order table; otherwise, an error message is displayed. In order to cancel an order, one is expected to use the cancel_item procedure to first cancel all the order items belong to this order and then use this cancel_order procedure to cancel the order itself.

The change_header procedure inputs an order header (inp_header). The procedure first checks whether such an order header exist in the database (line 1 and 2). If so, it updates the order header according to the submitted information; otherwise line 4 displays an error message.

Figure 6-4. Procedure change_header and its CFG

6.3 A theory for inferring input error correction

In this section, we present some common properties of input errors and input error correction that we have discovered through empirical study. These properties form a theoretical basis for inferring the provision of this feature from source code. The theory is an integration of invariant and empirical properties. The statistical validation for all the empirical properties will be reported in Section 6.8.1.

6.3.1 Modeling user input error

User input can be represented as a composition of data items using the three basic control constructs: **sequence**, **selection** and **iteration**. Let d_x and d_y be two data items. The sequential composition of d_x and d_y , denoted by $d_x + d_y$, represents an input which is composed of one instance of d_x and one instance of d_y . The iteration composition of d_x , denoted as $\{d_x\}^*$, represents an input which is formed by multiple instances of d_x together. The selection of either d_x or d_y , denoted by $[d_x \mid d_y]$, represents an input which can be formed by either one instance of d_x or one instance of d_y . We refer to this representation as the **input structure of the procedure**. The input structure of a procedure gives the number of data items and the data type of each data item in any input to the procedure.

For example, input of the process_order procedure given in Figure 6-1 consists of one data items of type OrderHeader (inp_header) and multiple data items of type OrderItem (inp_items); as such, it can be represented as: OrderHeader + {OrderItem}*.

Let x_k be an input of a procedure, a maximal sequential sub-composition of data items of x_k forms an **input block**. x_k itself is also an input block. We use “ $\diamond$ ” to denote an input block. For instance, there are two input blocks in the input of the `process_order` procedure: $\langle \text{OrderItem} \rangle$ and $\langle \text{OrderHeader} + \{\text{OrderItem}\}^* \rangle$.

Definition 6-1 – User input error. Let b_k be an input block in the input of a procedure. Adopting term from [219], we classify user input errors into one or a combination of the following:

1. *Error of commission (EC) in b_k : An extra instance of b_k has been entered by user.*
2. *Error of omission (EO) in b_k : An instance of b_k has been left out by user.*
3. *Value error (VE) in b_k : The values of a subset of data items in b_k have been entered incorrectly by user*

This classification is due to the fact that input blocks are the only sub-compositions that can be omitted or in extra independently. In the procedure in **Figure 6-1**, $\langle \text{OrderItem} \rangle$ is an input block. User can submit an extra instance of the type `OrderItem` or can omit an instance of type `OrderItem` that s/he was supposed to submit, which will lead to an EC or EO in the input block $\langle \text{OrderItem} \rangle$, respectively. If some attributes of the order item are submitted incorrectly, for instance Qty-Order, a VE in the input block has been created by the user.

The **path domain** of a complete path c_p is defined by the conjunction of all the branch predicates along the paths. Consequently, **input to a complete path** c_p through a procedure is input submitted to the procedure which satisfies the path domain of c_p . As such, input to a complete path forces the execution of the procedure along the path c_p . Based on this, we further define the **input structure of a path** c_p in a procedure P_T as a sub-composition of the input structure of P_T such that all the data items in the input block affect the execution of c_p . For example, the input structure of the procedure in **Figure 6-1** is $\langle \text{OrderHeader} + \{\text{OrderItem}\}^* \rangle$. However, the input structure of the path (begin, 1, 2, 3, 4, end) is only the sub-composition $\langle \text{OrderHeader} \rangle$ since data items of type `OrderItem` will not affect the execution of the path.

As any input submitted to a procedure will force the execution of the procedure along a specific complete path, each user input error can be modeled as an error in the input to a complete path; we shall refer to this as **input error in a complete path**. Similar to user input error, an input error in a complete path can also be classified into one or a combination of EC, EO or VE in an input block in the input structure of the path.

Note that an input error in a complete path is not a real input error made by user. Instead, an input error in a complete path is the reflection of a user input error made during a specific execution of the procedure. By modeling user input error as input error in a complete path through the CFG, it is possible to analyze the source code and infer all the possible types of user input errors that might occur during the execution of a system. We summarize the above modeling in the following Property 6-1. This property is invariant and is inferred directly from the fact that each user input leads to the execution of a specific complete path.

Property 6-1 (Invariant) – Modeling user input error. *Any user input error made during the execution of a procedure can be expressed as one or a combination of an input error in a complete path in the procedure.*

Proof. Let x_k be an input and b_k be an input block in x_k . Let ξ be a type of input errors in b_k . Let c_k be the completed path executed by x_k . As such, x_k is the input to the complete path c_k and ξ is also a type of input errors in the complete path c_k . This completes the proof.

In the `process_order` procedure shown in **Figure 6-1**, if a user submits a list of two valid order items and a new order header, the condition in line 2 is true and the following path will be executed: (begin, 1, 2, 3, 4, 5, 6, 4, end). Suppose that one of the two items is redundant. This is an error of commission in the number of order items made by the user. This user error can be expressed as an EC in the input block <OrderItem> in the input to the path (begin, 1, 2, 3, 4, 5, 6, 4, end).

In any error correction strategy, before being able to correct the error, one needs to detect the occurrence of the error. Based on this idea, to recover input error correction implemented by a database application, we need to first infer what the possible types of input errors that might occur in any execution of a program. As any user input error can be modeled as an input error in a complete path, in the rest of this chapter, unless otherwise stated, we will use the term “**input error**” to refer to an input error in a complete path.

6.3.2 Inferring effect error

In the execution of a program, the permanent effects are raised by its **output statements** (excluding those that display message on nonpermanent media such as screen). In **Figure 6-1(a)**, statements 3, 5 and 6 are output statements. Output statements represent the external effects raised by the program. Therefore, we can take output statements as the basic unit of the external effects. For

analyzing the errors in the external effects, we can analyze the errors in executing an output statement.

An execution of an output statement requires a set of values to be given to the required data items. We shall call each such data item an **attribute** for executing the output statement. We also classify attributes into **key** and **non-key attributes**. A set of key attributes uniquely identify a record from other records in a database table. If an output statement references an input value, an error in the input value might raise unwanted external effects such as updating the database wrongly or inserting a redundant record into a table, etc. We refer to these unwanted effects as **effect errors**.

Definition 6-2 – Effect error. *Any effect error that may happen in executing an output statement n_e is one of the following types of errors or a combination of them:*

1. *Error of commission (EC) in executing n_e : an extra execution of n_e has been committed.*
2. *Error of omission (EO) in executing n_e : an execution of n_e has been omitted.*
3. *Value Error (VE): incorrect values in some non-key attributes for executing n_e .*

Note the VE in key attributes can be represented a combination of the above basic effect errors. For example, VE in the attribute Order-No in executing statement 4 in Figure 6-3 results in an incorrect execution of a delete statement such that records that are supposed to be deleted are not deleted but some other records are incorrectly deleted. This error is the combination of the EC in statement 4 for the incorrect execution and the EO in statement 4 for the correct execution.

Input to a complete path will force the execution of the procedure along the path to raise external effects and update the databases. As such, an input error will lead to effect errors in executing output statements along the path. The following property presents the inference of effect errors resulting from an input error in a path.

Property 6-2 (Empirical) – Inference of effect error. *Let b_k be an input block of input to a path c_p . Let n_e be an output statement in c_p . An input error (EO, EC or VE) in b_k lead to the following types of effect error in executing n_e :*

1. *If b_k is the input to the path c_p , then EO or EC in b_k leads to EO or EC in executing n_e , respectively.*
2. *If n_e is in a loop whose decision node references data items in b_k , then EC or EO in b_k leads to EC or EO in executing n_e respectively.*

3. *If some non-key attributes for executing n_e reference data items in b_k then VE in b_k leads to VE in these non-key attributes*

The first sub-property is implied directly from the fact that EO or EC in the input to a path leads to the omission or extra execution of the path, which lead to EO or EC in all the output statements along the path, respectively. The second sub-property reflects the fact that if n_e is in a loop and the decision node of the loop references some data items in the input block b_k , then the number of iterations of the loop (thus, number of execution of n_e) is influenced by values of the data items in b_k . Therefore, any EO or EC in b_k leads to extra iterations of the loop or some iterations are left out. The third sub-property is quite obvious.

For example, consider EC in the input block <OrderItem> of the input to the following path through the procedure in **Figure 6-1**: (begin, 1, 2, 3, 4, 5, 6, 4, end). The output statements 5 and 6 are in a loop whose decision node is 4. Statement 4 references to the number of instances of OrderItem in the input block. Therefore, according to sub-rule 2 of Property 6-2, EC in this input block will lead to EC in the two output statements. Indeed, EC in the input block means an extra order item has been submitted by the user. Thus, there is an extra element in the array `inp_items`, which leads to an extra iteration of the for-loop (4, 5, 6). This leads to extra execution of output statements 5 and 6; hence EC in 5 and EC in 6.

6.3.3 Inferring input error correction

6.3.3.1 Error correction node

The objective of providing error correction is to correct all the effect errors which might result from user input errors. In a database application, an effect error in executing an output statement n_e can be corrected by executing another output statement n_f with a set of attribute values that can be derived from the attribute values given to the erroneous execution of the output statement n_e . We define n_f as the **error correction node** for correcting the effect error in executing n_e .

Without loss of generality, we classify output statements into three types: **insert**, **modify** and **delete**. Due to control reasons, records in some record types might not be allowed to modify or delete once they have been inserted. As such, we classify record types in a database application into: **updatable**, **modifiable** and **non-updateable**.

Property 6-3 (Invariant) – Error correction node. *The required output statement to correct each type of effect errors in each of the following types of output statements that operate on a record type R is as follows:*

1. *Insert node n_e :*
 - a. *EO: n_f is also an insert node operating on R and sets the same attributes as n_e*
 - b. *EC: If R is updateable, then n_f is a delete node operating on R . Otherwise, it is an insert node operating on record type S (S can be another record type or R itself) to indicate that the wrongly inserted record has been deleted.*
 - c. *VE: If R is updateable or modifiable, then n_f is a modify node operating on R to modify the attributes set by n_e . Otherwise, it is an insert node operating on record type S (S can be another record type or R itself).*
2. *Modify node n_e : For each type of error (EC, EO or IE), n_f is a modify node operating on R to modify the attributes set by n_e .*
3. *Delete node n_e :*
 - a. *EO: n_f is a delete node operating on R .*
 - b. *EC: n_f is an insert node operating on R .*
 - c. *VE: this cannot be corrected directly.*

Proof. Clearly, the error correction node for correcting EO in any output statement is the output statement itself. This implies the result for all the EO cases. For correcting EC in an insert statement, if R is updateable, clearly, it is a delete statement operating on R to delete the wrongly inserted record. Otherwise, a new record in the same record type or another record type can be inserted to indicate that the wrongly inserted record has been deleted. As such, in this case, the required error correction statement is an insert statement operating on a record type S . For correcting VE in an insert statement, if R is updatable or modifiable, clearly, it is a modify statement operating on R to modify the record inserted to the right value. Otherwise, a new record in the same record type R or another record type S can be inserted to indicate that the wrongly inserted record has been modified. As such, in this case, the required error correction statement is an insert statement operating on a record type S . For correction EC or VE in a modify statement, the wrongly modifies recode should be modified to the previous value or the correct value respectively. As such, the required error correction statement is a modify statement operating on the same record type. For correcting EC in a delete statement, clearly, the error correction statement is an insert statement to insert back the wrongly deleted record. For correcting VE in a delete statement, the wrongly deleted record should be inserted and the right record should be deleted. Therefore, this type of

error cannot be corrected directly by one correction node. This completes the proof.

Based on Property 6-3, we can derive the required output statement to correct each type of errors in each output statement in a program. In the process_order procedure given in **Figure 6-1**, statement 3 is an output statement. An EC in executing statement 3 inserts a record into the Order table. This error can be corrected by executing an SQL statement which deletes the record from the Order table. Therefore, the required output statement for correcting this EC in executing statement 4 is the output statement 4 in Figure 6-3(a).

6.3.3.2 Basic collection of error correction paths

Let c_p be a complete path in a procedure P_T . Let $C = \{c_{i_1}, \dots, c_{i_k}\}$ be a collection of complete paths through some procedures (P_T might be one of them), where $k \geq 1$. If an input error in c_p can be corrected by executing the paths in the collection C and the collection is minimal, then the collection is called a **collection of error correction paths** for correcting the error. Furthermore, if for each output statement n_e whose execution is in error due to the input error, there is a unique j , $1 \leq j \leq k$, such that the error in executing n_e is corrected by the execution of the path c_{i_j} in C , then the collection is called a **basic collection of error correction paths** for correcting the input error. Each path in the basic collection is referred to as an **error correction path** for correcting the input error.

For each input error, at least one basic collection of error correction paths for correcting the input error should be provided by the database application. We discover some probable patterns that exist between a complete path and a basic collection of error correction paths for correcting a type of input errors in the path. Next we present the patterns as an empirical property for inferring basic collections of error correction paths for correcting an input error.

Property 6-4 (Empirical) – Basic collection of error correction paths. *Let c_p be a path in a procedure P_T and ξ be a type of input error in c_p . Let $\{c_{i_1}, \dots, c_{i_k}\}$ be a collection of complete paths through some procedures (P_T might be one of them). The collection is probably a basic collection of error correction paths for correcting ξ if and only if there exists a partition $\{O_p^1, \dots, O_p^k\}$ of output statements in c_p such that for each j , $1 \leq j \leq k$, there is a one-to-one mapping f_j from O_p^j to the set of output statements in c_{i_j} with the following properties:*

1. For each output statement $n_e \in O_p^j$, $f_j(n_e)$ is an error correction node for correcting the effect error resulting from ξ in executing n_e .
2. For any n_e and $n_f \in O_p^j$, n_e and n_f are iteration-isomorphic in n_p if and only if $f_j(n_e)$ and $f_j(n_f)$ are iteration-isomorphic in c_{i_j} .

Two statements are said to be **iteration isomorphic** if and only if they are always in the same loop. A **partition** of a set O is a set of non-empty subsets of O such that every element in O is in exactly one of these subsets.

The rationale of Property 6-4 is as follows: By satisfying the first condition, it is probable that an execution of statement $f_j(n_e)$ can correct the resulting error in an execution of statement n_e . By satisfying the second condition, it is probable that the number of times that $f_j(n_e)$ is executed in an execution of c_{i_j} can be made identical to the number of times that n_e is executed in an execution of c_p . As a result, we can execute the associated program through c_{i_j} with appropriate user input such that the execution of statement $f_j(n_e)$ corrects the resulting error in the executions of n_e in an execution of P_T through c_p .

Take the path $c_p = (\text{begin}, 1, 2, 3, 4, 5, 6, 4, \text{end})$ through the CFG shown in **Figure 6-1** (b) as an illustration for Property 6-4. Let ξ be EC in the input block $b_p = \langle \text{OrderHeader} + \{\text{OrderItem}\}^* \rangle$ in the input to c_p . We apply Property 6-4 to infer that the following paths form a basic collection of error correction paths for correcting ξ :

- $c_x = (\text{begin}, 1, 2, 3, 4, \text{end})$ through the CFG in Figure 6-3(b)
- $c_y = (\text{begin}, 1, 2, 3, 4, \text{end})$ through the CFG in

Figure 6-2(b)

By applying Property 6-2, we can infer that EC in the input block b_p results in the following effect errors in executing c_p :

- EC in the execution of statement 3
- EC in the execution of statement 5
- EC in the execution of statement 6

We can form a partition of output statements in c_p as follow: $O_p = \{O_p^1, O_p^2\}$, where $O_p^1 = \{3\}$ and $O_p^2 = \{5, 6\}$. Based on Property 6-3 on properties of error correction nodes, the mapping functions f_1, f_2 can be defined as follows: $f_1(3) = 4$ in c_x and $f_2(5) = 3$ in c_y and $f_2(6) = 4$ in c_y . Furthermore, output statements 5 and 6 are iteration-isomorphic in c_p and 3 and 4 are iteration isomorphic in c_y . As the set $\{c_x, c_y\}$ satisfies all the conditions of Property 6-4, it is a basic collection of error correction paths for correcting ξ . Indeed, by executing c_x , EC

in statement 3 in c_p which inserts an extra record into the Order table can be corrected by statement 4 in c_x which deletes a record from the Order table. By executing c_y , EC in statement 5 in c_p which updates the Product table can be corrected by statement 3 in c_y which updates the Product table and sets the Qty-Sold attributes to a correct value. Similarly, EC in statement 6 in c_p which inserts a new record in the Order-Item table can be corrected by statement 4 in c_y which deletes a record from the Order-Item table.

Although correcting an input error requires the entering of right values that can only be determined from the dynamic behaviors of the program, from empirical studies, we find that all the likelihood conditions stated in Property 6-4 are sufficient to conclude that a collection of paths is a basic collection of error correction paths for correcting an input error in a complete path. Property 6-4 is powerful as it allows one to conclude whether a set of paths is a basic collection of error correction paths for correcting an input error without considering computation details; which means it does not depend on the input values. As such, it is clear that a type of input error can be corrected by using the same basic collection of error correction paths regardless of the input values.

The second condition of this property implies that if two output statements are iteration-isomorphic, as long as their error correction nodes are also iteration-isomorphic, any error in the output statements can be corrected. As such, the exact number of iterations of a loop (if any) is not important. This condition frees our analysis from loop-handling. Based on this property, we define representative paths.

A **representative path (r-path)**, r_p , in a CFG is a complete path in the CFG such that any loop included is iterated with exactly one time. The second condition of Property 6-4 implies that, we need not to be concerned with the exact number of times that a loop is iterated. Therefore, we use an r-path to represent all the complete paths in the CFG of a program that are identical with the r-path except iterating some loops included in the representative path a different positive numbers of times.

For example, in **Figure 6-1 (b)**, the r-path (begin, 1, 2, 3, 4, 5, 6, 4, end) represents all the paths in the following set:

{(begin, 1, 2, 3, 4, 5, 6, 4, end),
(begin, 1, 2, 3, 4, 5, 6, 4, 5, 6, 4, end)....
(begin, 1, 2, 3, 4, 5, 6, 4... 4, end)}.

On the other hand, the r-path (begin, 1, 2, 3, 5, end) represents only itself as there is no loop in the r-path.

In opposite to the total number of paths in a CFG, the total number of r-paths in the CFG is always finite. This makes our approach scale well for real applications. In the rest of this chapter, we will only consider r-paths. Moreover, we will only consider r-paths which contains output statements because only these paths raise external effects.

Table 6-1 lists all the r-paths in each procedure in the E-Order system. Table 6-2 gives the input structure of each r-path and the input blocks in the r-path. Table 6-3 lists all the possible types of errors in each r-path and their respective basic collections of error correction paths for correcting the type of input errors which can be computed by applying Property 6-4.

Table 6-1. R-paths in the E-Order application

Procedure	R-Path	Input structure
process_order	$r_1 = (\text{begin}, 1, 2, 3, 4, \text{end})$	<OrderHeader>
	$r_2 = (\text{begin}, 1, 2, 3, 4, 5, 6, 4, \text{end})$	<OrderHeader + {OrderItem}*>
	$r_3 = (\text{begin } 1, 2, 4, 5, 6, 4, \text{end})$	<OrderHeader + {OrderItem}*>
cancel_item	$r_4 = (\text{begin}, 1, 2, 3, 4, \text{end})$	<Integer + Integer>
cancel_order	$r_5 = (\text{begin}, 1, 2, 3, 4, \text{end})$	<Integer>
change_header	$r_6 = (\text{begin}, 1, 2, 3, \text{end})$	<OrderHeader>

Table 6-2. Input structures and input blocks

R-Path	Input structure	Input block
r_1	<OrderHeader>	<OrderHeader>
r_2	<OrderHeader + {OrderItem}*>	<OrderHeader + {OrderItem}*>, <OrderItem>
r_3	<{OrderItem}*>	<OrderItem>
r_4	<Integer + Integer>	<Integer + Integer>
r_5	<Integer>	<Integer>
r_6	<OrderHeader>	<OrderHeader>

For many procedures, the correctability of all of its input errors can be deduced from the correctability of some of these errors. This property is presented next.

Property 6-5 (Invariant) – Program correctibility. *Let P_T be a procedure in a database application. Any input error in P_T is correctable if and only if for*

each r -path through P_T , there exists a basic collection of error correction paths for correcting all input errors of type EC in the r -path.

The proof for this property is straightforward. Assume that any EC in each r -path through P_T is correctable. Any EO in the r -path can be corrected by executing the program through the r -path itself. Any VE can also be corrected by treating it as an EC followed by an EO . Therefore, we execute the correction path for correcting EC in the r -path followed by executing the procedure through the r -path with the correct input values. This completes the proof.

Table 6-3. Basic collections of error correction paths

R-path	Input error type	Effect error	BCECP
r_1	EC in <OrderHeader>	EC in node 3	{ r_5 }
	EO in <OrderHeader>	EO in node 3	{ r_1 }
	VE in OrderHeader	VE in node 3	{ r_6 }
r_2	EC in <OrderHeader + {OrderItem}*>	EC in nodes 3, 5, 6	{ r_4, r_5 }
	EO in <OrderHeader + {OrderItem}*>	EO in nodes 3, 5, 6	{ r_2 }
	EC in <OrderItem>	EC in nodes 5, 6	{ r_4 }
	EO in <OrderItem>	EO in nodes 5, 6	{ r_3 }
	VE in OrderHeader	VE in node 3	{ r_6 }
	VE in OrderItem	VE in nodes 5, 6	{ r_4, r_2 }
r_3	EC in <OrderHeader + {OrderItem}*>/<OrderItem>	EC in node 5, 6	{ r_4 }
	EO in <OrderHeader + {OrderItem}*>/<OrderItem>	EO in node 5, 6	{ r_3 }
	VE in OrderItem	VE in nodes 5, 6	{ r_4, r_2 }
r_4	EC in <Integer + Integer>	EC in nodes 3, 4	{ r_3 }
	EO in <Integer + Integer>	EO in nodes 3, 4	{ r_4 }
r_5	EC in <Integer>	EC in nodes 4	{ r_1 }
	EO in <Integer>	EO in nodes 4	{ r_5 }
r_6	EC in <OrderHeader>	EC in node 3	{ r_6 }
	EO in <OrderHeader>	EO in node 3	{ r_6 }
	VE in OrderHeader	VE in node 3	{ r_6 }

Legend: BCECP – Basic collection of error correction paths

From Table 6-1, Table 6-2 and Table 6-3, it can be verified that any EC in each r -path in the process_order (**Figure 6-1**) procedure is correctable. Therefore, according to Property 6-5, any input error in executing the procedure is also correctable. This can also be verified from Table 6-3 as for any other type of input errors in each r -path in the procedure, there exists one basic collection of error correction paths for correcting the type of input errors.

Lastly, we present an empirical property, which is the integration of Property 6-4 and Property 6-5 to provide a practical mechanism for the recovery of error

correction provided by a database application. Before presenting the property, we shall define error correction program. If each r-path in a procedure P_S is in a basic collection of error correction paths for correcting a type of input error in an r-path through a procedure P_T , then P_S is called an **error correction procedure** for P_T .

Property 6-6 (Empirical) – System correctibility. *It is probable that any input error of a procedure P_S is correctable if and only if one of the following condition holds:*

1. *Based on basic collection of error correction paths, Property 6-5 infers that any input error of P_S is correctable.*
2. *Procedure P_S is an error correction procedure for another procedure P_T , whose all input errors are correctable.*

The rationale of the second condition of Property 6-6 is as follows. As P_S is an error correction procedure for P_T , any input error of P_S is resulted from an erroneous execution of P_S which was actually aimed to correct an erroneous execution of P_T . It is probable that we can treat the resultant external effect from the erroneous execution of P_S and P_T as an EC in an r-path through P_T and the correct execution of P_T as an EO in another r-path through P_T . As any input error of P_T is correctable, both of the EC and the EO can be corrected. Consequently, any input error of P_S is correctable. However, this property is empirical and needs statistical proof.

As an illustration for this property, it can be verified from Table 6-1, Table 6-2 and Table 6-3 that procedures in

Figure 6-2, Figure 6-3 and Figure 6-4 are error correction procedures for the procedure shown in **Figure 6-1** as each r-path in these procedures is in a basic collection of error correction paths for correcting a type of input errors in the procedure in **Figure 6-1**. As all input errors in the procedure in **Figure 6-1** are correctable, all the input errors in the procedures shown in

Figure 6-2, Figure 6-3 and Figure 6-4 are also correctable. This can be verified easily from Table 6-3.

