Automated recovery, maintenance and testing of input validation

Liu, Hui

2009

Liu, H. (2009). Automated recovery, maintenance and testing of input validation. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/47047>

<https://doi.org/10.32657/10356/47047>

Nanyang Technological University

Downloaded on 26 Apr 2023 21:59:27 SGT

59
21
D/Blue

Automated Recovery, Maintenance and Testing of Input Validation

LIU HUI

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

Acknowledgement

I wish to express my deepest gratitude to my supervisor, Dr. Tan Hee Beng Kuan, for his guidance and support. Thank him for giving me valuable guidance on the difficulties and opportunities of different research topics and approaches. His constant encouragement has been critical to the success of the project. His sincerity, uprightness and sense of purpose have also influenced my life.

Special thanks are due to my colleagues in InfoComm Research Lab for many fruitful discussions about the research issues. Thank them for sharing ideas about the project and contributing suggestions.

Last but not least, I wish to thank my parents to whom I owe the greatest debt, and my husband Qincai for all his love and support. Their understanding, love, and care give unwavering confidence in me. I could not have come this far without their support and encouragement.

Table of Contents

Acknowledgement ii

Table of Contents..... iii

Summary vi

List of Figures viii

List of Tables x

Chapter 1 Introduction.....1

1.1 Motivation..... 3

1.2 Objective6

1.3 Major Contributions..... 7

1.4 Thesis Organization 9

Chapter 2 Background 11

2.1 Program Analysis 11

2.1.1 Control Dependence Analysis..... 13

2.1.2 Data Dependence Analysis 19

2.2 Program Slicing..... 21

2.2.1 Static Slicing Techniques..... 22

2.2.2	Other Slicing Techniques.....	26
2.3	Software Testing.....	28
2.3.1	Black-box Testing Techniques.....	28
2.3.2	White-box Testing Techniques.....	30
2.3.3	Test Data Generation.....	33
2.4	Empirical Research in Software Engineering.....	35
Chapter 3 Automated Recovery of Input Validation		39
3.1	The Analysis Model.....	41
3.2	A Method for the Recovery of Input Validation	45
3.3	Evaluation.....	47
3.3.1	The Prototype Tool	47
3.3.2	Statistical Validation.....	51
3.3.3	Case Study on the Existence of Input Validation	54
3.3.4	Case Study on the Recovery of Input Validation	58
3.4	Conclusion	60
Chapter 4 Maintenance of Input Validation.....		62
4.1	Validation Flow Graph	64
4.2	Techniques for the Maintenance of Input Validation.....	66
4.2.1	VFG-guided Slicing	66
4.2.2	Effect-oriented Decomposition Slicing	70
4.3	Evaluation.....	74
4.3.1	The Prototype Tool.....	74
4.3.2	Case Study on the Maintenance of Input Validation	77
4.4	Conclusion.....	80

Chapter 5 Testing of Input Validation.....81

5.1 Extraction of Code Behavior on Input Validation 83

5.2 A Method for the Extraction of Input Conditions..... 92

5.3 Techniques for the Testing of Input Validation 94

5.3.1 White-box Input Validation Testing 94

5.3.2 The Integrated Approach 100

5.4 Evaluation 108

5.4.1 The Prototype Tool 108

5.4.2 Experimental Design 111

5.4.3 Threats to Validity..... 116

5.4.4 Results and Discussions 117

5.5 Conclusion 122

Chapter 6 Related Work..... 124

6.1 Feature Recovery 124

6.2 Using Slicing for Program Maintenance..... 127

6.3 Input Validation Testing..... 129

Chapter 7 Conclusion..... 135

7.1 Conclusion 135

7.2 Recommendations and Future Research..... 138

Appendix 142

Author’s Publications..... 146

Bibliography..... 147

Summary

Input validation is the enforcement in software systems to ensure that only valid input is accepted to raise effects. It is essential and very important to a large class of software systems and usually forms a major part of data-intensive systems. Currently, the design and implementation of input validation are carried out by application developers. The recovery and maintenance of the input validation implemented in software systems are challenging. Furthermore, as input validation is vital to the robustness of software system, having an effective method for testing input validation is important for software quality assurance.

In this thesis, we establish the approaches for the recovery, maintenance and testing of input validation based on the techniques of program analysis, program slicing and software testing augmented with empirical properties. We present some properties that characterize the implementation of input validation and an algorithm for the automated recovery of input validation information from program source code. The information recovered provides the basis for the maintenance and testing of input validation. We then introduce a variant of control flow graph called validation flow graph for representing the overall structure of the input validation implemented in software systems and a set of

guidelines to facilitate the understanding of input validation. We extend the technique of decomposition slicing and introduce effect-oriented decomposition slicing to aid the maintenance of input validation.

We propose a method that partitions the paths through the control flow graph of a program according to the code characteristics of input validation. Based on that, we introduce two test coverage criteria for white-box input validation testing. We also propose a method to extract valid and invalid input conditions from program source code. The extracted input conditions can be used to verify and reconcile against the input conditions defined in the specification. They can also be used to aid the test case generation and selection in black-box input validation testing. This enables the integration of white-box and black-box input validation testing.

Prototype tools have been built to demonstrate the feasibility of the proposed approaches. We have also evaluated the approaches through case studies and experiments. The results show that our approaches are useful and effective in practice.

List of Figures

Figure 2.1 An example program..... 15

Figure 2.2 The CFG of the example program 15

Figure 2.3 A slice of the program w.r.t. criterion (9, sum) 25

Figure 3.1 A simplified shopping program..... 42

Figure 3.2 The CFG of the shopping program..... 42

Figure 3.3 An algorithm for the recovery of input validation..... 46

Figure 3.4 The architecture of the prototype tool IVRecovery..... 48

Figure 3.5 Code extracted from Roomba System..... 56

Figure 3.6 Code extracted from Smacs System..... 57

Figure 4.1 Comparison of the CFG and the VFG of the shopping program..... 66

Figure 4.2 The architecture of the prototype tool IVMaintenance..... 75

Figure 4.3 An algorithm for the construction of VFG..... 76

Figure 5.1 Partial specification of a credit card validation program 85

Figure 5.2 The pseudocode for the credit card validation program 86

Figure 5.3 The CFG of the credit card validation program..... 86

Figure 5.4 An algorithm for the extraction of input conditions..... 93

Figure 5.5 The architecture of the prototype tool IVTesting 108

Figure 5.6 An algorithm for the computation of a basis set 109

List of Tables

Table 2.1 Computation of the slice w.r.t. criterion (9, sum)	24
Table 3.1 Sample for hypothesis testing	52
Table 3.2 Results of the case study	55
Table 3.3 Results of the case study	59
Table 4.1 Results of the case study	79
Table 5.1 Input validation partition	87
Table 5.2 Valid and invalid input conditions.....	90
Table 5.3 Sample test suites based on path-based IVC	97
Table 5.4 Reconciliation of input conditions	105
Table 5.5 The number of faults seeded.....	113
Table 5.6 The percentage of faults observed using method 1	117
Table 5.7 The percentage of faults observed using method 2	118

Table 5.8 Complexity of the predicate at validation nodes 119

Table 5.9 The percentage of faults observed using method 3 120

Table 5.10 The percentage of faults observed using method 4..... 121

Chapter 1

Introduction

A large class of software systems process input submitted from external environment through structured interfaces. Information systems and database applications belong to this class. In such systems, an important feature is to process input submitted through various interfaces including user interfaces and file interfaces. Valid input data must conform to a specific format or rule. A system that processes input submitted is required to properly handle both expected and unexpected input so that only valid input is accepted to raise effects, while invalid input is rejected and no effects are raised. The enforcement in the system is called **input validation**. For example, in an order processing system, the outstanding balance of a customer must not exceed the credit limit. A database transaction takes place only if the new order placed by the customer does not lead to the exceeding of the limit; hence, input validation is incorporated into the system to enforce that only a valid new order is accepted for the transaction to occur.

Input validation plays a key role in the control of the input submitted to a system. Input can take many forms. Methods that are most widely used to obtain input include keyboard entries and mouse clicks. Many software applications also process file input produced by users or generated by software to obtain information. The advancement of technologies allows greater flexibility in the input form, for instance, screen touches. Various forms of input have brought in additional complexities for input validation and made software systems more error prone.

The ways of input validation range from basic checks such as input data type and format to rules on sophisticated business logic to verify whether input submitted is valid. The checking of whether an input is an integer greater than or equal to 18 or whether there is a simple “@” in an email address are examples of the basis checks. The checking of whether a user name already exists in the database involves business logic. In some software applications in which input data requirements are relatively simple, input validation can be straightforward and can be performed through basic checks. However, in majority of enterprise applications, input validation forms a vital part. In such applications, extensive validation rules are used to validate the input submitted in order to ensure the accuracy of the input and the robustness of the system.

In Web Applications, input validation also plays an important role in ensuring the security of the systems. Inadequate validation of input is the most common cause of security exploits in Web applications today. Well-known vulnerabilities include buffer overflows, SQL injection and cross-site scripting in which malformed input data are inserted into the applications and lead to serious security

attacks [129]. Input validation has been considered as one of the strongest measures of defense against today's application attacks [128]. It has become one of the most important elements in application design.