6.4 Automated recovery

Property 6-1 can be used to derive all the possible types of input errors. Also from the same property, it is highly probable that any input error is one of these basic types of errors or a combination of them. As a result, if all these basic types can be corrected, then any other error can also be corrected. Therefore, the approach proposed in this section can be applied to recover input error correction provided for the basic error types. Recovery for other types of input

errors can be based on the recovered information of the basic types of errors. The automated recovery is carried out in four steps: (1) program analysis, (2) derive input errors and effect errors, (3) recover basic collections of error correction paths and (4) recover the input correctability of a system.

1. **Step 1 – Program analysis.** For each procedure in the database application, constructs its CFG. For each CFG, compute all the r-paths through the CFG. Each time, we traverse through the CFG via an r-path until all such paths have been traversed. That is, we traverse through the CFG in such a way that each selection construct (if statement) is traversed via one of its branches. Each iteration constructs (loop statement) is traversed once or skipped totally.
2. **Step 2 – Derive input error and effect error.** For each r-path r_p , derive the set $\xi(r_p)$ of all the possible types of input errors in the path by applying Property 6-1. We identify all the input blocks in the input to the path. For each input block, include an EO and an EC into the set $\xi(r_p)$. For each data item in the input block which influences non-key attributes, include a VE into the set $\xi(r_p)$. For each type of input error in each r-path, Property 6-2 is applied to derive the effect errors in executing each output statement in the r-path.
3. **Step 3 – Recover basic collection of error correction paths.** For each r-path r_p , for each type ξ of input errors identified in Step 2, we recover basic collections of error correction paths for correcting the type of input errors by applying Property 6-3 and Property 6-4.
First, we compute a set $R = \{r_{q_1}, \dots, r_{q_n}\}$ of r-paths, whose number of output statements is equal or less than the number of output statements in r_p . The basic idea of this step is that for each r-path r_{q_i} in R , we construct all the possible one-to-one mappings from a subset of the output statements in r_p onto the set of output statements in r_{q_i} such that the following condition holds: An output statement n_f in r_{q_i} is only mapped to an output statement n_e in r_p if n_f is the error correction node for correcting the effect error resulting from ξ in executing n_e . Property 6-3 is used to identify whether n_f is the error correction node of n_e . These one-to-one mappings are referred to as **error correction mappings**. Assuming that the following error correction mappings can be formed:

$$\begin{aligned} f_1: O_p^1 &\rightarrow O_{q_1} \\ &\dots \\ f_n: O_p^n &\rightarrow O_{q_n} \end{aligned}$$

where $O_p^i, i = 1..n$, is a subset of the output statements in r_p and $\{O_{q_i}, i = 1..n\}$, is the set of output statements in the r-path $r_{q_i} \in R$. We then compute all subsets L_k of the set $\{O_p^1, \dots, O_p^n\}$ such that L_k forms a partition of the output statements in r_p . If there is no such a subset L_k then according to Property 6-4, there is no basic collection of error correction paths for correcting ξ ;

If there is such a subset $L_k = \{O_p^i\}$ where $i \in \{s, \dots, t\}, 1 \leq s \leq t \leq n$, then clearly $\mathcal{L}$ is a partition of the output statements in r_p and there are one-to-one mappings $f_i: O_p^i \rightarrow O_{q_i}, i = 1..n$, which satisfy the first condition of Property 6-4. We then check whether the mappings satisfy the second condition of the property. If so, the set of r-paths $\{r_{q_i}\}$, where $i \in \{s, \dots, t\}, 1 \leq s \leq t \leq n$, forms a collection of error correction paths for correcting ξ .

As an illustration for this step, we try to recover the basic collection of error correction path for correcting EC in the input block $\langle \text{OrderHeader} + \{\text{OrderItem}\}^* \rangle$ in the input to path r_2 through the process_order in **Figure 6-1**. All the notations used in this example are from Table 6-1, Table 6-2 and Table 6-3. This input error leads to EC in nodes 3, 5 and 6. Let R be the set of r-paths through the E-Order system whose total number of output statements is less than or equal to that of r_2 : $R = \{r_1, r_2, r_3, r_4, r_5, r_6\}$. According to Table 6-3, EC in the input block leads to EC in nodes 3, 5 and 6 in the CFG of procedure process_order in **Figure 6-1**. Applying Property 6-3, we can infer the following error correction nodes:

- output statement 3 in r-path r_4 is an error correction nodes for correcting EC in output statement 5 in r_2 .
- output statement 4 in r-path r_4 is the error correction node for correcting EC in output statement 6 in r_2 .
- output statement 4 in r-path r_5 is the error correction node for correcting EC in output statement 3 in r_2 .

As such, the following error correction mappings can be formed

$$\begin{aligned} f_1: \{5, 6\} \text{ in } r_2 &\rightarrow \{3, 4\} \text{ in } r_4 \\ f_2: \{3\} \text{ in } r_2 &\rightarrow \{4\} \text{ in } r_5 \end{aligned}$$

We then compute all the subsets of the following set $\{\{5, 6\}, \{3\}\}$. Obviously, the following subset $\{\{5, 6\}, \{3\}\}$ forms a partition of output statements of r_2 . Moreover, nodes 5 and 6 are iteration-isomorphic in r_2 and nodes 3 and 4 are iteration-isomorphic in r_4 . As such, mappings f_1 and f_2 satisfy the second condition of Property 6-4. We conclude that $\{r_4, r_5\}$ form a basic collection of error correction paths for correcting EC in the input block `<OrderHeader + {OrderItem}*>` in the r-path r_2 .

4. **Step 4 – Recover system correctability.** Let P_T be a procedure, for each r-path through P_T , the results computed from Step 3 is checked to verify whether EC in the r-path is correctable. If it is not correctable, then checking for P_T is completed with no conclusion from Property 6-4. If it is correctable for each r-path through P_T , then P_T satisfies Property 6-5 and therefore it is concluded that all after effect input errors of P_T are correctable. Let Ω be the set of procedures which all input errors have been concluded as correctable in this way. Let Ψ be the remaining procedures.

Next, we iterate over Ψ until the values of Ω is stabilized (that is its value does not change from the previous iteration). For each iteration, for each procedure P_S in Ψ , check whether P_S is an error correction procedure for another procedure P_T in Ω . If this is affirmative, according to the second condition of Property 6-6, all the input errors of P_S are also correctable. Therefore, P_S is removed from Ψ and added to Ω .

Finally, set Ω contains all the procedures whose input errors are correctable and set Ψ contains all the procedures whose input errors are not correctable. This step completes the recovery of input error correction for the whole system.

6.5 Integrating functional and structural testing

In order to test software more accurately, both code and specification behaviors must be covered. Most existing techniques are either solely code-based or specification-based through adopting structural or functional testing strategies, respectively. The implementation of input error correction recovered by the approach proposed in Section 6.4 enables the reconciliation of input error correction implementation and specification. As a result, it is possible to combine both structural and functional testing in one test suite. The reconciliation can be done based on existing approaches [32, 34, 40, 176]. In

this section, we present an integrated functional and structural approach called **Black&White** to testing input error correction. The approach involves the following four steps:

1. Form the code-based equivalence partition with respect to input error correction.
2. Form the specification-based equivalence partition with respect to input error correction.
3. Reconcile input error correction implementation and specification.
4. Generate test cases based on the reconciled code-based equivalence partition.

(1) Form the code-based equivalence partition

Any input to a complete path will force the execution of the program along the path and raise some external effects. Consequently, input to a path could possibly lead to an input error and result in effect errors. The use of r-paths implies that the possible types of input errors in all the paths represented by an r-path are identical to the possible types of input errors in the r-path. That means any input value in the path domain of the r-path which also satisfy the input structure of the r-path would possibly lead to one of these input errors.

Table 6-4. CIE partition of E-Order

R-path	CIE class			IEC-implementation
	Notation	Input structure	Path domain	
r ₁	C ₁	<OrderHeader>	(order == NULL) and (inp_items.length == 0)	{{r ₅ }, {r ₁ }, {r ₆ }}
r ₂	C ₂	<OrderHeader + {OrderItem}*>	(order == NULL) and (inp_items.length > 0)	{{r ₄ , r ₅ }, {r ₂ }, {r ₄ }, {r ₃ }, {r ₆ }, {r ₄ , r ₂ }}
r ₃	C ₃	<OrderHeader + {OrderItem}*>	(order != NULL) and (inp_items.length > 0)	{{ r ₄ }, {r ₃ }, {r ₄ , r ₂ }}
r ₄	C ₄	<Integer + Integer>	(orderItem != NULL) and (orderItem.Qty-Delivered == 0)	{{r ₃ }, {r ₄ }}
r ₅	C ₅	<Integer>	(order != NULL) and (order_items == NULL)	{{r ₁ }, {r ₅ }}
r ₆	C ₆	<OrderHeader>	(order != NULL)	{{r ₆ }}

Property 6-4 implies that it is highly probable that a type of input errors in an r-path can be corrected by applying the same basic collections of error correction paths, regardless of the input values. That means any input value in the path domain of the r-path which satisfies the input structure of the r-path and leads to the same type of errors can probably be corrected by the same basic collections

of error correct paths. Therefore, with regard to input error correction behavior, each such input value is handled in the same manner (by executing the same basic collections of error correction paths for correcting the same type of input errors). Consequently, input values in the path domain of an r -path which satisfy the input structure of the r -path form a kind of equivalence class with respect to the input error correction feature. We refer to this kind of equivalence classes as **IEC-equivalence classes**. If x_k and x_l belong to the same IEC-equivalence class, we say that the two input values are equivalent with respect to the input error correction feature, or they are IEC-equivalent. We shall refer to this as **code-based IEC-equivalence (CIE) class**. Each CIE class is attached with a set of basic collections of error correction paths for correcting all the possible types of input errors resulting from input values in the equivalence class. We refer to this set of basic collections as **IEC-implementation** of the equivalence class. The set of all the CIE classes forms the **CIE partition** of the system. CIE partition is derived directly from the recovered input error correction. For example, the CIE partition of the E-Order system can be derived from Table 6-1, Table 6-2 and Table 6-3. The partition is shown in Table 6-4.

(2) Form the specification-based equivalence partition

Before reconciliation, the specification is partitioned into sub-domains, either formally or informally, such that input values in each sub-domain are treated uniformly by the specification. A specification-based equivalence class with respect to the input error correction feature is constructed either manually or systematically depending on the formality of the specification. If requirements are specified using formal specification languages such as those based on predicate calculus or state transformations, the program input domain can be easily partitioned into a finite number of input classes [176]. However, through numerous discussions with software engineers, we found that formal specification does not exist for functional features like input error correction. In this case, partition-based functional testing techniques can be applied.

Equivalence class testing [108] is a functional testing method which can be used to partition the program input space into equivalence classes from the requirements. Therefore, we can assume that it is possible to manually partition the input error correction requirements into equivalence classes; we refer to these equivalence classes as **spec-based IEC-equivalence (SIE) classes**. Similar to the CIE classes, each SIE class is defined by an input structure and a sub-spec domain specifying the constraints for all the input values in the class. Each SIE class is also attached with an **IEC-specification** describing how input error correction should be handled for all the input values in this equivalence

class. The collection of all the SIE classes forms the **SIE partition** of the system.

Assuming that for the E-Order system, we can derive the SIE partition as shown in Table 6-5. The IEC-specification describes the input error correction feature for each SIE class. For example, IEC-specification for S_1 states that any omission in the submission of a new order (including an order header and a list of order items) can be corrected by executing the use case itself. Any commission in the submission of a new order can be corrected by executing the “cancel order item” use case followed by the “cancel order” use case. Erroneous values of the order header submitted can be corrected by the “change order header” use case. Erroneous values of an order item submitted (e.g. quantity) can be corrected by the “change order item” use case.

Table 6-5. SIE partition of E-Order

Use cases	SIE classes			IEC-specification
	Notation	Input structure	Sub-spec domain	
new order	S_1	-An order header -A list of order items	(new order header) and (list of order items is not empty)	-EO: self-correctable -EC: “cancel order item” followed by “cancel order” -VE in order header: “change order header” -VE in an order item: “change order item”
add order items	S_2	-An order header -A list of order items	(valid order header) and (list of order items is not empty)	-EO: self-correctable -EC: “cancel order item” -VE in an order item: “change order item”
cancel order	S_3	-An order number	(valid order number) and (the list of order items is empty)	-EO: self-correctable -EC: “new order”
cancel order item	S_4	-An order number -A product number	(valid order number) and (Qty-delivered = 0)	-EO: self-correctable -EC: “add order item”
change order header	S_5	-An order header	valid order header	-EC, EO, VE: self-correctable
change order items	S_6	-An order header -A list of order	(valid order header) and (valid product numbers of order items)	-EO, EC, VE: self-correctable

		items		
--	--	-------	--	--

(3) Reconciliation

The CIE partition and the SIE partition represent two ways the input error correction for a database application can be decomposed and solved. Testing based on either the SIE partition (functional) or CIE partition (structural) alone is not adequate to ensure the reliability of the system. Therefore, testing that can be done on the intersection of these partitions to combine both implementation and specification characteristics. The intersection of a CIE partition and a SIE partition is referred to as the **program IEC-equivalence (PIE) partition**. The PIE partition is constructed by taking pair wise intersection of the set of CIE classes and the SIE classes. A CIE class can only be intersected with a SIE class if their input structures are compatible and the intersection of the CIE class's path domain and the SIE class's sub-spec domain is not empty. For example, a CIE class whose input structure is an order header cannot be intersected with a SIE class whose input structure is an order item.

The non-empty intersection of a SIE class and a CIE class forms a **PIE class**. If the SIE class and the CIE class are not equal; there will be some input values which are not in the intersection of these two classes. We refer to those values in the SIE class but not in the CIE class as **specification non-compatible input class (SNIC)**. Similarly, those values which are in the CIE class but not in the SIE class are referred to as **code non-compatible input class (CNIC)**. The following reconciliation procedure should be carried out manually:

1. For each PIE class, evaluate the "equality" between the IEC-implementation and the IEC-specification. The IEC-implementation is considered equal to the IEC-specification if the set of correction paths in the IEC-implementation represent the description in the IEC-specification for correcting the input errors in the PIE class.
2. For each SNIC or SIE class which does not have any compatible CIE class, the specification is first checked whether it is correct. It is unreasonable to assume that the specification is always correct. If there are some errors in the specification it is rectified. Otherwise, check whether there are path missing errors which occur when the specification requires some actions for the class but the program does not contain the corresponding paths. If this is the case, refine the implementation accordingly.
3. For each CNIC or CIE class which does not have any compatible SIE class, first check the specification if it is too abstract and can be split into smaller use case scenarios. Otherwise, check whether there are redundant path

errors which occur when the program contains some paths which do not correspond to any requirement.

As an illustration, we reconcile the SIE partition and CIE partition of the E-Order application. The SIE partition is given in Table 6-5 and the CIE partition is given in Table 6-4. Based on the input structures, the path domains of CIE classes and sub-spec domain of SIE classes, we can take pair wise intersection of CIE classes and SIE classes. The non-empty intersection of these classes is shown in Figure 6-5. For each PIE class, the equality of the IEC-implementation and the IEC-specification can be easily verified from the r-paths and the descriptions. Take the PIE class created from the intersection of S_1 and C_2 as an example. In S_1 , EC can be corrected by executing the “cancel order item” use case followed by the “cancel order” use case. In C_2 , EC in $\langle \text{OrderHeader} + \{\text{OrderItem}\}^* \rangle$ can be corrected by executing r_4 and r_5 . r_4 represents a scenario where users can cancel an order item and r_5 represents a scenario there users can cancel an order if all the ordered items have been cancelled.

Table 6-6. Non-empty intersection of SIE classes and CIE classes

Intersection	Input structure	PIE class	CNIC	SNIC
$S_1 \cap C_2$	$\langle \text{OrderHeader} + \{\text{OrderItem}\}^* \rangle$	$(\text{order} == \text{NULL})$ and $(\text{inp_items.length} > 0)$	-	-
$S_2 \cap C_3$	$\langle \text{OrderHeader} + \{\text{OrderItem}\}^* \rangle$	$(\text{order} != \text{NULL})$ and $(\text{inp_items.length} > 0)$	-	-
$S_3 \cap C_5$	$\langle \text{Integer} \rangle$	$(\text{order} != \text{NULL})$ and $(\text{order_items} == \text{NULL})$	-	-
$S_4 \cap C_4$	$\langle \text{Integer} + \text{Integer} \rangle$	$(\text{orderItem} != \text{NULL})$ and $(\text{orderItem.Qty-Delivered} == 0)$	-	-
$S_5 \cap C_1$	$\langle \text{OrderHeader} \rangle$	$(\text{order} != \text{NULL})$	-	-

As such, in term of EC in the whole input, the IEC-implementation and IEC-specification are equivalent. However, IEC-implementation of C_2 also contains correction paths for correcting EC in the submission of order item, which is not specified in the IEC-specification of S_1 . At this point, IEC-specification and the IEC-implementation need to be reconciled. Similarly, the IEC-specification also misses the description of error correction for correcting EO in the order items. Assume that the E-Order application must provide error correction for EC and EO in the submission of order items. Consequently, the IEC-specification of S_1 should be refined as shown in Figure 6-5.

From Table 6-6, we can see that CIE class C_1 has no compatible SIE class and the SIE class S_6 has no compatible CIE class. The CIE class C_1 is derived from the r-path r_1 which represents a scenario where the user just submits an order header without submitting any order item and the order header is inserted into the Order table. Even though the input structure of C_1 is compatible to that of S_5 , these two equivalence classes cannot be intersected because the intersection of C_1 's path domain and S_5 's sub-spec domain is empty. This might be a redundant path error or a specification error depending on the customer requirements. This point needs to be clarified with the customer and then either the implementation or the specification needs to be rectified accordingly. The SIE class S_6 represents a scenario which allows the user to modify an ordered item, for example the Qty-Order attribute. The E-Order application provides no implementation for this SIE class; clearly, this is a missing path error.

1. EO in both order header and order items: self-correctable 2. EC in both order header and order items: "cancel order item" followed by "cancel order" 3. EO in order items: "add order item" 4. EC in order items: "cancel order item" 5. VE in order header: "change order header" 6. VE in an order item: "change order item"

Figure 6-5. Refined IEC-specification of S_1

(4) Test data generation

Once the CIE partition and the SIE partition are reconciled, the CIE classes and SIE classes should be consistent. Hence, test cases can be generated from the reconciled CIE classes to fully test the input error correction. After the reconciliation, each input value in each CIE class is treated uniformly by both specification and implementation. Therefore, test cases generated from the reconciled CIE partition exercise both code and specification characteristics.

For each CIE class, based on the r-path representing the class, for each type of input errors in the r-path, a test case is designed to create an input error of the type and a test case is designed to exercise each error correction path in the basic collection of error correction paths for correcting the type of input errors. Let r_p be an r-path and B_ξ be a basic set of error correction paths for correcting the type ξ of input errors. The test data generation of the error correction for ξ is carried out in the following steps:

1. Design a test case to force the execution of r_p and create an input error of the type ξ .
2. Execute the test case and check the results.

3. For each path r_q in B_ξ
 - a. Based on the input value of the test case design in (1), design a test case to execute r_q .
 - b. Execute the test case design in 3(a)
4. Check the database to see whether the error rose during the execution of the test case in (2) is indeed corrected.

It can be easily verified that attribute values for executing an error correction path in Step 3(a) can be generated based on the erroneous values designed in step 1. In deed, for EO cases, the required attribute values for executing the error correction node are the attribute values supposed to be given to the execution of the original node. For EC cases, the required attribute values for executing the error correction node can be derived from the attribute values given to the erroneous execution of the original node. For VE cases, the require attribute values for executing the error correction nodes can be derived from the attribute values supposed to be given to the execution of the original node and the attribute values given to the erroneous execution of the original node.

6.6 Aiding maintenance of input error correction

In Section 6.4, we have described an approach for the automated recovery of input error correction in a database application from source code. As input error correction is an important functional feature, clearly, the information recovered is useful in maintenance and comprehension of database applications. When a programmer writes a new or edits an existing input error correction feature, s/he hopes to make this change without breaking the system. How can one check that the new feature does not interfere with existing features? In this section, we present the effect-oriented decomposition slicing technique which makes use of the recovered information to assist software engineers in maintaining this feature.

6.6.1 Effect-oriented decomposition slicing

We are interested in program statements which implement the input error correction feature for the database application. Therefore we adopt the idea of decomposition slicing to decompose a program with respect to input error correction feature. We introduce the concept of effect-oriented decomposition slicing. The idea is that an effect-oriented decomposition slice of a program, taken with respect to the external effects raised during the execution of the program, should contain only statements which affect the attributes of some external effects raised during the execution of the program. Below, we provide the formal definition of an effect-oriented decomposition slice.

Definition 6-3 – Effect-oriented decomposition slice. Let O_P be a set of some output statements in a program P . The statements in $\mathcal{ES}(O_P) = \bigcup_{N_k \in O_P} S(\text{refs}(n_k), n_k)$, where $\text{refs}(n_k)$ is a set of variables referenced by statement n_k and $S(\text{refs}(n_k), n_k)$ is the slice taken at statement n_k with respect to $\text{refs}(n_k)$, forms the effect-oriented decomposition slice of the program taken with respect to O_P .

For example, in the procedure `process_order` in **Figure 6-1**, if $O_P = \{6\}$ then $\mathcal{ES}(O_P) = \{4, 6\}$ is the effect-oriented decomposition slice of the program taken with respect to O_P . Clearly, $\mathcal{ES}(O_P)$ contains only statements which control the execution of the output statement 6.

Definition 6-4 – Error-correction decomposition slice. Let ξ be a type of input errors in an r -path r_p . Let r_q be a path in a basic collection of error correction set for correcting ξ . Let O_q be the set of output statements in r_q . We define the effect-oriented decomposition slice taken with respect to all the output statements in r_q , $\mathcal{ES}(O_q)$, as the error-correction decomposition slice taken with respect to r_q for correcting the type ξ of input errors.

As an illustration, consider the r -path r_2 through the procedure in **Figure 6-1**. Let ξ be EC in the input block `<OrderItem>`, which leads to EC in nodes 5, 6 in the CFG in **Figure 6-1** (b). According to Table 6-3, this error can be corrected by path r_4 through the procedure in

Figure 6-2. Therefore, $\mathcal{ES}(O_{r_2}) = \{3, 4\}$ and $\mathcal{ES}(O_{r_4}) = \{1, 2, 3, 4\}$ is the error correction decomposition slice taken with respect to r_4 for correcting the type of input errors.

Note that in this simple example the effect-oriented decomposition slice is not much different from the program itself. However, during our empirical studies, we find that, a program might implement more than one functional feature; thus the size of the effect-oriented decomposition slice is much smaller than the program. As such, it is useful in isolating code implementing input error correction.

Let $\mathcal{ES}(O_{r_p})$ and $\mathcal{ES}(O_{r_q})$ be two error-correction decomposition slices taken with respect to r -paths r_p and r_q (r_p and r_q are in the same program) for correcting some input errors. Adopting terms from [68], statements in $\mathcal{ES}(O_{r_p}) \cap \mathcal{ES}(O_{r_q})$ are called **error-correction dependent statements**

with respect to the two slices. Error-correction dependent statements are shared among several error-correction decomposition slices and should not be modified during the maintenance of error correction for a type of input errors or else it will affect the behaviors of error corrections for other types of input errors. Based on the definition of error-correction dependent statements, other statement and variable dependency concepts are inherited from decomposition slicing [68] as presented in Section 3.2.4.

For instance, r -paths r_1 and r_3 through the CFG in Figure 6-1 are error correction paths for correcting EC in input block <Integer + Integer> in path r_4 and EC in input block <Integer> in path r_5 , accordingly. The error-correction decomposition slice taken with respect to r_1 and r_3 are $\mathcal{ES}(O_{r_1}) = \{1, 2, 3\}$ and $\mathcal{ES}(O_{r_3}) = \{1, 2, 4, 5, 6\}$. As such, the error-correction dependent statements are $\{1, 2\}$.

6.6.2 Removing error correction for a type of input errors

During the maintenance phase, the engineer might decide that the error correction for a certain type of input errors is no longer required or some basic collections of error correction paths need to be removed. Let B_ξ be a basic collection of error correction paths for correcting the type ξ of input errors in an r -path. B_ξ might contain paths which belong to different procedures in the system. To successfully remove B_ξ from the system, at least one path in the set must be successfully removed from the containing procedure. The following corollary states the conditions in which a basic collection of error correction paths cannot be removed from a system. A software maintainer should check for this condition before deciding to remove a basic collection of error correction paths from the system.

Corollary 6-1. *Let B_ξ be a basic collection of error correction paths for correcting a type of input errors ξ . If for each r -path r_p in B_ξ such that the error-correction decomposition slice taken with respect to r_p for correcting ξ is strongly dependent on the error-correction decomposition slice taken with respect to an r -path r_q for correcting some other input error, where r_p and r_q are in the same procedure, then B_ξ cannot be removed from the system.*

This is a direct corollary of Rule 1. Let $\mathcal{ES}(O_{r_p})$ and $\mathcal{ES}(O_{r_q})$ be the error-correction decomposition slices taken with respect to r_p and r_q respectively. If $\mathcal{ES}(O_{r_p})$ is strongly dependent on $\mathcal{ES}(O_{r_q})$, where r_p and r_q in the same

procedure, then no statements can be removed from r_p because it is essential for the correction path r_q to correct some other types of input errors. As such, r_p cannot be removed from the containing procedure. If for each such path r_p in B_ξ , r_p cannot be removed from the containing program, then the basic collection of error correction paths B_ξ cannot be removed.

Table 6-3 shows that $\{r_1\}$ is a basic collection of error correction paths for correcting EC in the input block <Integer> in r-path r_5 . The error-correction decomposition slice taken with respect to r_1 is $\mathcal{ES}(O_{r_1}) = \{1, 2, 3\}$. In the process_order procedure (Figure 6-1), path r_2 is also in some other basic collections of error correction paths for correcting some other types of input errors as shown in Table 6-3. The error-correction decomposition slice taken with respect to r_2 is $\mathcal{ES}(O_{r_2}) = \{1, 2, 3, 4, 5, 6\}$. Clearly, $\mathcal{ES}(O_{r_1})$ is strongly dependent on $\mathcal{ES}(O_{r_2})$; thus we cannot remove r_1 from procedure process_order as well as from the basic collection $\{r_1\}$.

Corollary 6-2. *Let B_ξ be a basic collection of error correction paths for correcting a type of input errors. If there exists a path r_p in B_ξ such that the error-correction decomposition slice taken with respect to r_p for correcting the input error is maximal then B_ξ can be completely removed from the system.*

This is another direct corollary of Rule 1. If the error-correction decomposition taken with respect to a correction path is maximal, there are some independent statements in the slice which can be removed. It is obvious that if at least one statement is removed from a decomposition slice, the path is considered removed from the respective program; thus B_ξ is considered removed.

To remove a basic collection of error correction paths for correcting a type of input errors we first check whether B_ξ is removable by verifying against Corollary 6-1 and Corollary 6-2. If so, for each path r_p in B_ξ , we compute the error-correction decomposition slice taken with respect to r_p . If the error-correction decomposition is a maximal decomposition slice, we compute the set of independent statements in the decomposition slice. To remove r_p , we can remove all the independent statements. Note that when deleting a statement n_s , the following statements must also be deleted:

1. All the statements which are transitively control dependent on n_s .
2. All the statements which reference variables defined by n_s .

If at least one path in the basic collection of error correction paths can be removed, the collection is considered removed from the application.

For example, assume that table Order is non-updatable. As such, the external effect “insert into table Order” is not reversible. That means, once a record is inserted into the table, it is not possible to remove it. Subsequently, the developer wants to remove the basic collection of error correction paths for correcting EC in the input block <OrderHeader> in procedure process_order in **Figure 6-1**. According to Corollary 6-1, the developer needs to remove path r_5 from the all the basic collections of error correction paths that contain r_5 . The error-correction decomposition slice taken with respect to r_5 in procedure cancel_order is $\mathcal{ES}(O_{r_5}) = \{1, 2, 3, 4\}$. Since r_5 is the only r-path through the procedure in Figure 6-3, $\mathcal{ES}(O_{r_5})$ is a maximal decomposition slice and all the statements in the slice are independent statements. As such, to remove r_5 , we have to remove statements 1, 2, 3, 4 and 5 from procedure cancel_order in Figure 6-3. Therefore, essentially, we have completely remove procedure cancel_order from the E-Order system.

6.6.3 Adding error correction for a type of input errors

Basically, the problem of adding a new error correction for correcting a type of input errors to a system is equivalent to adding a code segment, which implements the new error correction, into an existing decomposition slice. This problem has been well addressed by Gallagher et al. [68]. A detailed example is also given in the paper. The basic idea is that the new code segment must follow modification Rule 2 and Rule 4. To satisfy Rule 2, for each variable used in the new code segment, which is dependent in the existing slice, we create a new local variable, whose name does not conflict with any variable in the existing slice. We copy the value of the dependent variable to the new local variable and replace all the uses of the former with the latter in the new code segment. To satisfy Rule 4, the software maintainer must ensure that a new control statement does not surround any dependent statements when specifying the position for it in the decomposition slice. A code segment satisfying Rule 2 and Rule 4 can be automatically merged with the old code by applying the algorithm given in Section 3.2.4.

6.7 The prototype system

A prototype system called EmRTM (Empirical Recovery, Testing and Maintenance) for the recovery, testing and maintenance of input error correction feature following the approaches presented in Section 6.4, Section 6.5 and Section 6.6. The architectural diagram is shown in Figure 6-6. EmRTM utilizes (1) program analysis information provided by EmAnalyzer to recover

input error correction feature (2) test data generation function provided by EmValidator to generate structural test cases for input error correction and (3) slicing functions of Kaveri to perform program slicing and assist in maintenance.

EmRTM consists of four main components: **EmAnalyzer**, **IEC** (Input Error Correction) Recovery, **DeSlice** (Decomposition slicing) and **Black&White**.

EmAnalyzer is based mainly on existing research algorithms and tools including Soot, basis path finding algorithms and Dot graph. The component provides information for Step 1 of the approach for automated recovery of the input error correction proposed in Section 6.4. Note that the Soot framework in EmAnalyzer has been extended to **Soot⁺⁺** which includes also a simplified open-source HTML parser [95], a SQL query analyzer. A PHP parser [166] has also been integrated into the prototype system. The PHP parser analyzes programs written in PHP and generates a CFG for each program in a system. Each node in the CFG is also attached with control and data dependency information.

Figure 6-6. The prototype system

A program in a database application page might contain markup languages (HTML) and SQL syntax. The HTML parser crawls through the source code to identify code segments which contain HTML code and extract information on input variables and their values from input forms. Code segments which might contain HTML code include HTML blocks, statements which assign or define values for strings, output statements such as print and echo statements and some other functions which produce HTML code.

The SQL query analyzer is extended from an open-source SQL parser [196], which provides information about a SQL statement such as the statement type (i.e. insert, update, delete, etc...), what tables and attributes are used in the statement. For systems like web applications, some SQL statements are

dynamically generated at run-time. As such, the external effect raised by a SQL statement varies from execution to execution of the web application. Therefore, an original SQL parser could not solve this problem. This problem, however, cannot be solved easily with dynamic analysis. Dynamic analysis is based on executing the application on particular inputs, monitoring the execution trace and analyzing the dynamic information. Normally, the code is instrumented to gather information during execution. Though dynamic analysis can gather run-time information, results from several executions cannot be generalized to all executions. Moreover, dynamic analysis relies very much on the design of inputs to force an execution trace. We have proposed an approach [149] using symbolic execution and heuristic rules to identify all the external effects in a system automatically. Each r-path is symbolically executed by applying a set of symbolic evaluation rules to derive a set of symbolic expressions for the possible types of external effects raised by the path. The **symbolic evaluation rules** speed up and simplify the symbolic evaluation process. The types of external effects are then inferred from the symbolic expressions by applying the **inference rules** proposed in [149]. Basically, the approach is still based on static analysis. However, the approach is facilitated by heuristic rules which make its performance much better than other approaches using symbolic evaluation. The algorithm for identifying external effects in a system has been implemented for the SQL analyzer.

The IEC Recovery component consists of three modules: **EE Inference** (Effect Error), **BCECP Constructor** (Basic Collection of Error Correction Paths) and **PIEC Collector** (Programs with Input Errors Correctable), each of which implements Step 2, 3 and 4 in the proposed recovery approach in Section 6.4, respectively. This component produces two levels of reports.

- A **system-level report** lists all the programs in a system and gives overview information on the input error correction features implemented in each program.
- A **program-level report** lists all the possible input errors in a program. For each input error, it shows all the resulting effect errors as well as the basic error-correction sets of the input error. Table 6-3 is a simplified program-level report for the E-Order system.

The core of DeSlice is a slicing tool for computing the various decomposition slices for the input error correction features. Slicing of Java programs is done through the use of the Kaveri tool [102] from Kasas State University, a program slicing Eclipse plug-in. It utilizes the Indus Java slicing tool to calculate slices of Java programs and then displays the result visually in the Eclipse Java editor. In addition, we implement two small tools namely IEC Removing and IEC Adding following the approach presented in Section 6.6.2 and 6.6.3, respectively. **IEC Removing** automatically removes a basic collection of error

correction paths for correcting a type of input errors specified by users from the code. It notifies the user in the case where the set cannot be removed. Currently, **IEC Adding** inputs a text file which contains the code of the new feature and specifies the program to which it will be added. The tool automatically merges this code segment with the program following the algorithm proposed by Gallagher et al. [68].

The Black&White component implements the Black&White testing approach proposed in Section 6.5. Black&White is composed of **IEC Partition** and **IEC TestGen** which implement the first and the last step of the Black&White approach, respectively. IEC Partition takes the recovered information produced by IEC Recovery and outputs all the CIE classes. For each CIE class, IEC Partition provides the input structure, the path domain and a list of all the basic collections of error correction paths for correcting each type of input errors in the class. The formation of SIE classes and reconciliation of SIE partition and CIE partition are done manually. After implementation has been reconciled with the specification, the reconciled implementation is re-input into EmRTM to recover the refined implementation of input error correction, subsequently, the reconciled CIE classes. IEC TestGen generates test cases for each CIE class based on the last step of the Black&White approach. IEC TestGen extends the TestGen component of EmValidator to assist users in test data generation for CIE classes. As such, test data generation is done semi-automatically.

6.8 Validation and evaluation

In this section, we report the result of binominal testing conducted to statistically validate the empirical properties presented in Section 6.3.2 and 6.3.3. We also present case studies to evaluate the usefulness of the proposed approaches to testing and maintenance of input error correction in database applications.

6.8.1 Statistical validation

6.8.1.1 Experiment design

Table 6-7. Description of systems for hypothesis testing

System	KLOC	Source	Description
Java systems			
TinaPOS	49	Open source	Point of sale application
GFP	60		Personal finance management
eLawOffice	23		Data management for law firms
ICEcore	151		Team collaboration software

Webberchat	6	Student	Customer support for chat application
Shoddy Battle	37		Game application
Smart_project	12		Online project management system
Smart_exam	21		Exam assistant management system
GradeBook	8		Automatic grading system
Healthcare	137	Industrial	Health care system
PHP systems			
SchoolMate	11	Open source	School Management system
Open Bib	58		Automated library system
WebCalendar	147		Schedule management application
CMapp	48		Contract management system
NetOffice	42		Online project management environment
Vacation-Sched	19		Online vacation scheduling system

We have conducted binomial tests [77] to validate each of the proposed empirical property. For each empirical property $\mathfrak{R}$, the alternate hypothesis states that $\mathfrak{R}$ holds for equal or more than 98% of the cases. And, therefore, the null hypothesis states that $\mathfrak{R}$ holds for less than 98% of the cases.

That is:

$$H_0 \text{ (null hypothesis): } p(\mathfrak{R} \text{ holds}) < 0.98$$

$$H_1 \text{ (alternate hypothesis): } p(\mathfrak{R} \text{ holds}) \geq 0.98$$

The binomial test statistics z is computed as follows:

$$z = \frac{X/n - p}{\sqrt{(p(1-p)/n)}}$$

where n is the total sample size for the test and X is the number of cases that support the alternative hypothesis H_1 . Taking 0.005 as the type I error, if $z > 2.408$, we reject the null hypothesis, otherwise, we accept the hypothesis.

According to the Tiobe programming index [1] in January 2009, Java is the most popular programming language. As such, we conduct our binomial tests on Java programs. There are two main benefits of choosing Java as the main programming language for our experiments. Firstly, we can collect a richer set of sample Java programs from various application domains. Secondly, we have found many program analysis tools for Java programs. These tools provide strong supports which further facilitate our experiments.

The sample programs for our experiments are real programs which have been carefully chosen to minimize all the threats to validity and to maximize the coverage of the experiment. More specifically, our sample programs are taken from three main sources including (1) academic/research institute, (2) industrial project and (3) open-source. This ensures that the sample programs are written by programmers of different expertise levels. Software taken from the first source is normally developed by undergraduate students, master/PhD students and researchers. Software taken from the industrial projects is generally developed by junior and senior software engineers of two to three years of experience. For open-source software, it is however impossible to access the expertise level of the developers as this information is not available. In this case, we ensure that no two sample programs are taken from the same group of developers. Furthermore, all the sample programs are selected from different application domains with reasonable size.

Besides Java programs, we also collect sample programs written in PHP from open-source only. Binomial tests are also conducted on these PHP program to demonstrate the validity of the empirical properties in other programming languages. The results obtained during the binomial tests on both PHP and Java are compared to see if they are consistent.

Table 6-7 gives a brief description of each Java and PHP system. All the open-source systems can be downloaded from [195] by searching for the systems' names. Smart_project, Smart-exam and Gradebook are developed by third year computer engineering students. Healthcare is industrial project written by programmers of 2-3 years of experience.

6.8.1.2 Experiment execution and results

As Property 6-1, Property 6-3 and Property 6-5 are invariant properties, we use the prototype system EmRTM for the validation of Property 6-2, Property 6-4 and Property 6-6.

Table 6-8. Sample size for testing empirical properties

Property Systems	Property 6-2	Property 6-4	Property 6-6
Java Systems			
TinaPOS	45	147	56
GFP	52	168	54
eLawOffice	47	144	60
ICEcore	78	243	198
Webberchat	54	168	90
Shoddy Battle	91	279	78

Smart_project	74	234	12
Smart_exam	46	147	55
GradeBook	89	288	54
NCS	134	435	378
Total	710	2253	1035
PHP systems			
SchoolMate	45	147	56
OpenBiblio	52	168	54
WebCalendar	47	144	60
NetOffice	78	243	198
Cmapp	54	168	90
Vacation-shed	91	279	78
Total	367	1149	536

The sample size for testing each property is given in Table 6-8. The validation procedure for each property is as follows:

1. **Property 6-2:** Each r-path through the CFG of each procedure forms a case for testing Property 6-2. For each r-path, for each type ξ of input errors in the r-path based on Property 6-2, EmRTM infers the all the possible effect errors resulting from the type of input errors. Results obtained from EmRTM are then double checked with the assistance of EmValidator. Basically, for each type of input errors, TestGen assists us in designing a test case to intentionally create an input error of the type. JTracer instruments all the associated output statements so that we can monitor the external effects raised due to the input error created. If the results obtained from EmRTM are confirmatory for all the effect errors computed by EmRTM earlier, we conclude that Property 6-2 holds for the case.
2. **Property 6-4:** Each type of input errors in each r-path through the CFG of each procedure in the sample forms a case for the testing Property 6-4. For each type of input errors, based on Property 6-4, EmRTM computes the basic collection of error correction paths for correcting the type of input errors. This result is double-checked for their correctness. We design a test case with the assistance of EmValidator to enforce the occurrence of the input error. For each path in the basic collection of error correction paths computed by EmRTM, we also design a test case to exercise the path. We then execute the paths in the basic collection of error correction paths and check whether the input error is indeed corrected. If the result is confirmatory, then we conclude that Property 6-4 holds for the type of input errors.

3. **Property 6-6:** Each procedure in the sample to which some inputs are accessed forms a case for testing of Property 6-6. EmRTM infers the correctability for input errors of each procedure based on Property 6-6. For cases which are inferred from the first condition of Property 6-6, no additional validation needed since all the basic collections have been confirmed correct through the testing of Property 6-4. For cases which are inferred from the second condition, we manually design test cases with the assistance of EmValidator and execute the procedure to check the result obtained by EmRTM. If the results are confirmatory, then we concluded that Property 6-6 holds for the procedure.

Table 6-9 provides the z-scores of all the binomial tests on both Java and PHP programs. Column “Violate” gives the number of cases which violate the property. The table shows that all the properties hold for all the cases in both Java and PHP sample programs. Clearly, all the z-scores for testing all the above-mentioned properties on both Java and PHP are greater than 2.408. Thus, we reject all the null hypotheses and conclude that all the empirical properties hold for more than or equal to 98 percent of all the cases at 0.5 percent level of significance. It is also concluded that the test results are consistent for both Java and PHP.

Table 6-9. Z-scores

Language	Property 6-2		Property 6-4		Property 6-6	
	Violate	Z-score	Violate	Z-score	Violate	Z-score
Java	0	3.81	0	6.78	0	4.60
PHP	0	2.74	0	4.84	0	3.31

6.8.1.3 Threats to validity

This study, like any other empirical study, has some limitations. First, the binomial tests are influenced by our choice of sample sets and the subject applications. Even though we have randomly chosen systems from various application domains and different sources, the chosen systems might not be representative of the population. However, we surmise that our conclusions will generalize since they are based on results that are consistent over a population of sample sets that are greatly varied in domain, size, language and source.

Second, the hypothesis testing was conducted on a large sample size in which the results produced by the prototyping tool were manually validated and tested with the aid of the EmValidator to check whether a particular property held for

a case. This is a tedious and error-prone task. We have spent more than one year on the validation.

6.8.2 Exploratory study 1: Testing input error correction

We conduct an exploratory study to evaluate the usefulness of the Black&White approach in testing input error correction. We show that the Black&White approach performs better than either basis path testing or equivalent class testing or their combination in term of fault detection capability. Moreover, most of the faults can be detected by the Black&White approach during the reconciliation stage.

Three Msc. students are involved in the study. Each student is assigned the task of testing the input error correction feature implemented in a Java database application using a different approach. The first student uses a structural testing approach called basis path testing [157]. The second student uses a functional testing approach called equivalent class testing [108]. The last student uses the Black&White testing approach described in Section 6.5.

The Java application GFP [72] is chosen for the study. GFP is a personal finance management system which is designed to help people with little finance knowledge to manage their finances. Functions are provided for managing a variety of financial instruments including checkbooks, credit cards, assets, investments and budgets. The system also manages financial transactions and related contacts. Altogether, there are 988 classes with 59777LOCs.

We try to set up an environment not to favor the Black&White approach as much as possible. All students are equipped with the EmAnalyzer and EmValidator tool to facilitate program analysis, program understanding and test data generation. The major difficulty in this study is the lack of explicit specification describing the requirements for input error correction in the GFP application. To address this issue, we prepare an informal specification to explicitly describe the input error correction required by GFP. All students are also provided with this specification.

Table 6-10. Fault detected by each approach

Fault type	Seeded	W	B	B&W	$W \cap B$
1 - Completely missing	42	23	42	42	42
2 - Partially missing	42	26	42	42	42
3 - Redundancy	42	0	29	42	29
4 - Predicate error	42	38	22	42	38
5 - Wrong value	42	40	35	42	40

6 – Side-effect error	42	39	0	42	39
7 – Non-specified error	42	0	0	42	0
Total	252	166	170	252	230

Legend: W – Basis path testing
 B – Equivalence class testing
 B&W – Black&White testing

We wish to access the performance of each testing approach with respect to the fault detection capability. Such faults are not available in the selected application. As such, we manually insert faults into the source code. A team of final year undergraduate students work together with one author to design and randomly insert faults into the source code. We classify the following common types of faults in implementing input error correction for a system.

1. *Completely missing*: A program whose input error is supposed to be correctable but implemented as not correctable.
2. *Partially missing*: The provision for correcting a type of user input errors is partially missing.
3. *Redundancy*: A program whose input error is supposed to be not correctable but implemented as correctable
4. *Predicate error*: The input error correction for correcting a type of user input errors is provided but unable to correct the type of input errors due to some predicate error in the coding.
5. *Wrong value*: The input error correction for correcting a type of user input errors is provided but attributes of the some database records are updated wrongly due to some errors in the detailed computation.
6. *Side-effect*: The input error correction for correcting a type of user input errors is provided. However, while correcting the input error of the type it also creates other unwanted side effects such as modifying other non-related database tables.
7. *Non-specified*: The input error correction implemented for some programs which are not specified at all in the requirement which might cause severe damage to the database.

42 faults of each type are inserted in GFP source code. As such, altogether 252 faults have been seeded into the system. The testing results obtained by each student are accessed based on the number of faults detected.

When comparing the results obtained by each student with different testing approaches, the outcome might be confounded if the background of the participating students on fault detection and the testing approaches are different. This issue can be addressed by considering the concepts of informational equivalence. Larkin and Simon [131] define informational equivalence thus: “two representations are informationally equivalence if all information in one is

also inferable from the other and vice versa.” The need to control for informational equivalence has been raised as a concern by Siau [188]. Siau has highlighted the importance of informational equivalence in software engineering experiments as “it ensures that we are not comparing apples to oranges”. According to Siau, different information models can be made informational equivalent through training and experience. As such, to ensure the informational equivalence in the test results obtained by each student, we provide them with sufficient training. A three-day tutorial has been conducted to help the participating students understand the concept of fault detection capability and how different testing techniques (Black&White, basis path testing and equivalent class testing) can be used to detect fault in software systems. We also give student concrete example and code for each type of fault in implementing the input error correction feature in software systems.

As shown in Table 6-10, the Black&White approach achieves the greatest fault detection capability with 100% of the seeded faults detected followed by equivalence class testing with 67.5% of the seeded faults detected. Basis path testing detects the least number of faults with only 65.9% of seeded faults detected. The number of seeded faults detected by both basis path testing and equivalence class testing is shown in the last column of Table 6-10. Altogether, there are 230 faults detected, which accounts for 91.3% of all the seeded faults.

Further investigation into the faults not detected by both basis path testing and equivalence class testing approaches reveals that both approaches fail to detect all the Non-specified faults. This is expected as basis path can only tell whether input error correction is correctly implemented. It is much more difficult for white-box testers to figure out that a certain feature is redundant because they normally generate test cases from source code. On the other hand, black-box testers only go through the specification and design test cases to test the functions described in the specification. The approach also cannot discover whether a function which is not specified in the requirement has been redundantly implemented in the code.

Basis path testing fails to detect some of the “missing” faults (types 1 and 2). Moreover, for those programs which are self-correctable, structural testing could not tell whether their input error corrections are missing from the implementation. Similar to Non-specified faults, basis path testing also fails to detect all the Redundancy faults.

Some Redundancy faults are also not detected by equivalent class testing. We believe that this is reasonable as making sure that some functions which are not supposed to be implemented are indeed not in the implementation is a rather tricky task.

Equivalence class testing fails to detect all the Side-effect faults. Moreover, it also missed some faults that due to predicate errors. This is also expected as all these errors are tightly associated with code characteristics. Side-effect faults cannot be detected by equivalence class testing as the approach only makes sure that for a given input value, the output is same as the expected output. Any other side effects could not be detected. Surprisingly, some of these faults are also not detected by basis path testing as shown in Table 6-10.

A major difference between the Black&White approach and the other two approaches observed from this study is that except faults of type 5 (Wrong value), the Black&White approach can detect all the seeded faults during its reconciliation before testing commences. These errors can be rectified before spending time on testing. We found that this saves much time in the whole testing process (including time taken for reconciliation). Basis path testing and equivalence class testing can only detect them during testing. As such, only one test suite is needed for the Black&White approach.

From this study, it is concluded that the Black&White approach outperforms either the functional approach (equivalence class testing) or the structural approach (basis path testing) or their combination. Only by combing information from code and specification, certain types of faults such as Non-specified faults can only be detected.

6.8.3 Exploratory study 2: Aiding maintenance

At the heart of our approach is the recovery of input error correction to assist program understanding and maintenance of the feature. Therefore, we conduct an exploratory study to assess the usefulness of our approach by mimicking a program understanding task assuming no previous knowledge of the program to identify the feature.

We choose E-Tutor and JMatchMaker as the subject systems for our study. Both systems are written in Java. E-Tutor is an e-learning tool that can be used at educational institutions to manage course work, particularly homework submission and return, and encourage online education. E-Tutor contains 17269 LOCs decomposing in 137 classes. JMatchMaker is a match-making system designed to help students in a university to find their ideal partners. This is a student project with 19574 LOCs containing in 209 classes. Both projects were developed by final year under graduate students. No integration testing has been done on either system. The incompleteness of the input error correction makes them interesting for our study.

We formulate the following program understanding task: *“identify possible input errors occur during the execution of the system which are uncorrectable*

and locate the code segments which are affected by the input errors". We tackle the program understanding task manually with the participation of two groups of programmers and systematically with the proposed approach using EmRTM tool. We then compare and discuss the results obtained from the two groups with the ones produced automatically by EmRTM. The first group consists of two final year students and the second group consists of two senior programmers with two years of experience. We provide them with EmAnalyzer which can be used for program analysis and visualization of analysis results. We also provide them with the DeSlice component of the EmRTM tool to aid the participants in isolating particular code segments of their interests. To ensure that the study environment does not favor the proposed approach and to control the informational equivalence in the results obtained by each group, we give the participants a tutorial on input error correction. We explain clearly the concept of input error and effect error. We show the participants how input errors in one program can be corrected by executing other programs. We also provide full documentation of the two systems.

Basically, there is no documentation associated with both of the systems. Therefore, we have to thoroughly study the system and manually create the documentation for E-Tutor and JMatchMaker. We also study the source code to manually work out the set of uncorrectable input errors and the associated code segments, which are considered as the **set of expected outputs** of the program understanding task. The precision of the results obtained by each participant (including EmRTM) will be assessed by comparing with the expected outputs.

For each input error concluded uncorrectable by each group, we check whether the input error is in the set of expected outputs. If so, we call this a **true positive** case; otherwise, it is a **false positive** case. Those input errors which are in the set of expected outputs (i.e., they are uncorrectable), however undiscovered by the participants are referred to as **true negative** cases. The same checking procedure is also applied to results obtained by EmRTM.

Table 6-11 presents the results of the study. Columns t , uncorrectable, TN, FP, gives the time spent by each group to conduct the task for both E-Tutor and JMatchMaker, the total number of uncorrectable input errors produced by each group, the number of false positive cases and the number of true positive cases, respectively.

We compute the precision and recall of the results obtained by each approach by adopting the definition of precision and recall from Information Retrieval field as follows. The precision of the information produced by each group is measured by the percentage of the input errors which are concluded

uncorrectable by the group and are also true positive. The precision is computed by the following formula:

$$Precision = \frac{uncorrectable - falsePositive}{uncorrectable} \times 100\%$$

The recall (completeness) of the information produced by each group measures the ratio between the number of input errors which are concluded uncorrectable by the participant and are also true positive and the actual number of all the uncorrectable input errors in the system.

$$Recall = \frac{uncorrectable - falsePositive}{uncorrectable - falsePositive + trueNegative} \times 100\%$$

We evaluate the overall quality of information produced by each group based on the traditional *f*-measure as follow:

$$f_measure = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

F-measure weights low values of precision and recall more heavily than higher values. It is high if both precision and recall are high.

Columns Pre, Recall and *f* in Table 6-11 give the precision, recall and *f*-measure of the result obtained by each group and EmRTM, respectively. It is not surprising that the senior programmers spent only 15 hours, 10 hours less than the students, and recovered 125 uncorrectable input errors, out of which, there were five false positive cases. As a result, the information produced by the programmers achieved 90.4% precision with 72.8% recall. Overall, the *f*-measure of the information produced by the programmers was 0.81. On the other hand, the information produced by the students only gave a precision of 87.9% with 52.6% recall and overall the *f*-measure was 0.66.

Table 6-11. Study results

Participant	<i>t</i>	uncorrectable	TN	FP	Pre	Recall	<i>f</i>
Students	25 hours	91	72	11	87.9%	52.6%	0.66
Programmers	15 hours	125	41	15	90.4%	72.8%	0.81
EmRTM	13137ms	151	0	0	100.0%	0.0%	1

Legend: TN – true negative
Pre – precision
FP – false positive

The results show that even with two years experience programmers, much time must be spent on understanding the systems and the input error correction

implemented, yet, the accurate information cannot be manually recovered from the source code. However, in this case, EmRTM outputs 151 uncorrectable types of input errors with no false positive or true negative cases. Thus, the overall f -measure of the information recovered by EmRTM is one. More importantly, all this information is recovered automatically and the time spent by EmRTM was approximately 13137 (ms) for two systems.

6.9 Comparison with related work

We have proposed in this chapter an approach for the automated recovery of input error correction from source code. We also show the use of the recovered information in testing and maintenance of input error correction in database applications.

In term of input error correction, though much research has been done in the area of computer interaction [16, 86], these works only deal with human computer interaction in general. They focus more on human behavior aspect and do not cover the code characteristics of programs which implement the input error correction features. In database research, compensating transaction was introduced in long transaction and concurrency control research [70]. Compensating transaction has some similarities to our error correction program in the sense that it is for undoing the effect of a part in a long transaction, which can be treated as a transaction by itself. However, compensating transaction is introduced in the framework of long transaction for the study of its behavior and concurrency. No details on how to recover, test and maintain compensating transaction have been discussed in database research.

In the area of software engineering, input error correction has not garnered as much attention. Most of the work so far, focuses more on detecting *program* error (bug) such as: internal consistency, redundant code, NULL consistency, etc [29, 62, 223]. The most popular means to detect user input error is through input validation. Input validation refers to those functions in software that attempt to validate the syntax and correctness of user provided commands/information. These functions enforce that only valid input is accepted to raise external effect such as updating databases or displaying information to end users. Invalid input submitted is rejected and no external effect is raised. Input validation plays a key role in the control and accuracy of user input submitted to a system. It plays a vital role on the robustness of the system. However, input validation can only detect pre-execution input error. Unfortunately, there are many input errors which can only be detected after the completion of execution. The approach presented in this chapter targets at post-execution input errors as this type of errors is more difficult to detect and it requires correction. The novelty of the proposed approach lies in the use of

empirical properties of post-execution input errors. All the possible types of user input error are derived through detecting these properties from the CFG. As such, information on all the possible input errors can be used to verify whether a system has provided sufficient correction features.

Program analysis has been used to solve many problems in design recovery. However, to the best of our knowledge, we have not seen any approach that uses any of these techniques for the automated recovery of input error correction. A semi-automatic approach to locating features in source code, which combines static and dynamic analysis and uses concept analysis, is presented by Eisenbarth et al. [60]. The limitation of this approach is that it relies on domain-specific expertise, which is not always available. Wilde [212, 213] develops the Software Reconnaissance method which utilizes dynamic information to locate features in existing systems. However, the approach does not guarantee to cover all the possible features or discover all code segments associated with a functional feature. Antoniol et al. [7] adopt the idea from epidemiology and propose an approach to accurately identify features in a large multithreaded object-oriented programs. The approach is based on statistical analysis of static and dynamic data. The accuracy of the approach is very much dependent on the collection of dynamic data. Chen and Rajlich [33] propose a semi-automated approach to feature location. An abstract system dependence graph (ASDG) is used as the high level representation for a program which describes the dependencies among functions, classes and global variables. The scalability of the approach for large-scale systems might be problematic since the approach requires much manual effort. Many other approaches [50, 118, 119, 183] use graph matching techniques to locate the functional features in source code by matching between predefined patterns or abstract high-level representations of the features with the structure of the source code itself. These methods require the constructions of the patterns or the models representing the functional features.

Our proposed approach recovers CIE classes, which enable the combination of functional and structural testing for input error correction in database applications. It covers both code and specification behaviors for the target feature in a single testing process and test suite. To the best of our knowledge, no testing method that covers both code and specification behaviors has been proposed for input error correction. The reconciliation of implementation and specification presented in Section 6.5 for the Black&White approach is based on the partition analysis technique proposed by Richardson and Clark [176]. The difference between our approach and the partition analysis technique is that we do not use symbolic execution to construct the specification-based and code-based equivalence partitions. We base on the recovered input error

correction to form the code-based equivalence partition (CIE partition). For specification-based equivalence classes, any partition-based black-box testing technique can be used. We believe that the use of symbolic execution is a limitation of the partition analysis approach as complicated functional features such as input error correction cannot be represented using symbolic expressions.

Software maintainers are faced with the problem of understanding the existing software and making changes without having negative impacts on the unchanged parts. Gallagher et al. [68] propose using the decomposition slicing technique to aid in making changes to a piece of software without unwanted side effect. A decomposition slice captures the computation performed by a program as a whole on a variable of interest. We adopt this idea to decompose a program in term of input error correction implemented by proposing effect-oriented decomposition slicing. While the decomposition slice [68] captures all the computations of a variable regardless of the program location, an effect-oriented decomposition slice captures all the statements involving in the implementation of an input error correction feature.

6.10 Summary

We have proposed in this chapter an approach for the automated recovery of input error correction from source code. Though the proposed approach uses the basic technique of program analysis for the automated recovery of input error correction, there is a major theoretical difference between the proposed approach and other approaches that use program analysis for the automated functional features recovery. The proposed approach is developed through the integration of invariant and empirical properties. All the empirical properties have been validated statistically with samples collected from a wide range of database applications. The validation gives evidence that all the empirical properties hold for more than 98 percent of all the cases at 0.5 level of significance. In fact, we have not found any case in which any of our empirical properties fails. Based on the empirical properties, our approach automatically recovers the implementation of input error correction from source code in four steps.

We have further proposed the Black&White approach to test input error correction in database applications. Black&White forms the code-based equivalence partition based on the recovered input error correction from source code. This code-based equivalence partition enables the reconciliation of implementation and specification as a specification-based equivalence partition can be formed through using any existing partition-based black-box testing technique. Our study has shown that the Black&White testing strategy outperforms either basis path testing or equivalence class testing or their

combination in term of fault detection capability. More importantly, most of the faults are detected during the reconciliation stage.

In this chapter, we have also shown that the recovered input error correction can also be used to aid in the maintenance of this feature. We have introduced the concept of an effect-oriented decomposition slice, which is adopted from the decomposition slicing technique [68] to decompose a program with respect to the input error correction. As such, removing or adding an error correction feature from and to a program can be done easily without introducing ripple effects to other features. Our study has shown that without using EmRTM, even for a programmer with two years of experience, much time and effort must be spent on understanding the system, yet, accurate information cannot be fully recovered.

Chapter 7

DATABASE INTERACTION EXTRACTION FOR WEB APPLICATIONS

7.1 Introduction

Nowadays, web applications have become fundamental parts, tightly integrated into many e-systems. A web application is a large and complex software system comprising of many components, which are written in different programming languages and distributed over the network. A large number of web applications utilize a Database Management System (DBMS) and one or more databases. They are also referred to as web database applications. Due to the dominance of the relational database in existing web database applications, in the rest of this chapter, we will use the term “web application” to refer to a web application that uses relational databases.

A program in a web application normally interacts with databases through statements which execute the SQL data manipulation language (DML) operations such as select, insert, update or delete. For web database applications, it is clear that database interactions are among the most essential functional features. As such, information on database interactions is useful in testing, understanding and maintenance of web applications. For example, a common problem encountered during testing of a web application is that if a database table is updated wrongly, the software engineer is interested in locating all the programs and statements which affect this interaction. Moreover, based on the extracted database interactions and related information such as program paths which lead to the interactions, test cases can be automatically or manually generated to test the database interaction feature. Information on database interactions is also useful for coverage analysis of web applications to calculate coverage requirements with respect to coverage criteria for database-driven applications such as those proposed in [112]. For another example, during the maintenance of a web application, a software engineer may need to know the answer to a question like: “*Which part of my application is*

affected if the Customer-Passwords data table is offline for maintenance?” [87]. Especially when a new DBMS is going to replace the old one, the engineer can rely on database interaction information to rectify all the affected code fragments accordingly.

Current industrial approaches to understanding and maintenance of the database interaction feature include using tools scanning through the code, consulting documentations or asking system experts. Unfortunately, in this current rapidly changing environment, web applications are facing extremely short time-to-market. As a result, many web applications are poorly documented. The software engineer has no choice but to use tools such as *grep* [73] to locate interesting code segments and then analyze them to identify all the possible database interactions. This task is tedious and should be automated to increase the productivity and reliability of the maintenance of web applications.

The difficulties in extracting database interactions from source code mainly result from the dynamic nature of web applications which are caused by statements generated only at run-time and dependent on user input. As such, the database interactions of a statement or a program vary from execution to execution of the web application. This problem, however, can neither be solved completely by static nor dynamic program analysis. Recent research work on locating functional features in source code which is based on static program analysis [60, 130, 212], cannot be applied directly to extract database interactions because these approaches do not differentiate between different types of interactions. That means, they can only locate statements which execute some database interactions but they cannot infer what possible types of database interactions that can be executed by the statements. On the other hand, dynamic analysis is based on executing the application on particular inputs, monitoring the execution trace and analyzing the dynamic information. Normally, the code is instrumented to collect runtime information. Though dynamic analysis is based on runtime information, results from several executions cannot be generalized to all executions. Moreover, dynamic analysis relies very much on the design of inputs to force an execution trace.

To overcome this problem, we propose in this chapter an empirical-based approach using symbolic execution and inference rules to extract all the possible database interactions automatically. The basic idea behind our approach is that we identify all the paths, whose executions could lead to database interactions. Each path is then symbolically executed by applying the proposed symbolic evaluation rules to derive a set of symbolic expressions for the possible database interactions executed by the path. Actually, we use a hybrid approach that inter-mixes concrete and symbolic execution, thus speeding up the symbolic evaluation process. We also develop inference rules,

which are basically heuristic rules derived during the empirical study of database interactions, that can be used to infer all the possible database interactions from symbolic expressions.

The rest of the chapter is organized as follow. Section 7.2 presents an example application and the difficulties in extracting all the possible database interactions of a web application from source code. Section 7.3 introduces our analysis model. Section 7.4 presents a set of symbolic evaluation rules. Section 7.5 proposes an approach to automated extraction of database interactions. Section 7.6 describes the prototype system and our experiments. Section 7.7 concludes the chapter.

7.2 An example application

```
<% ...
1. if (action == NULL)
2.   printt_html("<form action = 'manage_course.jsp'>Choose action<p>" +
    "<input type=radio name='action'" +
    "value='Insert'>Register a new course<br>" +
    "<input type=radio name='action'" +
    "value='Delete'>Delete courses<br>" +
    "<input type=submit value='submit'></form>");
3. if (action.equals("Insert")){
4.   if (course != NULL && studentid != NULL){
5.     stmt.executeQuery("insert into" + course +
        "values (" + studentid + ")");
6.     stmt.executeQuery("insert into Registration values ("
        + course + "," + studentid + ")");
7.     if (course.equals("Software")){
8.       final String[] LAB = {"Design_lab", "Testing_lab", "WebE_lab"};
9.       final int LAB_SIZE = 3;
10.      for (int i = 0; i < LAB_SIZE; i++)
11.        stmt.executeQuery("insert into" + LAB[i] +
            "values (" + studentid + ")");
12.    }
13.    print_html("Couese registered");
    ...
  }
  else
13.  print_html("<form action='register_course.jsp'" +
    "method='post'...> <p>" +
    "Choose course" +
    "<input type=radio name='course'" +
    "value='Software'>Software Engineering<br>" +
    "<input type=radio name='course'" +
    "value='Database'>Database Systems<br>" +
    "<input type=radio name='course'" +
    "value='Network'>Computer Networks<p>" +
    "Student ID: <input type=text name='studentid'><br>" +
    "<input type=submit value='submit'>" +
    "</form>");
  }
...
%>
```

Figure 7-1. register_course.jsp

In order to make our discussion concrete, we describe part of a simple application: Online Course Management System – a student project written in Java. Figure 7-1 and Figure 7-2 show the simplified source code of two programs in the application: `register_course.jsp` and `delete_course.jsp`.

The `register_course.jsp` page is displayed after the student logs into the system. The page displays a form in line 2, where the user can select an action: “Register a new course” or “Delete an existing course”. If the option “Register a new course” is selected, another form is displayed (line 13). The user can select a course and fill in the form his/her student id.

The page generates a string containing an SQL insert query, based on user inputs and then sends that query to the web server through the statement `executeQuery()` function in line 5, which inserts the student into the database table for that course. There are three tables namely Software, Database and Network which store the list of students attending Software Engineering, Database System and Computer Networks course respectively. It also forms and sends another query (line 6) to insert the student and the registered course into the Registrations table. This table maintains all the registration records. If a student registers Software Engineering, he/she is also required to take three additional laboratory classes including software design, testing and web engineering (line 8). The array `LAB` in line 8 includes the name of the three tables `Design_lab`, `Testing_lab`, `WebE_lab`, which stores the list of students who have to attend the respective lab class. Thus, the for-loop (line 10) iteratively adds the student into the list of each laboratory section by inserting the `studentid` into the respective table.

```
<% ...
1.  if (action.equals("Delete")){
2.      ArrayList course = (ArrayList)getPostData("course");
3.      String id = getPostData("studentid");
4.      for (int i = 0; i < course.size(); i++){
5.          stmt.executeQuery("delete from " + course.get(i) +
                             "where studentid = " + id);
6.          stmt.executeQuery("delete from Registrations where studentid = "
                             + id "and course = " + course.get(i));
      }
    }
%>
```

Figure 7-2. `delete_course.jsp`

If the option “Delete courses” is selected, the page `delete_course.jsp` will be loaded. This page allows a student to delete a number of registered courses.

“POST” data on student ids and the course names are collected into the variable `id` (line 3) and the array `course` (line 2) respectively. The procedure `getPostData()` returns the POST data given the field name. The for-loop (line 4) depends on the number of courses to be deleted. For each course, this page deletes the record of the student from the database table for the course (line 5) as well as the record of the student and the course from the Registration table (line 6).

Example 7-1. To illustrate the problems in extracting all possible database interactions from source code, take the “`register_course.jsp`” page as an example. Applying any of the current functional features extraction techniques, in general, we obtain the following statements for the database interaction feature:

- `stmt.executeQuery("insert into" + course + "values (" + studentid + ")")` (line 5)
- `stmt.executeQuery("insert into Registration values (" + course + ", " + studentid + ")")` (line 6)
- `stmt.executeQuery("insert into" + LAB[i] + "values (" + studentid + ")")` (line 11)

In line 6, we can confirm that there is only one interaction, which is “insert into Registration”. However, in lines 5 and 11, we do not know about the possible actual database interactions performed at each statement. Intuitively, there are three possible different interactions at line 5 namely “insert into Software”, “insert into Database” or “insert into Network” because the variable `course` can take three possible values including “Software”, “Database” and “Network”. Similarly, at line 11, there are also three possible interactions including “insert into Design_lab”, “insert into Testing_lab” or “insert into WebE_lab”. This is because different execution paths through the same statement in a program might lead to different interactions with the database being executed at the statement. Even the same execution path with different user inputs might also lead to different database interactions.

7.3 Analysis model

We propose in this section a contracted version of the CFG called Interaction Flow Graph (IFG) as the analysis model for our approach. Before introducing the IFG, we shall review and define a few terms which will be used throughout this chapter.

Without the loss of generality, in this chapter, we will only focus on the SQL DML operations of **select**, **insert**, **update** and **delete**. However, the proposed

approach can be applied equally to any DML operation. As such, there are two key parameters which determine a type of interactions namely **operationType** and **tableName** where operationType is the type of operations including select, insert, update or delete and tableName is the name of the database table.

A CFG node which executes, in an arbitrary run of the procedure, one of the SQL data manipulation language operations select, insert, update or delete is defined as an **interaction node (i-node)**. A database interaction in a procedure P_T corresponds to an execution of an i-node in the CFG of P_T . I-nodes are basically database access statements, each of which corresponds to one or more database interactions. For example, node 5 in the CFG in Figure 7-3 is an i-node, which corresponds to three possible interactions including “insert into Software”, “insert into Database” and “insert into Network”.

A **representative path (r-path)** in a CFG is a complete path in the CFG, such that any loop included is iterated with exactly one time. Representative paths have been defined in Section 6.3.3.2. In opposing to total number of all possible paths, the total number of r-paths in the CFG is always finite. An r-path is thus used to represent all the paths in a CFG which is identical to the r-path except iterating some loops included in the r-path with different positive number of times.

Figure 7-3. CFG of register_course.jsp

In the CFG shown in Figure 7-3, the r-path (begin, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 10, 12, end) represents all the paths through the CFG in the set:

{(begin, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 10, 12, end),
 (begin, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 10, 11, 10, 12, end),...,
 (begin, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 10...10, 12, end),...}.

On the other hand, the r-path (begin, 1, 2, 3, 4, 13, end) represents only itself as there is no loop in the r-path. An r-path which contains some i-nodes is called an **interaction path (i-path)**. From this definition, it is obvious that any loop included in an i-path is also iterated exactly one time. In our approach, we need not to concern with the number of iterations of a loop when we first form the i-

path; therefore, the number of i-paths is also finite. In Section 7.5.1, we provide a detailed explanation for this.

In the CFG in Figure 7-3, node 5, 6 and 11 are i-nodes; hence the path (begin, 1, 3, 4, 5, 6, 7, 12, end), (begin, 1, 2, 3, 4, 5, 6, 7, 12, end), (begin, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 10, 12, end) and (begin, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 10, 12, end) are i-paths.

A program might implement more than one functional feature; thus, it might contain information which is not necessary for the extraction of database interactions. For example, in the CFG in Figure 7-3, nodes 1, 2, 12 and 13 do not affect the execution of database interactions. As such, they can be removed from the CFG. Next, we shall introduce the formal definition of the interaction flow graph.

Definition 7-1. Interaction flow graph (IFG) – Let $G = (\mathcal{N}, \mathcal{E})$ be a CFG where $\mathcal{N}$ and $\mathcal{E}$ are its set of nodes and edges respectively. Let $\mathcal{N}'$ be the set of nodes in G which has the following properties:

1. All i-nodes are in $\mathcal{N}'$.
2. The begin node and end node are in $\mathcal{N}'$.
3. If a node n_w in $\mathcal{N}'$ is control-dependent on node n_v in $\mathcal{N}$, then n_v is also in $\mathcal{N}'$.
4. If an i-node n_w in $\mathcal{N}'$ references variables defined in node n_v in $\mathcal{N}$ then n_v is also in $\mathcal{N}'$.

Let $\mathcal{E}' = \{(n_v, n_w) \mid n_v, n_w \in \mathcal{N}', n_v \neq n_w \text{ and there exists a path } (n_v, n_{x_1}, \dots, n_{x_k}, n_w) \text{ in } G \text{ such that } k \geq 0 \text{ for all } j, 1 \leq j \leq k, n_{x_j} \notin \mathcal{N}'\}$, then the flow graph $G' = (\mathcal{N}', \mathcal{E}')$ is called the Interaction Flow Graph (IFG) generated from the CFG G where $\mathcal{N}'$ and $\mathcal{E}'$ are its set of nodes and edges respectively.

From the definition, the IFG of a procedure can be obtained by removing from the initial statements which do not affect the execution of i-nodes and then form the CFG of the reduced procedure. IFG can be derived from the CFG of a procedure by calling the algorithm *createIFG* (Figure 7-4) with input is the CFG of the procedure. Intuitively, the IFG should contain only the ‘begin’ node, the ‘end’ node, i-nodes and nodes which control the executions or influence the values used by the i-nodes.

Algorithm createIFGInput: $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ Output: $\mathcal{G}' = (\mathcal{N}', \mathcal{E}')$ // Construct the set $\mathcal{N}'$ of nodes of the IFG $\mathcal{N}' = \emptyset$ $\mathcal{N}' = \mathcal{N}' \cup \{begin, end\}$ $\mathcal{N}' = \mathcal{N}' \cup \{n_i \mid n_i \text{ is an i-node in } \mathcal{G}\}$ For every node $n_i \in \mathcal{N}'$: $W = \{n_j \mid n_j \in \mathcal{N} \text{ and } n_i \text{ is control dependent on } n_j\}$ $V = \{n_k \mid n_k \in \mathcal{N} \text{ and } n_i \text{ references variables defined in } n_k\}$ $\mathcal{N}' = \mathcal{N}' \cup W \cup V$ // Construct the set of $\mathcal{E}'$ of edges of the IFG $\mathcal{E}' = \{(n_v, n_w) \mid n_v, n_w \in \mathcal{N}', n_v \neq n_w \text{ and there exists a path } (n_v, n_{x_1}, \dots, n_{x_k}, n_w) \text{ in } \mathcal{G} \text{ such that } k \geq 0 \text{ for all } j, 1 \leq j \leq k, n_{x_j} \notin \mathcal{N}'\}$ **Figure 7-4. Algorithm createIFG** **Figure 7-5. IFG of register_course.jsp**

Applying *createIFG* algorithm on the CFG in Figure 7-3, we obtain the IFG in Figure 7-5. For the illustration purpose, we have simplified the source code. As such the size of the IFG is not much smaller than that of the CFG. However, in real situation where a code segment may implement several functional features or contain HTML code and scripts, it is much smaller.

7.4 Hybrid symbolic execution

In our approach, we symbolically execute all the i-paths through the IFG of a program to infer all the possible database interactions in the program. However, symbolic-only execution may end up with a complicated expression, which makes it difficult to extract the interaction types. Thus, we employ a hybrid approach that combines concrete and symbolic execution. The basic idea is that during the symbolic execution of an i-path, for each statement, we check whether all the values are concrete. If so, we use concrete execution. Otherwise, if at least one value is symbolic, we do symbolic execution.

To differentiate between symbolic and concrete value, we associate with each variable or expression s_k two fields $s_k.concrete$ and $s_k.symbolic$, which store the concrete and symbolic value of s_k respectively. Note that if $s_k.concrete$ is equal to NULL and $s_k.symbolic$ is equal to NULL then variable s_k has not been evaluated. Moreover, it is impossible that at the same time $s_k.concrete$ is not equal to NULL and $s_k.symbolic$ is not equal to NULL because a variable or expression can only have either concrete or symbolic value at a time. If $s_k.symbolic$ is not equal to NULL (which means $s_k.concrete$ is equal to NULL), s_k is **input-dependent**. On the other hand, if $s_k.concrete$ is not equal to NULL then the value of s_k can be evaluated to a concrete value. Next, we shall present some general Symbolic evaluation rules for evaluating symbolic and concrete values of each variable during the execution of an i-path. Each rule is designed for executing each type of possible statements in an i-path. The presentation of each rule does not follow syntax of any particular programming language. The implementation of the rules for any specific language must take the syntax of each statement into consideration.

7.4.1 Binary-expression rule

Figure 7-6. Binary-expression rule

The binary-expression rule for evaluation binary symbolic expression is given in Figure 7-6. Note that in most of programming language, there is also another

type of expression, which is unary expression of the form “ $\square s_v$ ”, where $\square$ is a unary operator. Unary expression can be easily and automatically converted into binary expression. Binary-expression rule is then applied to evaluate the unary expression.

Example 7-2. In the following statement:

query = “delete from notes where project in (” + id + “)”;

the string represented by query is the concatenation of three expressions s_1 , s_2 , and s_3 where:

- $s_1.concrete$ = “delete from notes where project in (” and $s_1.symbolic$ = NULL.
- $s_2.concrete$ = NULL and $s_3.symbolic$ = id.
- $s_3.concrete$ = “)” and $s_4.symbolic$ = NULL.

Applying binary-expression rule for the RHS of the statement, first of all, s_1 and s_2 is replaced by temporary variable s_4 by applying sub-rule 1.

- $s_4.concrete$ = NULL and $s_4.symbolic$ = “delete from notes where project in (id”.

s_4 and s_3 is replaced by temporary variable s_5 by applying sub-rule 3.

- $s_5.concrete$ = NULL and $s_5.symbolic$ = “delete from notes where project in (id’”.

Finally, applying assignment-statement rule we have:

- query.concrete = NULL.
- query.symbolic = “delete from notes where project in (id’”.

7.4.2 If-statement rule

The if-statement rule is given in Figure 7-7.

Given an if-statement:

If (s_c) then {...}

Else {...}

in which s_c is an expression or a variable. If-statement imposes constraints on possible concrete values of a symbolic variable. The constraints are generated as follows:

1. If the i-path takes the “true” branch, attach a constraint (s_c) to all nodes in the i-path that are control dependent on the if-statement.
2. If the i-path takes the “false” branch, attach a constraint $\neg(s_c)$ to all nodes in the i-path that are control-dependent on the else-statement, where $\neg(s_c)$ refers to the negation of (s_c).

Figure 7-7. If-statement rule

Let s_l be an assignment-statement. s_l can take one of the following two forms:

1. $s_l = \text{Const}$
2. $s_l = s_r$

where s_l and s_r are expressions or variables and Const is a concrete value of the same data type as s_l . The symbolic and concrete values of the LHS expression of e are evaluated as follow.

1. For the first form:
 - $s_l.\text{concrete} = \text{Const}$,
 - $s_l.\text{symbolic} = \text{null}$
2. For the second form:
 - $s_l.\text{concrete} = s_r.\text{concrete}$
 - $s_l.\text{symbolic} = s_r.\text{symbolic}$

Figure 7-8. Assignment-statement rule

Example 7-3. For the program shown in Figure 7-1, whose IFG is shown in Figure 7-5, applying if-statement rule to nodes 3 and 4 in i-path $i_p = (\text{begin}, 3, 4, 5, 6, 7, \text{end})$, the following constraints are attached to node 5, 6 and 7: $\text{action.equals("Insert")}$, $\text{\$course} \neq \text{NULL}$, and $\text{\$studentid} \neq \text{NULL}$. We can rewrite those using set notations as follow: $\text{action} \in \{\text{"Insert"}\}$, $\text{course} \notin \{\text{""}\}$, $\text{studentid} \notin \{\text{""}\}$.

7.4.3 Assignment-statement rule

Assignment-statement rule is given in Figure 7-8.

Example 7-4. Consider line 5 of the code segment shown in Figure 7-1:

`query = "insert into" + course + "values (" + studentid + ")";`

The RHS of this assignment-statement can be treated as an expression which contain symbolic values $\text{\$course}$ and $\text{\$studentid}$. As such, applying the second sub-rule we obtain:

- $\text{query.concrete} = \text{NULL}$
- $\text{query.symbolic} = \text{"insert into " + course.symbolic + "values (" + studentid.symbolic + ")"}'$

However, symbolic execution of assignment statements involves more subtleties than one may expect if it involve array access expressions like $a[k]$. Below are two special cases that we need to take into consideration while handling assignment-statements.

Case 1 – LHS contains an array access expression

The problem with array access expression is that they can refer to many different symbolic variables. For example, given an array a of size n and a

symbolic index i . A simple expression like $(a[i] = 0)$ could update any value in the array a . The following **array rule** is applied to symbolically evaluate these kinds of expressions

Given an assignment statement of the form $a[k] = x$.

1. if $(k.concrete \neq \text{null})$
 Apply assignment-statement rule with $s_l = a[k]$ and $s_r = x$
2. if $(k.symbolic \neq \text{null})$
 In this case, we do not know the element that k references. Thus we add the constraint that if $k = 0$ then apply assignment-statement rule for $a[0] = x$, if $k = 1$ then apply assignment-statement rule for $a[1] = x$, etc. More precisely:
 $(k = 0 \wedge a[0] = x) \wedge \dots \wedge (k = m \wedge a[m] = x)$
 where m is the size of array a .

Figure 7-9. Array-rule 1

Example 7-5. Line 8 in the program given in Figure 7-1 is basically equivalent to the following three assignment statements (this can be easily achieved through rewrite rules incorporated into compilers):

LAB[1] = "Design_lab"; LAB[2] = "Testing_lab"; LAB[3] = "WebE_lab".

Applying assignment-statement rule with the LHS being an array access expression we obtain:

LAB[1].concrete = "Design_lab";
 LAB[2].concrete = "Testing_lab";
 LAB[3].concrete = "WebE_lab".

Case 2 – RHS contains an array access expression

Given an assignment statement of the form $x = a[k]$.

1. if $(k.concrete \neq \text{null})$, apply assignment rule with $s_l = x$ and $s_r = a[k]$
2. if $(k.symbolic \neq \text{null})$, add the following constraint:
 $(k = 0 \wedge x = a[0]) \wedge \dots \wedge (k = m \wedge x = a[m])$
 where m is the size of the array a .

Figure 7-10. Array-rule 2

Example 7-6. Statement 6 in the program given in Figure 7-2 contains an assignment statement with the RHS being an array access expression:

course = course[i]

In this case, the following constraints will be attached to the statement

$$\begin{aligned}
& (i = 0 \wedge \text{course} = \text{course}[0]) \wedge \\
& (i = 1 \wedge \text{course} = \text{course}[1]) \wedge \dots \wedge \\
& (i = \text{LAB_SIZE} - 1 \wedge \text{course} = \text{course}[\text{LAB_SIZE} - 1])
\end{aligned}$$

7.4.4 Loop-statement rule

During the symbolic execution, the number of iterations might be changed. This is because some variables may turn out to have concrete values during symbolic execution. In this case, the i -path is modified to reflect the change. This is indicated in the first sub-rule of the loop-statement rule where the i -path i_x is modified to reflect the change in the number of iterations.

Let n_l be a loop-statement

$$\text{while } (v_l < s_e) \text{ do } \{ \dots \}$$

in which v_l is the variable which controls the number of iterations of the loop and s_e is a variable or an expression. The i -path i_x containing n_l has to be modified as follows:

1. If ($s_e.\text{concrete} \neq \text{NULL}$): n_l has to iterates $s_e.\text{concrete}$ times. Therefore, i_x will be modified such that it contains $s_e.\text{concrete}$ iterations of n_l .
2. Otherwise: no change has to be made to i_x

Figure 7-11. Loop-statement rule

Example 7-7. Consider path $i_x = (\text{begin}, 3, 4, 5, 6, 7, 8, 9, 10, 11, 10, \text{end})$ through the IFG in Figure 7-5. The for-loop in node 10 contains symbolic value, `LAB_SIZE`. Applying the assignment-statement rule for node 9 we obtain `LAB_SIZE.concrete = 3`. Therefore, path i_x is modified such that it contains 3 iterations of the loop:

$$i_x = (\text{begin}, 3, 4, 5, 6, 7, 8, 9, 10, 11_1, 10_1, 11_2, 10_2, 11_3, 10_3, \text{end})$$

where $11_1, 10_1, 11_2, 10_2, 11_3, 10_3$ refer to the 1st, 2nd and 3rd execution of node 11 and 10 respectively. Note that in this case, the symbolic execution of i_x should continue from node 10_1 instead of starting from the beginning.

7.4.5 Function-call rule

Based on the rules for hybrid symbolic execution of binary expression, assignment statement and loop statement, the hybrid symbolic execution of function call is conducted as the normal symbolic execution for function calls [31]. In general, there are two approaches to handle function calls: the **macro-expansion approach** and the **lemma approach**. In the macro-expansion approach, the source code of the called method is inline within the caller program. This approach requires access to the source code of the called function. Whereas in the lemma approach, the called function must already be

symbolically executed. That means all the possible return values together with their constraints must be known. This obviates the need to symbolically execute all paths in the called function and does not require the access to the called method's source code.

In our approach, we combine both approaches in a *demand-driven* manner. That means, if a function is called the first time, the function is symbolically executed to determine all the possible return values. After the first time, the function will be marked as “analyzed”. Any further call made to the function, the lemma approach is used.

To further improve the performance of symbolic execution of function calls, preliminary symbolic evaluation is applied to frequently used functions to determine all the possible return values and their constraints. For example, many of the programs encountered during our experiments use the *equals()* library function to compare two strings. Most implementation of *equals()* would traverse one of the strings and would compare each character in the first string with the corresponding character in the second string and would return a value when two characters differ or when the end of a string has been reached. Thus, the routine would return to the caller approximately $2n$ times, each time with a different set of values and constraints. For this case, we predefine all the possible return values (*true*, *false*) and their associated constraints (for example, two string must be equal for the *true* value) and then mark the function as “analyzed”.

7.5 Automated extraction

Based on the analysis model and our hybrid symbolic execution technique presented in the previous section, we propose in this section an approach to extract database interactions from web applications source code automatically. The approach consists of two stages. In the first stage, all i-paths through the IFG of a given program are constructed. Each i-path is then symbolically executed to derive an expression, which represent the possible interactions at the i-node. In the second stage, a set of **Inference rules** are applied to infer all the possible types of database interactions for each symbolic expression. Each of the following sub-sections will describe each stage in detail.

7.5.1 Stage 1 – Symbolic expressions derivation

In this stage, for each procedure P_s in the web application, we follow the below steps to derive a symbolic expression for each i-node in each i-path through the IFG of the program.

1. Identify all i-paths though the IFG of P_s

2. For each i-path i_p identified in step 1, symbolically execute i_p by applying symbolic evaluation rules presented in Section 7.4.

Step 2 is straight forward. Next, we shall explain more on Step 1.

Recall that IFG contains all the nodes which influence the execution of i-nodes in the program. Therefore, in the worst case, the number of i-paths is equal to the number of database interactions extracted. For each IFG, all the i-paths can be computed as follows. Each time, we traverse through the IFG via an r-path until all such paths have been traversed. That is, we traverse through the IFG in such a way that each selection construct is traversed via one of its branches and each loop construct is traversed via one of the two alternatives following the ipath-loop rule in Figure 7-12.

Example 7-8. In the IFG in Figure 7-5, there are only two i-paths through the IFG, which is (begin, 3, 4, 5, 6, 7, end) and (begin, 3, 4, 5, 6, 7, 8, 9, 10, 11, 10, end). In the for-loop in line 10 (Figure 7-1), the number of iterations is determined by LAB_SIZE, which is a symbolic value; as such, applying the second loop-rule, the loop is skipped in the first path and traversed only once in the second i-path.

ipath-loop rule

1. If the number of iterations is a concrete value V_k , the loop is either skipped or traversed V_k times.
2. Otherwise, if the number of iterations is controlled by some symbolic values, the loop is either skipped or traversed only once.

Figure 7-12. Ipath-loop rule

In the second sub-rule of *ipath-loop* rule, by traversing the loop one time, the number of i-paths is finite; yet, essential information to extract all the possible database interactions is maintained. Below, we explain this.

1. If the number of iterations is controlled by some symbolic variables which can be evaluated to concrete values during the symbolic execution then the i-path will be modified accordingly by applying loop-statement rule (Example 7-7). All the possible interactions can then be inferred through the symbolic execution of the modified i-path.
2. If the number of iterations is controlled by some user inputs then it is impossible to identify from the code concrete values of the two parameters, operationType and tableName for executing the i-node in

each iteration as the number of iterations is unknown. As a result, there are only two possibilities:

- a. Values of the two parameters `operationType` and `tableName` in the *i*-node are concrete and identical in all iterations: In this case, we can identify all the possible interactions based on the concrete values of the parameters.
- b. At least one parameter contains symbolic values, which means they are input-dependent and can only be identified at run-time: We identify all the possible database interactions by applying Inference rules, which are going to be presented in the next stage for inferring interaction types from symbolic expressions. In general, if the parameters contain symbolic values, the rules construct a set of all the possible concrete values for each symbolic value.

Example 7-9. Consider the “delete_course.jsp” page in Figure 7-2. The for-loop (line 4) depends on the number of courses to be deleted by a student. Therefore, the number of iterations is controlled by user inputs. Line 6 illustrates point 2.a where `operationType` = “delete” and `tableName` = “Registrations”. In this case, in any iteration of the loop, the database interaction is always “delete from Registrations”. However, in the SQL statement in line 5, the parameter `tableName` = `course[i]`, thus, it contains symbolic variables and it is input-dependent. In this case, Inference rule 3 is applied to determine all the possible database tables for the parameter `tableName`. In both case, only one iteration of the for-loop is needed, thus, there is only one *i*-path $i_p = (\text{begin}, 1, 2, 3, 4, 5, 6, 4, \text{end})$ through the program

7.5.2 Stage 2 – Database Interactions Inference

After the first stage, we can derive a symbolic expression for each of the two parameters `operationType` and `tableName`. Very often, these two parameters are combined to form a SQL query string. As such, an SQL parser is normally used to extract information on each parameter from the query string. For example, in line 5 in the code shown in Figure 7-1, the following expressions can be derived for `operationType` and `tableName` from `query.symbolic`:

`operationType` = “Insert”, and `tableName` = `course`.

Below we present Inference rules for inferring the possible types of interactions from the symbolic expressions of the two parameters. Except the first inference rule, other inference rules are empirical. The statistical for the empirical rules are reported in Section 7.6.2.1.

7.5.2.1 Inference Rule 1 (Invariant)

Inference rule 1 is applied when both `operationType` and `tableName` are input-independent. In this case, there is only one possible interaction type that is `(operationType.concrete, tableName.concrete)`. For example, if `operationType.concrete` is equal to 'insert' and `tableName.concrete` is equal to 'student' then we infer that the interaction type is "insert a record into table student".

7.5.2.2 Inference Rule 2 (Empirical)

Inference rule 2 is applied when only the `tableName` parameter is input-dependent, that means:

`operationType.concrete != NULL` and `operationType.symbolic == NULL`
`tableName.concrete == NULL` and `tableName.symbolic != NULL`.

Since the `operationType` attribute can be evaluated to a concrete value, the most obvious way to infer all the possible interactions is to assume that the `tableName` can be any table in the database which permits the operation indicated by `operationType.concrete`. We shall refer to the set of possible values for the `tableName` in this first approximation as T_A .

However, this might not be a tight approximation because some interaction types might not be executed at all due to some constraints imposed on the values of symbolic variables, which is referenced by the `tableName.symbolic`. Take for illustration the code segment in Figure 7-1, after the first approximation, the set of all the possible values for the parameter `tableName` of the query in line 5 contains all the tables in the database. However, as we can see later, the feasible values for `tableName` are only 'Software', 'Database' and 'Network'. We remove infeasible values from T_A by following a filtering procedure. The filtering algorithm is given in Figure 7-13.

After identifying the set T_A of all the possible values for the `tableName`, we construct the set I of all the possible interaction types as follow:

$$I = \{(\text{operation}, \text{table}) \mid \text{operation} = \text{operationType.concrete}, \text{table} \in T_A\}.$$

Filtering procedure for T_A

1. Construct the set $\text{refs}(\text{tableName})$ of variables referenced by $\text{tableName.symbolic}$.
2. Identify the set C of constraints imposed on variables in the set $\text{refs}(\text{tableName})$ from the set of constraints generated during symbolic execution as follow. For each variable v_k in $\text{refs}(\text{tableName})$, we collect all the related constraints from all nodes in the i-path and put them in a set $C(v_k)$. There is also another type of constraints which are derived by analyzing the HTML form corresponding to variable v_k .

Example 7-10. To illustrate this second type of constraints, let us revisit the `register_course.jsp` page in Figure 7-1. Consider the form displayed by the print statement in line 13. In this case, if the variable `course` is in $\text{refs}(\text{tableName})$ then one constraint can be derived from the form is: $\text{course} \in \{\text{"Software"}, \text{"Database"}, \text{"Network"}\}$.

We use a simplified HTML parser, which can analyze HTML forms to derive constraints (if any) on variable v_k . Note that only input elements such as buttons, checkboxes, radio buttons, lists and menus are constrained by the values indicated in the HTML forms. These constraints are added to set $C(v_k)$. After this process has been repeated for all variables in $\text{refs}(\text{tableName})$, C then can be computed as:

$$C = \bigcup_{v_k \in \text{refs}(\text{tableName})} C(v_k)$$

3. Filter T_A using constraints in C :
 - If C is empty, no further filtering is needed.
 - Otherwise, for each constraint of the form " $v_k \in V_k$ ", in which V_k denotes a set of concrete values and v_k is in the set $\text{refs}(\text{tableName})$, remove from T_A operation types whose concrete values are evaluated using v_k and $v_k \notin V_k$. Conversely, for each constraint of the form " $v_k \notin V_k$ " and v_k is in the set $\text{refs}(\text{tableName})$, remove from T_A operation types whose concrete values are evaluated using v_k and $v_k \in V_k$.

Figure 7-13. Filtering procedure for T_A

Example 7-11. As an illustration for the filtering process for T_A , we infer all the possible database interactions from the statement in line 5 of the code segment in Figure 7-1. In the first approximation, the set T_A contains all the tables in the database. Definitely, this is not a tight approximation. Next, we apply the filtering procedure to refine T_A .

1. Compute $\text{refs}(\text{tableName})$: $\text{refs}(\text{tableName}) = \{\text{course}\}$.

2. Identify set C with respect to $\text{refs}(\text{tableName})$: C contains two constraints, one is generated in Stage 1 (Example 7-3), which is $\text{course} \notin \{\text{" "}\}$, and the other one is derived from the respective HTML form (line 13), which is $\text{course} \in \{\text{"Software"}, \text{"Database"}, \text{"Network"}\}$
3. Filter T_A using constraints in C , we obtain:

$$T_A = \{\text{"Software"}, \text{"Database"}, \text{"Network"}\}.$$
4. Construct set I of all possible database interactions, we have:

$$I = \{(\text{Insert}, \text{"Software"}), (\text{Insert}, \text{"Database"}), (\text{Insert}, \text{"Network"})\}.$$

7.5.2.3 Inference rule 3 (Empirical)

Inference rule 3 is applied when only the `operationType` parameter is input-dependent, that means:

`operationType.concrete == NULL` and `operationType.symbolic != NULL`
`tableName.concrete != NULL` and `tableName.symbolic == NULL`.

When only `operationType` is input-dependent, we apply the same procedure as presented in Inference rule 2 to infer all the possible values for `operationType`. Let $\text{refs}(\text{operationType})$ is the set of variables referenced by `operationType.symbolic`. For the first approximation, we can assume that `operationType` can be any operation which is permitted by the database table indicated by `tableName.concrete`. We shall refer to the set of all possible values for `operationType` in the first approximation as O_A . After the first approximation, we apply the filtering procedure (Figure 7-13), which is described in Inference rule 2, with respect to $\text{refs}(\text{operationType})$ to refine set O_A . After that, we construct the set I of all possible interaction types as follow:

$$I = \{(\text{operation}, \text{table}) \mid \text{operation} \in O_A, \text{table} = \text{tableName.concrete}\}.$$

7.5.2.4 Inference rule 4 (Empirical)

Inference rule 4 is applied when both `operationType` and `tableName` are input-dependent, which means:

`operationType.concrete == NULL` and `operationType.symbolic != NULL`
`tableName.concrete == NULL` and `tableName.symbolic != NULL`.

Inference rule 4 first applies Inference rule 2 and Inference rule 3 to compute two sets T_A and O_A which contain possible values for the `tableName` and `operationType` respectively. After that, the two sets are combined to infer all the possible interactions as follow:

$$I = \{(\text{operation}, \text{table}) \mid \text{operation} \in O_A, \text{table} \in T_A\}$$

7.6 Experiments and evaluation

In this section, we first describe the prototype system that we have implemented following the approach proposed in the previous section. We then present the design and results of two experiments which validate Inference Rule 2, 3 and 4 and evaluate the accuracy of the proposed approach. Finally, we discuss the time complexity of the proposed approach.

7.6.1 The prototype system

We have implemented a prototype system for web applications written in Java to demonstrate the feasibility of the proposed approach. Again, we emphasize that the approach is language-independent and can be applied to any web applications.

Figure 7-14. Prototype system

The implementation is based on EmAnalyzer. Figure 7-14 is the architecture diagram of the prototype system, which consists of two main modules: EmAnalyzer and the Database Interaction Extractor (DaBIE).

The Soot framework and the PathGen components of EmAnalyzer have been extended to Soot^{**} and PathGen^{**}. Basically, Soot^{**} contains an extra HTML form analyzer, a SQL analyzer and an IFG Constructor. The IFG Constructor implements algorithm createIFG presented in Figure 7-4. The HTML form analyzer is a simplified HTML parser [95] which analyzes HTML forms and generates constraints for input variables. SQL analyzer analyzes query strings and extracts symbolic expressions for operationType and tableName. PathGen^{**} includes also an i-path constructor. The i-path constructor is implemented following Step 1 of Stage 1 in the approach to automated database interactions extraction proposed in Section 7.5.

DaBIE is composed of two main components: the Symbolic Executor (**SyE**), the Inference Engine (**IE**); and two rule bases: Symbolic rules and Inference rules, which are used by SyE and IE respectively. SyE implements the second step in Stage 1 and IE implements the Stage 2 of the proposed approach presented in the previous section. It can be seen from the architecture diagram that the framework is extendable in the sense that more symbolic evaluation rules and inference rules can be incorporated into the tool to extract other functional features from source code.

7.6.2 Experiments

We conduct two experiments (1) to validate the empirical inference rules proposed in Section 7.5.2 and (2) to compare the performance of the proposed approach with a conservative approach which assumes all the database interactions.

7.6.2.1 Experiment 1

We have conducted binomial test to validate each empirical inference rule proposed in Section 7.5.2. For each inference rule $\mathcal{R}$, the alternate hypothesis states that $\mathcal{R}$ holds for equal or more than 95% of all the cases. And therefore, the null hypothesis states that $\mathcal{R}$ holds for less than 95% of all the cases; that is:

$$H_0(\text{null hypothesis}) : p(\mathcal{R} \text{ holds}) < 0.95$$

$$H_1(\text{alternate hypothesis}) : p(\mathcal{R} \text{ holds}) \geq 0.95$$

Taking 0.005 as the type I error, if $z > 2.408$, we reject the null hypothesis; otherwise, we accept the hypothesis.

The sample for the validation of the empirical inference rule was collected from seven database applications written in Java including open source systems, student-developed systems and industrial systems. Table 7-1 gives the sample size for testing each rule. JMatchMaker and One-Square are developed by final year students. JMatchMaker is an online match-making system among students.

One-Square is an online real estate management. Service Portal and MOE are industrial systems provided by our industrial partners. For confidential reason, we do not disclose project details here. The rest are open source applications, which can be downloaded from Sourceforge [195] by searching for the project's name. TinaPOS is a point-of-sale system. GFP is a personal finance management and Facecart is an online shopping system.

The validation and their results are summarized as follows:

- **Inference rule 2:** Each i-node whose tableName is input dependent forms a case (an individual) for testing Inference rule 2. For each i-node in the sample, the prototype system computes the possible database interactions by applying Inference rule 2. For each possible interaction inferred by the prototype system, we design a test case with the help of EmValidator to force the execution of the i-node and lead to the database interaction. If such a test case can be generated, we execute the test case and then check the database to confirm on the interaction. If it is confirmative, we conclude that Inference rule 2 holds for the case. As shown in Table 7-1, the total number of cases for testing Inference rule 2 is 535. As there are three cases which violate the rule, the z-score is 4.71 ($X = 532, n = 535, p = 0.95$).
- **Inference rule 3:** Each i-node whose operationType is input dependent forms a case for testing Inference rule 3. For each i-node in the sample, the validation process for Inference rule 3 is similar to that for Inference rule 2. As shown in Table 7-1, the sample size for testing Inference rule 3 is 540. As there are five cases which violate the rule, the z-core is 4.34 ($X = 535, n = 540, p = 0.95$).
- **Inference rule 4:** Each i-node whose both tableName and operationType are input dependent, forms a case for testing Inference rule 4. For each i-node in the sample, the validation process for Inference rule 4 is similar to that for Inference rule 2. As shown in Table 7-1, the total number of cases for testing Inference rule 4 is 672. As there is only one case which violate the rule, the z-score is 5.77 ($X = 671, n = 672, p = 0.95$).

Clearly, the z-scores for testing all the above-mentioned inference rules are greater than 2.408. Thus, we reject all the null hypotheses and conclude that all the inference rules hold for more than or equal to 95 percent of all the cases at 0.5 percent level of significance.

Table 7-1. Sample for testing Inference rules 2, 3 and 4

System	KLOC	Source	IR ₂	IR ₃	IR ₄
JMatchMaker	12	Student	46	25	34
One-square	31		52	39	49
Service Portal	296	Industrial	112	85	74
MOE	178	Industrial	78	42	63
TinaPOS	49	Open source	87	14	67
GFP	60		35	24	31
Facecart	68		311	125	354
Total			535	540	672

7.6.2.2 Experiment 2

In this experiment, we apply both the proposed approach and a conservative approach to extract database interactions from the following ten Java web applications. According to the Tiobe programming index [1] in January 2009, Java is the most popular programming language. As such, there are two main benefits of choosing Java as the main programming language for our experiments. Firstly, we can collect a richer set of sample Java programs from various application domains. Secondly, we have found many program analysis tools for Java programs. These tools provide strong supports which further facilitate our experiments.

Table 7-2. Sample Java systems

System	KLOC	Source	Domain
JMatchMaker	12	Student	Match making
One-square	31		Real estate management
Service Portal	296	Industrial	E-commerce
MOE	178	Industrial	Education
ASM	7	Open source	Animal Shelter Management
TinaPOS	49		Point of sale
Mytracka	12.5		Tracking system
OL	31.8		Logistic
Jbilling	103.8		Enterprise billing system
JAMWiki	32.3		Wiki

The sample programs for our experiments are real programs which have been carefully chosen to minimize all the threats to validity and to maximize the coverage of the experiment. More specifically, our sample programs are taken from three main sources including (1) academic/research institute, (2) industrial project and (3) open-source. This ensures that the sample programs are written by programmers of different expertise levels. Software taken from the first source is normally developed by undergraduate students, master/PhD students and researchers. Software taken from the industrial projects is generally

developed by junior and senior software engineers of two to three years of experience. For open-source software, it is however impossible to access the expertise level of the developers as this information is not available. In this case, we ensure that no two sample programs are taken from the same group of developers. Furthermore, all the sample programs are selected from different application domains with reasonable size.

Table 7-2 gives an overview on the sample programs. All the open-source systems can be downloaded from Sourceforge [195]. Below are the detailed descriptions of each system:

1. *JmatchMaker* is a match-making system designed to help students in a university to find their ideal partners. The system provides services such as dating, private messaging, mutual matching and search agent. The system is developed by third year computer engineering students
2. *One-square* is a student software project under development. The objective of the project is to develop an online real estate management system which maintains a collection of rooms for rental so that potential tenants may view and select suitable rooms from the system. All property agents can add new rooms to the system, and modify and remove their existing rooms maintained in the system.
3. *SevicePortal* is a system contributed by our industrial partner. For confidential reason, we do not provide the project details here. The application domain is E-commerce. ServivePortal is developed by a team of both junior and senior programmers.
4. *MOE* is a system contributed by our industrial partner. For confidential reason, we do not provide the project details here. The application domain is Education. ServivePortal is developed by a team of both junior and senior programmers.
5. *Animal Shelter Manager* is a web-based solution for managing animal sanctuaries and shelters. The application includes features for animal management, document generation, full reporting, charts, internet publishing, pet search engine integration and more.
6. *TinaPOS* is a web based point of sale system designed to help small business keeping track of customers, supplier, product items and sales. The application also generates reports on sales for each customer.
7. *MyTracka* is a web-based issues checking tool. The application includes Feature Tracker, Bug Tracker and Chore Tracker for software development. It also provides functions for projects, users,

tasks and attachment management. The application comes with flexible task workflows and view.

8. *Overactive Logistics* is a freight forwarding software solution being developed for the cargo transportation industry. The application can be run both as a web-based or stand alone tool. In both modes, the system provides a graphical user interface with rich client capabilities.
9. *Jbilling* is a web-based enterprise billing system for business of all sizes. The system manages subscribers with automatic invoicing through email and PDF and payment processing including credit cards, checks and direct deposit.
10. *JAMWiki* is a wiki engine implemented using Java/JSP that attempts to provide much of the functionality of a media wiki. The application is designed to be fact and easy to set up. It can be run with or without a database.

Besides Java programs, we also collect sample from open-source programs written in PHP. The same experiment is conducted on these PHP programs to show that (1) the proposed approach can be applied to other programming language and (2) the results is consistent with the experiment results obtained from Java programs. We have selected seven open-source PHP systems which are all downloaded from Sourceforge [195]. Table 7-3 gives an overview on the sample PHP systems. Below are detailed descriptions of each system.

1. *Nalanda* is a library management system. This system provides common management features like issuing and returning books, adding users, etc. In addition to these features, Nalanda also generates statistics like total books issued, total fined collected, for a particular day or over a period.
2. *POS* (Php Point of sale) is a web based point of sale system designed to help small businesses keeping track of customers, supplier, product items, and sales. The application also generates reports on sales for each customer.
3. *Mantis* is a web based bug tracking system which allows individual or groups of developers and programmers to keep track of outstanding bugs in their product effectively. Mantis provides features to track bugs and code changes, to communicate between team-mates and to manage quality of the product.
4. *PhpESP* (Php Easy Survey Package) is a survey management system. Non-technical users can easily create surveys and then put them online. They can also track the survey, collect results and then use the system to generate different statistics.

5. *PhpPlanner* is a basic event tracker/calendar and scheduler with many functions to customize the general behavior of the calendar.
6. *Moodle* is a course management system (CMS) which is designed to help educators create effective online learning communities. Moodle provides four groups of features for administrators, teachers, students and developers respectively.
7. *WebCollab* is a collaborative web-based system for projects and project management; WebCollab is easy to use, and encourages users to work together. The software is ideally suitable for tracking multiple projects and innumerable small tasks across an organization of any size.

To facilitate this experiment, a PHP parser [166] has been integrated into the EmAnalyzer component so that it can analyze also PHP applications. The PHP parser constructs a CFG for each PHP program. Each node in the CFG is also attached with control and data dependency information.

Table 7-3. Sample PHP systems

System	KLOC	Domain
Nalanda	5	Library management
POS	87	Point of sale
Mantis	147	Bug tracking
PhpESP	77	Survey management
PhpPlanner	2	Event tracker
Moodle	587	Course management
WebCollab	37	Project collaboration

For the proposed approach, we run the prototype system on one application at a time to extract all possible interactions of the application with its database. In this experiment, the symbolic execution of each function call statement was monitored. If the number of execution paths started to grow exponentially, the symbolic execution of the function call statement is terminated. In this case, we apply a conservative approach, which assumes all the possible values for symbolic variables.

We also implement a conservative approach. For each i-path in each program in the web application, we skip the symbolic execution stage. Therefore, for each symbolic variable used by each i-node, there is no constraint on the possible values of the symbolic variable. As a consequence, in the inference stage, the conservative approach assumes all the values for each symbolic variable in its domain. For example, for the symbolic expression denoting operationType, all

the operations including select, insert, update, delete are possible; for the symbolic expression denoting tableName all the tables in the database are possible values.

For each i-path in each program, we use TestGen to generate a symbolic expression representing the i-path. We then analyze and solve the symbolic expression manually to identify all the possible types of database interactions when executing the i-path. For each possible type of database interactions identified, we double check the correctness result through test case generation and execution. We then design a test case with the assistance of EmValidator and populate database content based on the corresponding i-path to force the execution of the i-node. We also use the code instrumentation tool in JTracer to insert a statement immediately following each i-node to print a traveling marker, which serves as an indication of the successful execution of the i-node. We run the test case and check the database to confirm that the interaction has been executed and the database is updated. If a type of database interaction is confirmed through both symbolic solving and test case execution, we include the type in the set Π . The set Π is referred to as the **set of expected database interactions**.

For each possible database interaction extracted from the source code by both the proposed approach and the conservative approach, we verify the correctness of the information by comparing them with the database interactions in the set Π . If the extracted interaction type is not in the set Π , we conclude that it is a **false positive** case; It is called a **true positive** case. If a case is in the set of expected database interactions but it is undiscovered by an approach (proposed approach or conservative approach), it is classified as a **true negative** case with regard to the approach.

Let I_t , I_{fp} and I_{tn} be the total number of database interactions extracted from the web application, the number of false positive cases and the number of true negative cases respectively. Adopting the definition of precision from Information Retrieval field, the precision of the proposed approach with respect to a web application is measured by the percentage of extracted database interactions that are also true positive, which is computed as follow:

$$Precision = \frac{I_t - I_{fp}}{I_t} \times 100\%$$

The recall (completeness) of the proposed approach with respect to a web application measures the ratio between the number of extracted database interaction which are true positive and the actual number of all the possible database interactions of the web application.

$$\text{Recall} = \frac{I_t - I_{fp}}{I_t - I_{fp} + I_m} \times 100\%$$

Since we are conservative in some cases where we cannot evaluate the exact values of a symbolic variable, the number of true negative cases is always zero; thus the proposed approach always results in 100% recall. We then evaluated the performance of the proposed approach based on the traditional f -measure as follow:

$$f_measure = 2 \times \frac{precision \times recall}{precision + recall} = 2 \times \frac{precision}{precision + 1}$$

F -measure weights low values of precision and recall more heavily than higher values. It is high if both precision and recall are high.

The results of experiment 2 for both Java and PHP sample programs are shown in Table 7-4. The first four columns present the results of applying the proposed approach for each web application. Columns I_t , I_{fp} , Pre and f give the total number of database interactions extracted, the number of positive cases, the precision of the proposed approach and the f -measure, respectively. The last four columns present the results of applying the conservative approach for each web application. The results are also presented in term of I_t , I_{fp} , Pre and f . The “Average” row gives the average precision and f -measure of the proposed approach and the conservative approach.

The results show that on average, the proposed approach gives a precision of 80.6% for Java systems and 79.2% for PHP systems. The highest achievable precision score and lowest precision score for Java systems is 95.3% and 69.5% respectively. The highest achievable precision score and the lowest precision score for PHP systems is 100% and 60.9% respectively. We inspect all the false positive cases and found that they are false for two main reasons.

- First, symbolic executions of function call statements terminate early due to the exponential number of execution paths in the body of the called function. This early termination leads to the conservative inference of possible values for symbolic variables.
- Second, in the presence of polymorphism and pointers, no symbolic evaluation rules are available; thus in the inference stage, all possible values in the domain are assigned for each symbolic variable involving in polymorphism calls and pointers.

Table 7-4 also shows the improvement of the proposed approach in comparison with that of the conservative approach implemented.

Table 7-4. Results of experiment 2

System	Proposed Approach				Conservative approach			
	I_t	I_{fp}	Pre(%)	f	I_t	I_{fp}	Pre(%)	f
Java systems								
JMatchMaker	11	3	72.7	0.84	18	10	44.4	0.62
One-square	24	5	79.2	0.88	34	15	55.9	0.72
ServicePortal	107	14	86.9	0.93	147	54	63.3	0.78
MOE	318	68	78.6	0.88	466	216	53.6	0.70
ASM	32	7	78.1	0.88	61	36	41.0	0.58
TinaPOS	124	19	84.7	0.92	292	187	36.0	0.53
Pentaho	315	96	69.5	0.82	501	282	43.7	0.61
MyTracka	64	3	95.3	0.98	176	115	34.7	0.51
OaL	76	14	81.6	0.90	227	165	27.3	0.43
JBilling	1012	197	80.5	0.89	1981	1166	41.1	0.58
JAMWiki	511	106	79.3	0.88	1567	1162	25.8	0.41
Average			80.6	0.89			42.4	0.59
PHP systems								
Nalanda	23	2	91.3	0.95	68	47	30.9	0.47
POS	87	34	60.9	0.76	211	158	25.1	0.40
Mantis	331	115	65.3	0.79	431	215	50.1	0.67
PhpESP	7	0	100	1.00	25	18	28.0	0.44
PhpPlanner	53	14	73.6	0.85	119	80	32.8	0.49
Moodle	1023	233	77.2	0.87	2178	1388	36.3	0.53
WebCollab	789	112	85.8	0.92	1673	996	40.5	0.58
Average			79.2	0.88			34.8	0.51

Legend: ASM – Animal Shelter Manager
 POS – Point Of Sale
 OaL – Overactive Logistic

For Java systems, on average, the f -score of the proposed approach is greater than that of the conservative approach by 0.3. In term of precision, the proposed approach gives an average precision of 80.6% while the conservative approach gives 42.4% on average. This means with the proposed approach, we are able to eliminate 38.2% false positive cases.

For PHP systems, on average, the f -score of the proposed approach is greater than that of the conservative approach by 0.37. In term of precision, the proposed approach gives an average precision of 79.2% while the conservative approach gives 34.8%. This means that the proposed approach is able to eliminate 44.4% false positive cases.

Consider both languages, on average, the proposed approach has given a precision of 41.3% more than that of the conservative approach. From these results, we can conclude that the experiment results obtained with PHP programs are consistent with the results obtained with Java programs. Both set

of results show that the proposed approach outperforms the conservative approach in term of precision score and f-score.

7.6.3 Time complexity

It is clear that the cost of the proposed approach is dominated by the symbolic execution of i-paths. Symbolic execution becomes complicated with the presence of function calls, especially nested function call. In the worst case, the time for extracting database interactions can be exponential. In this chapter, we have introduced the following heuristics to improve the performance of the proposed approach in the presence of function calls:

- Demand-driven symbolic execution of function calls
- Hybrid mix of symbolic execution and concrete execution
- Preliminary symbolic execution of frequently used function

However, even with these heuristics, the run-time still grows exponentially in some cases where there are many levels of nested function call. As such, in our experiment, we chose to early terminate the symbolic execution in these cases. As shown in the results of our experiments, this approach resulted in some false positive cases. Despite that, the proposed approach has shown its competitiveness over a pure conservative approach in eliminating a large number of false positive cases. The results also suggest further improvements to minimize the number of false positive cases.

7.7 Summary

We have presented in this chapter an approach to the extraction of database interactions from web application source code by using hybrid symbolic execution and inference rules. All the empirical inference rules also empirically validated via extensive experiments on ten web applications. The statistical validation gives evidence that all the inference rules hold for equal or more than 95 percent of all the cases at 0.5 percent level of significance.

We have introduced IFG which facilitates the extraction of database interactions. Our approach consists of two stages. In the first stage, for each procedure in the system, all the i-paths through the IFG are identified. After that, symbolic evaluation rules are applied to symbolically execute each i-path. In the second stage, inference rules are used to infer all the possible interaction types from each symbolic expression derived in the first stage. We have compared the performance of the proposed approach with the conservative approach, which assumes all the possible values for each symbolic variable. Experiments on both Java and PHP systems give evidence that the proposed approach outperforms the conservative approach and on average, the proposed approach give a precision of 41.3% more than that of the conservative approach

Although the proposed approach has shown its improvements over the conservative approach, it still results in a number of false positive cases due to the limitations in handling function calls, pointers and polymorphism. In the future, more symbolic evaluation rules will be incorporated to solve this problem. In addition, we are also conducting some experiments, whose results suggest that information collected during run-time will be useful to speed up the inference process of the proposed approach.

Chapter 8

CONCLUSION AND RECOMMENDATIONS

This chapter summarizes the contributions of this research. It suggests future work that includes enhancing the presented empirical-based solutions in this thesis.

8.1 Conclusion

“Experimentation is central to the scientific process” [200]. This statement is especially true for software engineering because most of the research proposals cannot be proven formally. It has been proved by many researchers that experimentation is vital for software engineering to become a mature scientific discipline.

Program analysis techniques are central to many software engineering tasks such as testing, debugging, maintenance, program comprehension and reverse engineering. However many program analysis techniques are rarely used in practice because of their inscalability to large systems. In this thesis, we have demonstrated the application of traditional program analysis techniques augmented with empirical properties. Through applying this augmented approach, we have established solutions for the following problems in software analysis, testing and maintenance:

1. *Infeasible path detection*: Static program analysis is undoubtedly a fundamental technique in software engineering. However, one of the biggest challenges for static analysis is the existence of infeasible paths in that there is no input for that the paths will be executed. Detection of infeasible paths plays an important role in static program analysis and many program-analysis-based software engineering tasks, especially structural testing and coverage analysis. We have formalized in Chapter 5 four common patterns of infeasible program paths including: conflicting-decision pattern, mutually-exclusive-decision pattern, check-then-do pattern and looping-by-flag pattern. Our statistical validation

results have shown that the four patterns hold for 97% of all the cases in the sample at 0.1% level of significance. The sample was collected from seven systems including two student projects, one industry project and four open source applications.

We have further proposed algorithms for detecting infeasible paths based on the four code patterns. A prototype tool called InfeasibleDetector has also been developed by utilizing program analysis information provided by EmAnalyzer and implementing the proposed algorithms. Experiments have been conducted to evaluate the accuracy of the proposed approach in detecting infeasible program paths. The results show that on average, the proposed approach correctly detects 82.3% of all the infeasible paths, in which Property 5-3 detects the most number of paths with 44.6%, followed by Property 5-1 with 33.2%, Property 5-2 with 21.0% and Property 5-4 with only 1.2%. There are 17.7% of all the infeasible paths which are not detected by the proposed approach. We have carefully analyzed these cases and discovered that 4.2% of all the cases do not fall under any of the existing patterns and 13.5% of all the cases not detected because of the limitation on the current prototype system.

Most the experiments have been conducted on Java programs due to the availability of the program analysis tools including Soot, JDI and Kaveri. However, the set of empirical properties presented in this work are programming language-independent as all the properties are applied on a high level abstract representation of a program, the control flow graph. Nevertheless, in this thesis, we have also provided the implementation of our infeasible path detection technique for programs written in C language. The tool is called ArisPath which is based on the Aristotle [82] program analysis tool. We have also repeated our experiment on C programs. The results show that ArisPath is able to detect about 93.3% of all the infeasible paths.

This work has been published in [151, 153].

2. *Automated recovery, testing and maintenance of input error correction:* Many input errors occurred in the execution of a database application which cannot be detected automatically through input validation or specification integrity. As a result, such errors are only discovered after the completion of the execution of the system. Therefore, the provision of input error correction for handling the erroneous effects resulting from these errors is essential in any database application. Current design recovery methods are hard to apply to recover the provision of input

error correction because of the complex relationship between input errors, effect errors and error correction mechanism.

In this thesis, we have formed a theory in Chapter 6 for inferring the provision of input error correction in database applications. The theory covers the inference of user input errors from errors in the input to a complete path, the derivation of effect errors from an input error and the identification of input error correction. The foundation of the theory is the six properties of input errors, effect errors and input error correction discovered through empirical studies of various database applications. The first property states that each user input error can be represented as an input error in a complete path. The second property reflects the relationship between an input error and its resulting effect errors. The third property describes characteristics of an error correction node for correcting effect errors in an output statement. The fourth property states conditions for a collection of complete paths for correcting an input error. The last two properties specify conditions for which all input errors in a procedure and a system are correctable, respectively. Among these properties, there are three empirical properties which are validated statistically with samples collected from a wide range of database applications. The validation gives evidence that all the empirical properties hold for more than 98% of all the cases at 0.5% level of significance.

Through recognizing properties of input error correction, a novel approach for the automated recovery of this feature from source code has been proposed. The approach recovers input error correction feature in four steps: (1) compute program analysis information, (2) infer input errors and effect errors, (3) compute basis collection of error correction paths for correcting all the input errors and (4) recover the system's correctability. We have shown the usefulness of the recovered information in testing and maintenance of input error correction feature.

We have described a testing strategy called Black&White based on the recovered input error correction in which only a single test suite is required to cover both code and specification behaviors of the feature. A code-based IEC-equivalence partition can be formed from the recovered information such that input values in the same equivalence class are handled uniformly by the implementation with respect to the input error correction. This code-based equivalence partition can be reconciled with a specification-based equivalence partition constructed by using any existing partition-based black-box testing technique. Our Black&White strategy generates test cases based on the reconciled implementation;

thus it combines both code and specification behaviors. Our study has shown that this testing strategy outperformed basis path testing (structural) and equivalence class testing (functional). It detected 100% of the seeded faults and even performed better than the combination of both basis path testing and equivalence class testing.

Maintenance of input error correction is a time-consuming and costly task because it requires the maintainer to have a deep understanding of how the feature has been implemented in the system and to modify the feature according to the requirements. Moreover, effective tools should be provided to assist the maintainer in these activities as nowadays, a database application can be quite complicated which contains a large number of components in hundred thousand lines of code. Based on the recovered information on input error correction, we further propose an approach to aid the maintenance of this feature. The approach is a combination of decomposition slicing and empirical properties of input error correction. We adopt the idea of decomposition slicing and propose the effect-oriented decomposition slicing technique. The difference is that: the decomposition slicing technique computes a slice which contains all the statements that affect computation of a variable; while the effect-oriented decomposition slicing technique computes a slice which contains all the statements that affect the implementation of the input error correction feature for an input error. As such, effect-oriented decomposition slices are useful in understanding the existing implementation of input error correction in a database system and modifying the feature without affecting other parts of the system. An exploratory study has been conducted which shows that without using the proposed recovery approach, even a programmer with two years of experiences spent much time and effort on understanding the feature, yet precise information cannot be fully recovered.

A tool named EmRTM has been developed by extending EmAnalyzer and EmValidator for the automated recovery, testing and maintenance of input error correction.

This work has been published in [152, 150].

3. *Automated data extractions for web applications*: Many programs in a web application interact with databases to query for data, update data or store data. Database interaction is one of the most important functional features that should be provided in any web application that utilize a Database Management System and one or more databases. Information

on database interactions is useful in maintenance, test data generation and coverage analysis of web applications.

We have proposed in Chapter 7 an approach to automatically extract database interactions by statically analyzing web applications. The approach combines symbolic evaluation and inference rules to approximately infer all the possible database interactions from source code. To focus on the database interaction feature, we have introduced the Interaction Flow Graph (IFG) which contains all the nodes and edges from the respective CFG which affect the execution of all the interaction nodes in the CFG. We have defined five symbolic evaluation rules to evaluate the following types of statements in an IFG: binary expression, if-statement, assignment-statement including array access expressions, loop-statement and function-call statement. We have also proposed four inference rules to infer all the possible interactions in a symbolic expression. All the empirical inference rules have been statistically validated which show at 0.5% level of significance that all the inference rules hold for 95% of all the cases in the samples.

The proposed approach involves two stages. In the first stages, all the complete paths through the IFG which contain some interaction nodes are computed. These paths are then symbolically evaluated by applying the symbolic evaluation rules to obtain symbolic expressions representing possible database interactions in the paths. We have proposed using a hybrid form of symbolic evaluation and concrete evaluation to speed up the symbolic execution process. In the second stage, the four inference rules are applied to derive all the database interactions from each symbolic evaluation computed in the first stage.

A prototype tool called DaBIE has been developed based on EmAnalyzer following the proposed approach. We have conducted an experiment to compare the performance of the proposed approach with a conservative approach in term of the precision in extracting database information. The experimental results show that the proposed approach outperforms the conservative approach and on average the proposed approach gives a precision of 41.3% more than that of the conservative approach.

This work has been published in [154, 155].

8.2 Limitation of the study

In this study, a number of experiments have been conducted to evaluate the application of the proposed techniques in any targeted computer program. Therefore, an ideal set of experimental programs should include targeted programs of reasonable size which are (1) from different application domains, (2) written by programmers of different experience levels and (3) from different stages of the software development life cycle. These factors can be fully controlled for experimental programs taken from academic or industrial projects as all the information is available through the project documentation or through interviewing with the project members. However, the last two factors cannot be fully controlled for open source programs. Information on the developers' experience is unavailable for open source programs and the development status might not be synchronized with the project release. These factors could only be allowed to vary among the group on what was assumed to be a random manner. As such, there are chances that our experimental programs are written by developers of the same background or are at the same stage of the development process. If this is the case then our sample might not be representative of the population. We could re-run the experiments in a more controlled manner in the future when resources become available.

8.3 Recommendations for further research

This research has explored the use of empirical properties which are validated statistically to solve (1) the infeasible path detection problem, (2) the automated recovery, testing and maintenance of input error correction problem and (3) the database interaction extraction problem for web applications. All the solutions presented in this thesis have given the evidences to show that this is a promising research area, which might open a new avenue to automate many other software engineering tasks. In this section, we identify potential areas for future work.

1. **Hypothesis testing:** This research, like any empirical-based research, has some limitations. All the binomial tests conducted to validate the empirical properties are influenced by our choice of the sample sets and the subject applications. Even though we have randomly chosen systems from various application domains which are at different stages of the software development life cycle, the chosen system might not be representative of the population. Moreover, we only examine systems written in Java. Even though, all the theories formed are language-independent, more validation should still be conducted on other programming languages. This might also lead to the discovery of language-specific empirical

properties, which might speed up the analysis for certain programming languages.

2. **Language-independent program analysis tools:** All the program analysis tools described in this thesis are based on the EmAnalyzer tool for Java systems. In the future, to build language-independent tools, we need to develop a language-independent control flow graph representation. For each programming language, a parser or parser driver should be developed to parse systems written in that language and represent the system in the form of the language-independent control flow graph. All the algorithms will then be implemented based on the new representation.
3. **Interprocedural infeasible path detection:** The approach presented in this thesis is only for intraprocedural infeasible path detection. To support interprocedural program analysis, we need to propose new patterns to detect interprocedural infeasible paths. In software design, coupling is a measure of interprocedural connectivity. Two procedures have control-coupling if their decisions for carrying out actions are interdependent. In general, two procedures are control-coupled if (1) one passes data to the other that is used in the latter procedure to determine the actions to be performed or (2) they use shared data to determine actions to be performed. This control-coupling suggests that interprocedural infeasible paths are somehow related to this.
4. **Coverage criteria for testing input error correction:** An important key to ensure software quality is through the measurement of test quality. Coverage analysis of programs during testing not only gives a clear measure of testing quality but also reveals important aspects of software structure [92]. To the best of our knowledge, no proposed coverage criteria can be applied to measure the test coverage for input error correction. This is because a test suite must contain a test case to exercise the path which leads to the input error and raises effect errors. The test suite must also contain test cases to execute error correction paths for correcting the input error to confirm that there is no error in implementing the correction features. As such, a test suite for testing input error correction must cover the relationship between an input error and its error correction paths. However, traditional coverage criteria cannot guarantee this. For example, a test suite which achieves 100% coverage of the paths in the basis set might still miss testing some of the error paths. Based on the recovered information on input error correction, new

coverage criteria can be introduced which address the relationship between input errors and correction paths.

5. **Database interactions extraction:** Our approach to database interaction extraction results in a number of false positive cases due to the limitation of the proposed approach in handling function calls, pointers and polymorphism. In the future, more symbolic evaluation rules will be incorporated to solve this problem. On the other hand, we are investigating into using more empirical inference rules to reduce the number of symbolic evaluation rules; thus reduce the computational complexity of the proposed approach. In addition, we are also conducting some experiments, whose results suggest that information collected during program execution will also be useful to speed up the inference process of the proposed approach. This suggests the integration of empirical properties with dynamic information.

APPENDIX A

Table A-1. General program analysis notations

Variable Name	Meaning
P	A program
P_T, P_S	Arbitrary procedures within a program
$\mathcal{G}$	A control flow graph
$\mathcal{N}$	A set of control flow graph nodes
$\mathcal{E}$	A set of control flow graph edges
n_p, n_q	Arbitrary control flow graph nodes
e_x, e_y	Arbitrary control flow graph edges
n_p^{succ}	A successor of node n_p
$n_p^{\text{true}} / n_p^{\text{false}}$	The successor of predicate node n_p which is in the true/false branch of n_p
v_k	An arbitrary variable in a program
V_k	A concrete value
p_x	An arbitrary path in a control flow graph
c_x	A complete path in a control flow graph
r_x	A representative path in a control flow graph
s_k	A predicate of a predicate node or a variable or an expression in general

Table A-2. Slicing notations

Variable Name	Meaning
$S(v_s, n_s)$	The slice of a program on variable v_s at statement n_s
$DS(v_s)$	The decomposition slice of a program taken with respect to variable v_s
$ES(O)$	The effect-oriented slice of a program taken with respect to the set of output statements O

Table A-3. Additional notations used by Chapter 6

Variable Name	Meaning
EC	Error of commission
EO	Error of omission
VE	Value error
b_k	An input block
$\{r_{q_1}, r_{q_2}, \dots, r_{q_k}\}$	A set of representative paths in a control flow graph
ξ	A type of input error
B_ξ	A basic collection of error correction path for correcting the type ξ of input error
O_p	The set of output statements in the complete path c_p /or r-path r_p
O_p^i	A subset of the set of output statements in the complete path the complete path c_p /or r-path r_p
C_k	A code-based IEC-equivalence class
S_k	A specification-based IEC-equivalence class

Table A-4. Additional notations used by Chapter 7

Variable Name	Meaning
s_k	An expression or a variable in general
$s_k.symbolic$	The symbolic field of the variable/expression s_k . If $s_k.symbolic$ is not equal to NULL, s_k is a symbolic expression
$s_k.concrete$	The concrete field of the variable/expression s_k . If $s_k.concrete$ is not equal to NULL, s_k can be evaluated to a concrete value
I	A set of database interactions extracted from source code
tableName	A database table name
operationType	A type of SQL DML operation including insert, update, delete
i_x	An interaction path in a control flow graph

APPENDIX B

```
public static void main(String[] args) {  
    // 1 - Create the problem  
    AbstractProblem pb = new Problem();  
  
    // 2 - Create the variables x and y  
    IntVar x = pb.makeBoundIntVar("x",  
        Integer.MIN_VALUE, Integer.MAX_VALUE);  
    IntVar y = pb.makeBoundIntVar("y",  
        Integer.MIN_VALUE, Integer.MAX_VALUE);  
  
    // 3 - Create and post the constraints  
    pb.post(pb.lt(x, 5));           // x < 5  
    pb.post(pb.gt(y, 3));          // y > 3  
    pb.post(pb.lt(x, y));          // x < y  
  
    // 4 - Search for all solutions  
    pb.solveAll();  
  
    // 5 - Print the found solutions  
    System.out.println(pb.solutionToString());  
}
```

Figure B-1. An example program using Choco

AUTHOR'S PUBLICATIONS

- [1] **M. N. Ngo** and H. B. K. Tan, "Applying static analysis for automated extraction of database interactions in web applications," *Information and Software Technology*, vol. 50, pp. 160-175, 2008.
- [2] **M. N. Ngo** and H. B. K. Tan, "Program-mining augmented with empirical properties," in *Encyclopedia of Data Warehousing and Mining (Second Edition)*: IGI Global, March 2008.
- [3] **M. N. Ngo** and H. B. K. Tan, "Heuristics-based infeasible path detection for dynamic test data generation," *Information and Software Technology*, vol. Volume 50, pp. 641-655, June 2008.
- [4] **M. N. Ngo** and H. B. K. Tan, "Detecting large number of infeasible paths through recognizing their patterns," presented at Proceedings of the 6th joint meeting of the *European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, Dubrovnik, Croatia, 2007(top conference).
- [5] **M. N. Ngo** and H. B. K. Tan, "Empirical-based recovery and maintenance of input error-correction features," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, pp. 419-450, 2007.
- [6] **M. N. Ngo** and H. B. K. Tan "A method to aid recovery and maintenance of input error correction features", *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, September 2006, Philadelphia, Pennsylvania.
- [7] **M. N. Ngo**, H. B. K. Tan and D. Trinh, "Automated Extraction of Database Interactions in Web Application", *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC '06)*, June 2006, Athens, Greece.

BIBLIOGRAPHY

- [1] "Tiobe Programming Index,"
<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>,
2009.
- [2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques and Tools*: Addison-Wesley, 1986.
- [3] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An overview of the PTRAN analysis system for multiprocessing," *Journal of Parallel and Distributed Computing*, vol. 5, pp. 617-40, 1988.
- [4] F. E. Allen, "Control flow analysis " *SIGPLAN Not.* , vol. 5 pp. 1-19 1970
- [5] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Communications of the ACM*, vol. 19, pp. 137-46, 1976.
- [6] G. Antoniol, M. Di Penta, and M. Zazzara, "Understanding Web applications through dynamic analysis," presented at Proceedings. 12th IEEE International Workshop on Program Comprehension, 24-26 June 2004, Bari, Italy, 2004.
- [7] G. Antoniol and Y.-G. Gueheneuc, "Feature identification: an epidemiological metaphor," *IEEE Transactions on Software Engineering*, vol. 32, pp. 627-41, 2006.
- [8] Apache-Jakarta, "Tomcat," <http://tomcat.apache.org/>, 2007.
- [9] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," presented at 27th International Conference on Software Engineering, 15-21 May 2005, St. Louis, MO, USA, 2005.
- [10] J. P. Banning, "An efficient way to find the side effects of procedure calls and the aliases of variables " in *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* San Antonio, Texas ACM, 1979 pp. 29-41
- [11] J. M. Barth, "A practical interprocedural data flow analysis algorithm," *Communications of the ACM*, vol. 21, pp. 724-36.
- [12] V. Basili, F. Shull, and F. Lanubile, "Using experiments to build a body of knowledge," presented at Perspectives of Systems Informatics. Third International Andrei Ershov Memorial Conference. PSI'99. Proceedings, 6-9 July 1999, Akademgorodok, Russia, 2000.
- [13] V. R. Basili, "The role of experimentation in software engineering: past, current, and future," presented at Proceedings of IEEE 18th

- International Conference on Software Engineering, 25-30 March 1996, Berlin, Germany, 1996.
- [14] J. Beck and D. Eichmann, "Program and interface slicing for reverse engineering," presented at Proceedings of the 15th International Conference on Software Engineering, May 17-21 1993, Baltimore, MD, USA, 1993.
 - [15] W. G. Bently and E. F. Miller, "Ct coverage-initial results," *Software Quality Journal*, vol. 2, pp. 29-47, 1993.
 - [16] T. Berlage, "A selective undo mechanism for graphical user interfaces based on command objects " *ACM Trans. Comput.-Hum. Interact.* , vol. 1 pp. 269-294 1994
 - [17] S. Beydeda, V. Gruh, and M. Stachorski, "A graphical class representation for integrated black- and white-box testing," presented at Proceedings IEEE International Conference on Software Maintenance. ICSM 2001, 7-9 Nov. 2001, Florence, Italy, 2001.
 - [18] G. Bilardi and K. Pingali, "Framework for generalized control dependence," presented at Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation, PLDI, May 21-24 1996, Philadelphia, PA, USA, 1996.
 - [19] D. Binkley, "Precise executable interprocedural slices," *ACM Letters on Programming Languages and Systems*, vol. 2, pp. 31-45, 1993.
 - [20] D. Binkley, "Source Code Analysis: A Road Map " in *2007 Future of Software Engineering* IEEE Computer Society, 2007 pp. 104-119
 - [21] D. Binkley and M. Harman, "Results from a large-scale study of performance optimization techniques for source code analyses based on graph reachability algorithms," presented at Third IEEE International Workshop on Source Code Analysis and Manipulation. Proceedings, 26-27 Sept. 2003, Amsterdam, Netherlands, 2003.
 - [22] R. Bodik, R. Gupta, and M. L. Soffa, "Interprocedural conditional branch elimination," presented at Proceedings of the 1997 ACM SIGPLAM Conference on Programming Language Design and Implementation, PLDI, Jun 15-18 1997, Las Vegas, NV, USA, 1997.
 - [23] R. Bodik, R. Gupta, and M. L. Soffa, "Refining data flow information using infeasible paths," *ESEC/FSE '97. 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 22-25 Sept. 1997, vol. 22, pp. 361-77, 1997.
 - [24] M. D. Bond and K. S. McKinley, "Practical path profiling for dynamic optimizers," presented at International Symposium on Code Generation and Optimization, CGO 2005, Mar 20-23 2005, San Jose, CA, United States, 2005.

- [25] J. P. Bowen, J. A. Clark, R. M. Hierons, K. Bogdanov, M. Harman, and P. Krause, "FORTEST: Formal methods and testing," presented at 26th Annual International Computer Software and Applications Conference, Aug 26-29 2002, Oxford, United Kingdom, 2002.
- [26] A. B. Brown, "Coping with human error in IT systems," *ACM Queue*, vol. 2, pp. 34-41, 2004.
- [27] P. M. S. Bueno and M. Jino, "Identification of potentially infeasible program paths by monitoring the search for test data," presented at Proceedings of ASE 2000 15th IEEE International Automated Software Engineering Conference, 11-15 Sept. 2000, Grenoble, France, 2000.
- [28] P. M. S. Bueno and M. Jino, "Automatic test data generation for program paths using genetic algorithms," *International Journal of Software Engineering and Knowledge Engineering*, vol. 12, pp. 691-709, 2002.
- [29] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software - Practice and Experience*, vol. 30, pp. 775-802, 2000.
- [30] C. Cadar and D. Engler, "Execution Generated Test Cases: How to Make Systems Code Crash Itself," *ACM Symposium on Operating System Principle*, 2005.
- [31] T. E. Cheatham, Jr., G. H. Holloway, and J. A. Townley, "Symbolic evaluation and the analysis of programs," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 402-17, 1979.
- [32] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen, "In black and white: an integrated approach to class-level testing of object-oriented programs," *ACM Transactions on Software Engineering and Methodology*, vol. 7, pp. 250-95, 1998.
- [33] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," presented at Proceedings IWPC 2000. 8th International Workshop on Program Comprehension, 10-11 June 2000, Limerick, Ireland, 2000.
- [34] T. Y. Chen, P.-L. Poon, S.-F. Tang, and T. H. Tse, "On the identification of categories and choices for specification-based test case generation," *Information and Software Technology*, vol. 46, pp. 887-898, 2004.
- [35] E. J. Chikofsky and J. H. Cross, II, "Reverse engineering and design recovery: a taxonomy," *IEEE Software*, vol. 7, pp. 13-17, 1990.
- [36] Choco, "Control Solving Problem (CSP) solver," http://choco-solver.net/index.php?title=Main_Page, 2007.
- [37] J.-D. Choi, M. Burke, and P. Carini, "Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects," presented at 20th Annual ACM SIGPLAN-SIGACT Symposium on

- Principles of Programming Languages, Jan 10-13 1993, Charleston, SC, USA, 1993.
- [38] S. Chung and Y.-S. Lee, "Reverse software engineering with UML for Web site maintenance," presented at Proceedings of WISE 2000: 1st International Conference on Web Information Systems Engineering, 19-21 June 2000, Hong Kong, China, 2000.
 - [39] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. SE-2, pp. 215-22, 1976.
 - [40] P. J. Clarke, J. Ding, D. Babich, and B. A. Malloy, "A tool to automatically map implementation-based testing techniques to classes," *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, pp. 585-614, 2006.
 - [41] M. B. Cohen, M. B. Dwyer, and S. Jiangfan, "Coverage and adequacy in software product line testing," presented at ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis, ROSATEA '06, Jul 17-20 2006, Portland, ME, United States, 2006.
 - [42] T. D. Cook and D. T. Campbell, *Quasi-Experimentation: Design & Analysis Issues for Field Settings*: Houghton Mifflin Company, 1979.
 - [43] K. D. Cooper and K. Kennedy, "Interprocedural side-effect analysis in linear time," *SIGPLAN Notices*, vol. 39, pp. 217-18, 2004.
 - [44] S. Cornett, "Code Coverage Analysis," *Bullseye Testing Technology*, 2008.
 - [45] Crimson, "A java XML parser," <http://xml.apache.org/crimson/>.
 - [46] C. Csallner and Y. Smaragdakis, "Check 'n' crash: Combining static checking and testing," presented at 27th International Conference on Software Engineering, ICSE 2005, May 15-21 2005, Saint Louis, MO, United States, 2005.
 - [47] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451-90, 1991.
 - [48] R. Cytron, J. Ferrante, and V. Sarkar, "Compact representations for control dependence," presented at ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, 20-22 June, NY, USA, 1990.
 - [49] Y. Deng, P. Frankl, and J. Wang, "Testing Web Database Applications," *SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 1-10, 2004.
 - [50] P. Devanbu, "On "A framework for source code search using program patterns"," *IEEE Transactions on Software Engineering*, vol. 21, pp. 1009-10, 1995.

- [51] G. A. Di Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini, "Comprehending Web applications by a clustering based approach," presented at Proceedings 10th International Workshop on Program Comprehension, 27-29 June 2002, Paris, France, 2002.
- [52] G. A. Di Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini, "WARE: a tool for the reverse engineering of Web applications," presented at Proceedings of the Sixth European Conference on Software Maintenance and Reengineering, 11-13 March 2002, Budapest, Hungary, 2002.
- [53] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana, "Towards a better comprehensibility of web applications: lessons learned from reverse engineering experiments," presented at Proceedings Fourth International Workshop on Web Site Evolution. WSE 2002, 2 Oct. 2002, Montreal, Que., Canada, 2002.
- [54] G. A. Di Lucca, A. R. Fasolino, P. Tramontana, and U. De Carlini, "Abstracting business level UML diagrams from Web applications," presented at Proceedings Fifth IEEE International Workshop on Web Site Evolution Theme: Architecture, 22 Sept. 2003, Amsterdam, Netherlands, 2003.
- [55] G. A. Di Lucca, A. R. Fasolino, P. Tramontana, and U. De Carlini, "Recovering a business object model from a Web applications," presented at Proceedings 27th Annual International Computer Software and Applications Conference, 3-6 Nov. 2003, Dallas, TX, USA, 2003.
- [56] E. Duesterwald, R. Gupta, and M. L. Soffa, "Demand-driven computation of interprocedural data flow," presented at Proceedings of 22nd Symposium on Principles of Programming Languages (POPL'95), 22-25 Jan. 1995, San Francisco, CA, USA, 1995.
- [57] E. Duesterwald, R. Gupta, and M. L. Soffa, "A demand-driven analyzer for data flow testing at the integration level," presented at Proceedings of IEEE 18th International Conference on Software Engineering, 25-30 March 1996, Berlin, Germany, 1996.
- [58] E. Duesterwald, R. Gupta, and M. L. Soffa, "A practical framework for demand-driven interprocedural data flow analysis," *ACM Transactions on Programming Languages and Systems*, vol. 19, pp. 992-1030, 1997.
- [59] Eclipse, "An open development platform," <http://www.eclipse.org/>, 2007.
- [60] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, vol. 29, pp. 210-224, 2003.
- [61] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," presented at ACM SIGPLAN '94 Conference on Programming

- Language Design and Implementation (PLDI), 20-24 June FL, USA, 1994.
- [62] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," *Operating Systems Review 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, 21-24 Oct. 2001, vol. 35, pp. 57-72, 2001.
 - [63] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319-49, 1987.
 - [64] I. Forgacs and A. Bertolino, "Feasible test path selection by principal slicing," presented at ESEC/FSE '97. 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, 22-25 Sept., Zurich, Switzerland, 1997.
 - [65] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, vol. 14, pp. 1483-98, 1988.
 - [66] K. B. Gallagher, "Visual impact analysis," presented at Proceedings of International Conference on Software Maintenance, 4-8 Nov. 1996, Monterey, CA, USA, 1996.
 - [67] K. B. Gallagher, "Some notes on interprocedural program slicing," presented at Proceedings - Fourth IEEE International Workshop on Source Code Analysis and Manipulation, Sep 15-16 2004, Chicago, IL, United States, 2004.
 - [68] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering*, vol. 17, pp. 751-61, 1991.
 - [69] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*: Addison-Wesley Longman Publishing Co., Inc., 1995.
 - [70] H. Garcia-Molina and K. Salem, "SAGAS [database transactions]," presented at Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, 27-29 May, San Francisco, CA, USA, 1987.
 - [71] B. George and L. Williams, "A structured experiment of test-driven development," *Information and Software Technology*, vol. 46, pp. 337-42, 2004.
 - [72] GFP, "A Personal Financial Management System," <http://gfd.sourceforge.net/>.
 - [73] GNU, "Grep," <http://www.gnu.org/software/grep/>.

- [74] A. Goldberg, T. C. Wang, and D. Zimmerman, "Applications of feasible path analysis to program testing," presented at 1994 International Symposium on Software Testing and Analysis (ISSTA), 17-19 Aug., Seattle, WA, USA, 1994.
- [75] GraphViz, "Dotty," <http://www.graphviz.org/>, 2007.
- [76] F. J. Gravetter and L. B. Wallnau, *Essentials of statistics for the behavioral sciences*. Wadsworth, Belmont, CA, 2000.
- [77] F. J. Gravetter and L. B. Wallnau, *Statistics for the Behavioral Sciences*, 5th ed, 2000.
- [78] N. Gupta, A. P. Mathur, and M. L. Soffa, "Automated test data generation using an iterative relaxation method," *Proceedings of the 1998 ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering, FSE-6, SIGSOFT-98, Nov 3-Nov 5 1998*, pp. 231-244, 1998.
- [79] N. Gupta, A. P. Mathur, and M. L. Soffa, "Generating test data for branch coverage," presented at Proceedings of ASE 2000 15th IEEE International Automated Software Engineering Conference, 11-15 Sept. 2000, Grenoble, France, 2000.
- [80] W. G. J. Halfond and A. Orso, "Preventing SQL injection attacks using AMNESIA," presented at 28th International Conference on Software Engineering Proceedings, 20-28 May 2006, Shanghai, China, 2006.
- [81] M. J. Harrold and N. Ci, "Reuse-driven interprocedural slicing," presented at Proceedings of the 1998 International Conference on Software Engineering, Apr 19-25 1998, Kyoto, Jpn, 1998.
- [82] M. J. Harrold, L. Larsen, J. Lloyd, D. Nedved, M. Page, G. Rothermel, M. Singh, and M. Smith, "Aristotle: a system for development of program analysis based tools," presented at Proceedings SEC'95: 33rd Annual ACM Southeast Conference, 17-18 March 1995, Clemson, SC, USA, 1995.
- [83] M. J. Harrold, R. Miller, A. Porter, and G. Rothermel, "A collaborative Investigation of program analysis based testing and maintenance," presented at 2nd International Workshop on Empirical Studies of Software Maintenance, Bari, Italy, 1997.
- [84] M. J. Harrold, G. Rothermel, and S. Sinha, "Computation of interprocedural control dependence," *ISSTA 98 ACM SIGSOFT International Symposium on Software Testing and Analysis, 2-5 March 1998*, vol. 23, pp. 11-20, 1998.
- [85] M. J. Harrold and M. L. Soffa, "Efficient computation of interprocedural definition-use chains," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 175-204, 1994.
- [86] H. R. Hartson and P. D. Gray, "Temporal aspects of tasks in the user action notation," *Human-Computer Interaction*, vol. 7, pp. 1-45, 1992.

- [87] A. E. Hassan and R. C. Holt, "Architecture recovery of web applications," presented at 24th International Conference on Software Engineering (ICSE 2002), May 19-25 2002, Orlando, FL, United States, 2002.
- [88] J. H. Hayes and J. Offutt, "Input validation analysis and testing," *Empirical Software Engineering*, vol. 11, pp. 493-522, 2006.
- [89] D. Hedley and M. A. Hennell, "The causes and effects of infeasible paths in computer programs," presented at Proceedings of the 8th International Conference on Software Engineering (Cat. No.85CH2139-4), 28-30 Aug. 1985, London, UK, 1985.
- [90] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, "Automatic design pattern detection," presented at Proceedings 11th IEEE International Workshop on Program Comprehension, 10-11 May 2003, Portland, OR, USA, 2003.
- [91] D. Heuzeroth, S. Mandel, and W. Lowe, "Generating design pattern detectors from pattern specifications," presented at Proceedings 18th IEEE International Conference on Automated Software Engineering, 6-10 Oct. 2003, Montreal, Que., Canada, 2003.
- [92] J. R. Horgan, S. London, and M. R. Lyu, "Achieving software quality with testing coverage measures," *Computer*, vol. 27, pp. 60-9.
- [93] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs" *ACM Trans. Program. Lang. Syst.*, vol. 12 pp. 26-60 1990
- [94] S. Horwitz, T. Reps, and M. Sagiv, "Demand interprocedural dataflow analysis," presented at Proceedings of the 1995 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering, Oct 10-13 1995, Washington, DC, USA, 1995.
- [95] HTMLParser, "A HTML Parser," <http://htmlparser.sourceforge.net/>.
- [96] J. C. Huang, "An approach to program testing," *Computing Surveys*, vol. 7, pp. 113-28, 1975.
- [97] M. Hutchins and K. Gallagher, "Improving visual impact analysis," presented at Proceedings of the 1998 IEEE International Conference on Software Maintenance, ICSM, Nov 16-20 1998, Bethesda, MD, USA, 1998.
- [98] G. Janee and J. Frew, "The ADEPT digital library architecture," presented at JCDL'02: Joint Conference on Digital Libraries, 14-18 July 2002, Portland, OR, USA, 2002.
- [99] R. Jasper, M. Brennan, K. Williamson, and B. Currier, "Test data generation and feasible path analysis," presented at 1994 International Symposium on Software Testing and Analysis (ISSTA), 17-19 Aug., Seattle, WA, USA, 1994.

- [100] JasperReports, "Java Reporting Library," http://www.jasperforge.org/jaspersoft/opensource/business_intelligence/jasperreports/, 2007.
- [101] Java2HTML, "A Java-HTML converter," <http://www.java2html.com/>, 2007.
- [102] G. Jayaraman, V. P. Ranganath, and J. Hatcliff, "Kaveri: delivering the Indus Java program slicer to Eclipse," presented at Fundamental Approaches to Software Engineering. 8th International Conference, FASE 2005 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2005. Proceedings, 4-8 April 2005, Edinburgh, UK, 2005.
- [103] R. Jhala and R. Majumdar, "Path slicing," *SIGPLAN Notices*, vol. 40, pp. 38-47, 2005.
- [104] R. Jhala and R. Majumdar, "Interprocedural analysis of asynchronous programs," presented at 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Jan 17-19 2007, Nice, France, 2007.
- [105] JHotDraw, "A Java GUI framework," <http://www.jhotdraw.org/>.
- [106] R. Johnsonbaugh and M. Schaefer, *Algorithms*. Sinapore: Pearson, Prentice Hall, 2004.
- [107] M. Jorgensen, "Software quality measurement," *Advances in Engineering Software*, vol. 30, pp. 907-12, 1999.
- [108] P. C. Jorgensen, *Software Testing: A Craftman's Approach*, 2nd ed: CRC Press, 2002.
- [109] N. Juristo and A. M. Moreno, *Lecture notes on empirical software engineering*: World Scientific Publishing Co., Inc., 2003.
- [110] N. Juristo, A. M. Moreno, and S. Vegas, "Limitations of empirical testing technique knowledge " in *Lecture notes on empirical software engineering* World Scientific Publishing Co., Inc., 2003 pp. 1-38
- [111] M. Kamkar, P. Fritzson, and N. Shahmehri, "Interprocedural dynamic slicing applied to interprocedural data flow testing," presented at Proceedings of the Conference on Software Maintenance, Sep 27-30 1993, Montreal, Que, Can, 1993.
- [112] G. M. Kapfhammer and M. L. Soffa, "A Family of Test Adequacy Criteria for Database-Driven Applications," presented at Proceedings of the Joint 9th European Software Engineering Conference (ESEC) and 11th SIGSOFT Symposium on the Foundations of Software Engineering (FSE-11), Sep 1-5 2003, Helsinki, Iceland, 2003.
- [113] R. K. Keller, R. Schauer, S. Robitaille, and P. Page, "Pattern-based reverse-engineering of design components," *Proceedings of the 1999 International Conference on Software Engineering, May 16-May 22 1999*, pp. 226-235, 1999.

- [114] Kestrel, "Kestrel Institute Theorem Prover," <http://www.kestrel.edu/>, 2007.
- [115] S. Khor and P. Grogono, "Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically," presented at Proceedings. 19th International Conference on Automated Software Engineering, 20-24 Sept. 2004, Linz, Austria, 2004.
- [116] A. Kiss, J. Jasz, G. Lehotai, and T. Gyimothy, "Interprocedural static slicing of binary executables," presented at Third IEEE International Workshop on Source Code Analysis and Manipulation. Proceedings, 26-27 Sept. 2003, Amsterdam, Netherlands, 2003.
- [117] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on Software Engineering*, vol. 28, pp. 721-734, 2002.
- [118] K. Kontogiannis, R. DeMori, M. Bernstein, M. Galler, and E. Merlo, "Pattern matching for design concept localization," presented at Proceedings of 2nd Working Conference on Reverse Engineering, 14-16 July 1995, Toronto, Ont., Canada, 1995.
- [119] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Automated Software Engineering*, vol. 3, pp. 77-108, 1996.
- [120] T. Kontogiannis, "User strategies in recovering from errors in man-machine systems," *Safety Science*, vol. 32, pp. 49-68, 1999.
- [121] B. Korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, vol. 16, 1990.
- [122] B. Korel, "Automated test data generation for programs with procedures," presented at Proceedings of International Symposium on Software Testing and Analysis (ISSTA'96), 8-10 Jan. 1996, San Diego, CA, USA, 1996.
- [123] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, pp. 155-63, 1988.
- [124] B. Korel and J. Laski, "STAD-a system for testing and debugging: user perspective," presented at Proceedings of the Second Workshop on Software Testing, Verification, and Analysis (Cat. No.88TH0225-3), 19-21 July 1988, Banff, Alta., Canada, 1988.
- [125] B. Korel and J. Laski, "Dynamic slicing of computer programs," *Journal of Systems and Software*, vol. 13, pp. 187-95, 1990.
- [126] B. Korel and J. Rilling, "Dynamic program slicing methods," *Information and Software Technology*, vol. 40, pp. 647-59, 1998.
- [127] A. A. Kountouris, "Safe and efficient elimination of infeasible execution paths in WCET estimation," presented at Proceedings of 3rd

- International Workshop on Real-Time Computing Systems and Applications, 30 Oct.-1 Nov. 1996, Seoul, South Korea, 1996.
- [128] J. Krinke, "Advanced slicing of sequential and concurrent programs," presented at Proceedings - 20th IEEE International Conference on Software Maintenance, ICSM 2004, Sep 11-14 2004, Chicago, IL, United States, 2004.
 - [129] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," *SIGPLAN Notices*, vol. 39, pp. 476-89, 2004.
 - [130] F. Lanubile and G. Visaggio, "Extracting reusable functions by flow graph based program slicing," *IEEE Transactions on Software Engineering*, vol. 23, pp. 246-59, 1997.
 - [131] J. H. Larkin and H. A. Simona, "Why a Diagram is (Sometimes) Worth Ten Thousand Words," *Cognitive Science*, vol. 11, pp. 65-100, 1987.
 - [132] J. Laski and W. Szermer, "Identification of program modifications and its applications in software maintenance," presented at Conference on Software Maintenance 1992 (Cat.No.92CH3206-0), 9-12 Nov. 1992, Orlando, FL, USA, 1992.
 - [133] G. Lee, J. Morris, K. Parker, G. A. Bundell, and P. Lam, "Using symbolic execution to guide test generation," *Software Testing, Verification and Reliability*, vol. 15, pp. 41-61, 2005.
 - [134] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems*, vol. 1, pp. 121-41, 1979.
 - [135] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *SIGPLAN Notices*, vol. 40, pp. 15-26, 2005.
 - [136] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: a hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, vol. 32, pp. 831-48, 2006.
 - [137] J. P. Loyall and S. A. Mathisen, "Using dependence analysis to support the software maintenance process," presented at 1993 Conference on Software Maintenance, 27-30 Sept. 1993, Montreal, Que., Canada, 1993.
 - [138] N. Malevris, "A path generation method for testing LCSAJs that restrains infeasible paths," *Information and Software Technology*, vol. 37, pp. 435-41, 1995.
 - [139] N. Malevris, D. F. Yates, and A. Veevers, "Predictive metric for likely feasibility of program paths," *Information and Software Technology*, vol. 32, pp. 115-18, 1990.
 - [140] G. Marchionini and H. Maurer, "The roles of digital libraries in teaching and learning," *Communications of the ACM*, vol. 38, pp. 67-75, 1995.

- [141] T. J. McCabe, "Structured testing: A software testing methodology using the cyclomatic complexity metric," Nat. Bur. Stand., Washington, DC, USA 1982/12/ 1982.
- [142] K. S. McKinley, "Compiler optimization algorithm for shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, pp. 769-787, 1998.
- [143] J. Melton and A. Simon, *SQL:1999: understanding relational language components*: Morgan Kaufmann Publishers Inc., 2001.
- [144] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, vol. 27, pp. 1085-110, 2001.
- [145] R. Moore, T. A. Prince, and M. Ellisman, "Data-intensive computing and digital libraries," *Communications of the ACM*, vol. 41, pp. 56-62, 1998.
- [146] H. Muccini, P. Inverardi, and A. Bertolino, "Using software architecture for code testing," *IEEE Transactions on Software Engineering*, vol. 30, pp. 160-71, 2004.
- [147] G. J. Myers, *The art of software testing*: Wiley, 1979.
- [148] M. G. Nanda and S. Ramesh, "Interprocedural slicing of multithreaded programs with applications to Java," *ACM Transactions on Programming Languages and Systems*, vol. 28, pp. 1088-144, 2006.
- [149] M. N. Ngo and H. B. K. Tan, "Applying static analysis for automated extraction of database interactions in web applications," *Information and Software Technology*, vol. In Press, Corrected Proof, 2006.
- [150] M. N. Ngo and H. B. K. Tan, "A method to aid recovery and maintenance of the input error correction features," presented at IEEE International Conference of Software Maintenance (ICSM'06), Philadelphia, USA, 2006.
- [151] M. N. Ngo and H. B. K. Tan, "Detecting large number of infeasible paths through recognizing their patterns," presented at Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, Dubrovnik, Croatia, 2007.
- [152] M. N. Ngo and H. B. K. Tan, "Empirical-based recovery and maintenance of input error-correction features," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, pp. 419-450, 2007.
- [153] M. N. Ngo and H. B. K. Tan, "Heuristics-based infeasible path detection for dynamic test data generation," *Information and Software Technology*, vol. (Artical in press) doi:10.1016/j.infsof.2007.06.006, 2007.

- [154] M. N. Ngo and H. B. K. Tan, "Applying static analysis for automated extraction of database interactions in web applications," *Information and Software Technology*, vol. 50, pp. 160-175, 2008.
- [155] M. N. Ngo, H. B. K. Tan, and D. Trinh, "Automated Extraction of database interactions in web applications," presented at 14th IEEE International Conference on Program Comprehension, Athens, Greece, 2006.
- [156] J. Niere, W. Schafer, J. P. Wadsack, L. Wendenhals, and J. Welsh, "Towards pattern-based design recovery," presented at 24th International Conference on Software Engineering (ICSE 2002), May 19-25 2002, Orlando, FL, United States, 2002.
- [157] S. C. Ntafos, "A comparison of some structural testing strategies," *IEEE Transactions on Software Engineering*, vol. 14, pp. 868-74, 1988.
- [158] O. M. G. (OMG), "Unified Language Modeling Specification (version 2.0)," 2004.
- [159] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold, "An empirical comparison of dynamic impact analysis algorithms," presented at Proceedings - 26th International Conference on Software Engineering, ICSE 2004, May 23-28 2004, Edinburgh, United Kingdom, 2004.
- [160] L. M. Ott and J. J. Thuss, "The relationship between slices and module cohesion," presented at Proceedings. 11th International Conference on Software Engineering (Cat. No.89CH2718-5), 15-18 May 1989, Pittsburgh, PA, USA, 1989.
- [161] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment " *SIGSOFT Softw. Eng. Notes* vol. 9 pp. 177-184 1984
- [162] E. M. Panainte, K. Bertels, and S. Vassiliadis, "Interprocedural compiler optimization for partial run-time reconfiguration," *Architectures, Modeling, and Simulation for Embedded Processor, Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 43, pp. 161-172, 2006.
- [163] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering: a roadmap " in *Proceedings of the Conference on The Future of Software Engineering* Limerick, Ireland ACM, 2000 pp. 345-355
- [164] N. Pettersson and W. Lowe, "A non-conservative approach to software pattern detection," presented at 15th IEEE International Conference on Program Comprehension, ICPC'07, Jun 26-29 2007, Banff, AB, Canada, 2007.

- [165] I. Philippow, D. Streitferdt, and M. Riebisch, "An approach for reverse engineering of design patterns," *Software and Systems Modeling*, vol. 4, 2005.
- [166] PHPEclipse, <http://sourceforge.net/projects/phpeclipse/>.
- [167] A. Pinsonneault and K. L. Kraemer, "Survey research methodology in management information systems: an assessment," *Journal of Management Information Systems*, vol. 10, pp. 75-105, 1993.
- [168] J. Poole, "Method to Determine a Basis Set of Paths to Perform Program Testing.," United States 1995/11/ 1995.
- [169] Y. Qi, D. Kung, and E. Wong, "An agent-based testing approach for Web applications," presented at Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005), 26-28 July 2005, Edinburgh, UK, 2005.
- [170] R. V. Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. Mississauga, Ontario, Canada: IBM Press, 1999, pp. 13.
- [171] V. P. Ranganath and J. Hatcliff, "Slicing concurrent Java programs using Indus and Kaveri " *Int. J. Softw. Tools Technol. Transf.* , vol. 9 pp. 489-504 2007
- [172] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," presented at Proceedings of 22nd Symposium on Principles of Programming Languages (POPL'95), 22-25 Jan. 1995, San Francisco, CA, USA, 1995.
- [173] F. Ricca and P. Tonella, "Analysis and testing of Web applications," presented at Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001, 12-19 May 2001, Toronto, Ont., Canada, 2001.
- [174] F. Ricca and P. Tonella, "Understanding and restructuring Web sites with ReWeb," *IEEE Multimedia*, vol. 8, pp. 40-51, 2001.
- [175] F. Ricca and P. Tonella, "Web application slicing," presented at Proceedings IEEE International Conference on Software Maintenance. ICSM 2001, 7-9 Nov. 2001, Florence, Italy, 2001.
- [176] D. J. Richardson and L. A. Clarke, "Partition analysis: a method combining testing and verification," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1477-90, 1985.
- [177] A. Rizzo, O. Parlangeli, E. Marchigiani, and S. Bagnara, "The management of human errors in user-centered design," *SIGCHI Bulletin*, vol. 28, pp. 114-18, 1996.
- [178] Robocode, "A Computer Game," <http://robocode.sourceforge.net/>, 2007.

- [179] A. Rountev, S. Kagan, and J. Sawin, "Coverage criteria for testing of object interactions in sequence diagrams," presented at 8th International Conference on Fundamental Approaches to Software Engineering, FASE 2005, held as part of the Joint Conferences on Theory and Practice of Software, ETAPS 2005, Apr 4-8 2005, Edinburgh, United Kingdom, 2005.
- [180] J. R. Ruthruff, A. S. Elbaum, and A. G. Rothermel, "Experimental program analysis: a new program analysis paradigm," presented at Proceedings of the 2006 international symposium on Software testing and analysis, Portland, Maine, USA, 2006.
- [181] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock, "A scalable approach to user-session based testing of Web applications through concept analysis," presented at Proceedings. 19th International Conference on Automated Software Engineering, 20-24 Sept. 2004, Linz, Austria, 2004.
- [182] K. Sartipi and K. Kontogiannis, "A graph pattern matching approach to software architecture recovery," presented at Proceedings IEEE International Conference on Software Maintenance (ICSM 2001), Nov 7-9 2001, Florence, Italy, 2001.
- [183] K. Sartipi and K. Kontogiannis, "On modeling software architecture recovery as graph matching," presented at International Conference on Software Maintenance, 22-26 Sept. 2003, Amsterdam, Netherlands, 2003.
- [184] J. Seemann and J. Wolff von Gudenberg, "Pattern-based design recovery of Java software," presented at ACM-SIGSOFT'98. Sixth International Symposium on the Foundations of Software Engineering, 3-5 Nov., Lake Buena Vista, FL, USA, 1998.
- [185] A. Seesing and A. Orso, "InsECTJ: a generic instrumentation framework for collecting dynamic information within Eclipse " in *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange* San Diego, California ACM Press, 2005 pp. 45-49
- [186] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," presented at ESEC/FSE'05 - Joint 10th European Software Engineering Conference (ESEC) and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13), Sep 5-9 2005, Lisbon, Portugal, 2005.
- [187] N. Shi and R. A. Olsson, "Reverse Engineering of Design Patterns from Java Source Code " in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering* IEEE Computer Society, 2006 pp. 123-134

- [188] K. Siau, "Informational and computational equivalence in comparing information modeling methods," *Journal of Database Management*, vol. 15, pp. 73-86, 2004.
- [189] S. Sinha and M. j. Harrold, "Analysis and Testing of Programs with Exception Handling Construct," *IEEE Transaction on Software Engineering*, vol. 26, 2000.
- [190] S. Sinha, M. J. Harrold, and G. Rothermel, "Interprocedural control dependence," *ACM Transactions on Software Engineering and Methodology*, vol. 10, pp. 209-54, 2001.
- [191] SIR, "<http://sir.unl.edu/portal/index.html>," *Software artifact Repository*.
- [192] D. I. K. Sjöberg, T. Dyba, and M. Jorgensen, "The Future of Empirical Methods in Software Engineering Research " in *2007 Future of Software Engineering* IEEE Computer Society, 2007 pp. 358-378
- [193] D. I. K. Sjöberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A. C. Rekdal, "A survey of controlled experiments in software engineering," *IEEE Transactions on Software Engineering*, vol. 31, pp. 733-53, 2005.
- [194] SOOT, "A Java bytecode optimization framework," <http://www.sable.mcgill.ca/soot/>.
- [195] Sourceforge, "Open-source website," <http://sourceforge.net/>.
- [196] SQLParser, "<http://sourceforge.net/projects/osqlp/>."
- [197] B. Steensgaard, "Points-to analysis in almost linear time," presented at Proceedings of 23rd Annual ACM SIGACT-SIGPLAN Symposium, 21-24 Jan. 1996, St. Petersburg Beach, FL, USA, 1996.
- [198] E. Stolte, C. Von Praun, G. Alonso, and T. Grass, "Scientific Data Repositories - Designing for a Moving Target," presented at 2003 ACM SIGMOD International Conference on Management of Data, Jun 9-12 2003, San Diego, CA, United States, 2003.
- [199] H. B. K. Tan and J. T. Kow, "An approach for extracting code fragments that implement functionality from source programs," *Journal of Software Maintenance and Evolution*, vol. 13, pp. 53-75, 2001.
- [200] W. F. Tichy, "Should computer scientists experiment more?," *Computer*, vol. 31, pp. 32-40, 1998.
- [201] W. F. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz, "Experimental evaluation in computer science: a quantitative study," *Journal of Systems and Software*, vol. 28, pp. 9-18, 1995.
- [202] S. Tilley, D. Distant, and S. Huang, "Design Recovery of Web Application Transactions," in *Advances in Software Evolution with UML and XML*, H. Yang, Ed. Hershey: PA: Idea Group Publishing, 2005.
- [203] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, pp. 121-89.

- [204] T. B. Tok, S. Z. Guyer, and C. Lin, "Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers," presented at Compiler Construction. 15th International Conference, CC 2006. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006. Proceedings, 30-31 March 2006, Vienna, Austria, 2006.
- [205] P. Tonella, "Using a concept lattice of decomposition slices for program understanding and impact analysis," *IEEE Transactions on Software Engineering*, vol. 29, pp. 495-509, 2003.
- [206] P. Tonella, "Evolutionary testing of classes," presented at ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Jul 11-14 2004, Boston, MA, United States, 2004.
- [207] S. Visvanathan and N. Gupta, "Generating test data for functions with pointer inputs," *Proceeding of the 17th IEEE International conference on Automated Software Engineering (ASE'02)*, 2002.
- [208] A. Watkins and E. M. Hufnagel, "Evolutionary test data generation: A comparison of fitness functions," *Software - Practice and Experience*, vol. 36, pp. 95-116, 2006.
- [209] M. Weiser, "Program slicing," presented at 5th International Conference on Software Engineering, 9-12 March 1981, San Diego, CA, USA, 1981.
- [210] M. Weiser, "Reconstructing sequential behavior from parallel behavior projections," *Information Processing Letters*, vol. 17, pp. 129-35, 1983.
- [211] E. J. Weyuker, "The cost of data flow testing: an empirical study," *IEEE Transactions on Software Engineering*, vol. 16, pp. 121-8, 1990.
- [212] N. Wilde, J. A. Gomez, T. Gust, and D. Strasburg, "Locating user functionality in old code," presented at Conference on Software Maintenance 1992 (Cat.No.92CH3206-0), 9-12 Nov. 1992, Orlando, FL, USA, 1992.
- [213] N. Wilde and M. C. Scully, "Software reconnaissance: mapping program features to code," *Journal of Software Maintenance: Research and Practice*, vol. 7, pp. 49-62, 1995.
- [214] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. o. Regnell, and A. Wesslén, *Experimentation in software engineering: an introduction*: Kluwer Academic Publishers, 2000.
- [215] W. E. Wong, "An integrated solution for creating dependable software," presented at Proceedings 24th Annual International Computer Software and Applications Conference. Compsac 2000, 25-27 Oct. 2000, Taipei, Taiwan, 2000.
- [216] D. Woods, R. Cook, L. Johannesen, and N. Sarter, *Behind Human Error: Cognitive Systems, Computers, and Hindsight*: Crew Systems Ergonomics Information Analysis Center, 1994.

- [217] M. R. Woodward, D. Hedley, and M. A. Hennell, "Experience with path analysis and testing of programs," *IEEE Transactions on Software Engineering*, vol. SE-6, pp. 278-86, 1980.
- [218] M. R. Woodward and M. A. Hennell, "On the relationship between two control-flow coverage criteria: all JJ-paths and MCDC," *Information and Software Technology*, vol. 48, pp. 433-40, 2006.
- [219] P. Wright, B. Fields, and M. Harrison, "Deriving human-error tolerance requirements from tasks," presented at Proceedings of IEEE International Conference on Requirements Engineering, 18-22 April 1994, Colorado Springs, CO, USA, 1994.
- [220] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: a framework for generating object-oriented unit tests using symbolic execution," presented at Tools and Algorithms for the Construction and Analysis of Systems. 11th International Conference, TACAS 2005. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005. Proceedings, 4-8 April 2005, Edinburgh, UK, 2005.
- [221] X. Xu¹, J. Bu¹, C. Chen¹, and Y. Li¹, "Achieving group undo in bitmap-based collaborative editing systems," presented at IEEE TENCON 2003. Conference on Convergent Technologies for the Asia-Pacific Region, 15-17 Oct. 2003, Bangalore, India, 2003.
- [222] D. F. Yates and N. Malevris, "Reducing the effects of infeasible paths in branch testing," presented at ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification (TAV3), 13-15 Dec., Key West, FL, USA, 1989.
- [223] K. Yi, H. Choi, J. Kim, and Y. Kim, "An empirical study on classification methods for alarms from a bug-finding static C analyzer," *Information Processing Letters*, vol. 102, pp. 118-123, 2007.
- [224] R. K. Yin, *Case Study Research: Design and Methods*, 3rd ed: Thousand Oaks, CA: Sage, 2003.
- [225] M. Young and R. N. Taylor, "Combining static concurrency analysis with symbolic execution," *IEEE Transactions on Software Engineering*, vol. 14, pp. 1499-511, 1988.
- [226] M. V. Zelkowitz and D. Wallace, "Experimental validation in software engineering," *Information and Software Technology*, vol. 39, pp. 735-43, 1997.
- [227] A. Zeller, "Program analysis: A hierarchy," presented at ICSE 2003 Workshop on dynamic analysis, 2003.
- [228] J. Zhang, X. Chen, and X. Wang, "Path-oriented test data generation using symbolic execution and constraint solving techniques," presented at Proceedings of the Second International Conference on Software Engineering and Formal Methods, 28-30 Sept. 2004, Beijing, China, 2004.

- [229] J. Zhang and X. Wang, "A constraint solver and its application to path feasibility analysis," *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, pp. 139-56, 2001.
- [230] X. Zhang, M. Young, and J. H. E. F. Lassetter, "Refining code-design mapping with flow analysis," presented at Twelfth ACM SIGSOFT International Symposium on the Foundations of Software Engineering, SIGSOFT 2004/FSE-12, Oct 31-Nov 5 2004, Newport Beach, CA, United States, 2004.
- [231] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, pp. 366-427, 1997.