1.1 Motivation

Input validation is the most effective means to enforce the accuracy of the input submitted to a system. It is vital for the robustness of the system. It is also one of the strongest measures of defense against today's application attacks. However, not much research has been devoted to develop tools and approaches to aid the recovery, maintenance and testing of input validation. Currently, the design, implementation, and testing of input validation are usually carried out by a large population of application developers but not by a small group of highly specialized software experts.

A key activity in many software engineering tasks such as maintaining, testing, verifying and debugging is to understand and reason about various program artifacts. In practice, the cost of this activity is enormous. Program understanding starts after the features implemented in the program are recovered from the source code. In many cases, as the requirements change and functionalities evolve, software systems become largely undocumented and many of the documents become out of date or too complicated to be understood. On the other hand, software systems often contain a large number of modules each of size from hundreds lines of code to thousands of lines of code. Hence, in general, it is not obvious which parts of the code in a software system implement a specific feature, and manual recovery of the implementation of the feature is very tedious and

time-consuming. As human errors are unavoidable, developers are not confident to manually recover the feature from the code either. Therefore, the recovery of input validation is a challenging task and there is a need for a systematic approach and a tool to aid such activity.

Many of the software systems have been developed and need to be maintained for years. The maintenance activities include modifying the code to fix bugs, adding new requirements, or enhancing the structure of the code. Those changes are usually continually made to the programs after they are deployed, and the effort needed to make a change is often out of proportion to the magnitude of the change. During the course of the time, many of the documents may become inaccurate or do not exist at all. Furthermore, the software engineers who developed the system may not be available any more, or their view is outdated due to changes made by others. This creates great challenges in the maintenance of the input validation implemented in those systems.

Much research in testing specific software features to date has largely concentrated on the quality of software systems such as performance, security, configuration sensitivity, accountability, start-up and recovery [14, 85]. As input validation is vital to the robustness of software systems, to have an effective method for testing input validation is also very important for software quality assurance. Research has found that the handling of input is a notorious problem area, causing many issues at all levels of system design, implementation, testing and deployment. According to a study conducted by Perry and Evangelist [142, 143], at least 66% of faults arise from interfaces that process input submitted to the

system. In another study conducted by Nakajo et al. [132, 133], 56.9% of the faults occur at the input interfaces.

Currently, most existing methods for testing input validation are specification-based. Notably, Hayes and Offutt [78, 79] propose a systematic way to test input validation through analyzing input command syntax as defined in the requirements specifications. Their approach requires the input to be specified in a predefined format which is then used to prepare test cases for input validation testing early in the software development life cycle. Many researchers have emphasized that to test software more accurately, both the code and specification behaviors need to be tested. Therefore, it is desirable to have a systematic way for code-based input validation testing and if possible, a method that integrates code-based and specification-based input validation testing. By doing this, more accurate and thorough testing of input validation can be achieved.

Program analysis techniques that analyze program source code and automatically extract information from software systems have been used for improving the efficiency and effectiveness of many software engineering activities. Various approaches have been developed based on program analysis techniques to help developers in program understanding, software maintenance and testing; however, many of them have not been evaluated on real-world programs. Researchers have emphasized that in order to build feasible and cost-effect approaches, the design of the approaches must have feedback from empirical studies and the empirical evidences of the their effectiveness must be shown [75].

Empirical studies have been widely used in disciplines such as medicine, physics and social sciences. However, the empirical methods are not very widely disseminated among developers and researchers in software engineering research [87]. Most of the empirical studies are used to demonstrate the feasibility or compare the performance of tools, but rarely used to validate or quantify the fundamental assumptions underlying the research approaches [75]. Recently, more and more software engineering researchers have realized the importance of empirical studies and have started to explore the use of empirical methods to solve software engineering problems [8, 9, 94, 144, 157, 162, 185]. Ruthruff et al. [157] suggest that the use of empirical knowledge in program analysis might be able to draw inferences about properties of software systems in cases in which traditional program analyses have not succeeded.

1.2 Objective

The overall research objective of this thesis is to address the issues discussed in Section 1.1 and develop tools and approaches for the recovery, maintenance and testing of input validation implemented in software systems based on program analysis techniques and empirical studies. More specifically, the objective includes the followings three aspects.

- Develop a theoretical model for inferring the characteristics of the input validation implemented in software systems and develop a method for the automated recovery of input validation from program source code.

- Based on the input validation information recovered, develop approaches to aid the understanding and maintenance of input validation.
- Develop theory and approaches for more accurate and thorough testing of input validation.

1.3 Major Contributions

The work presented in this thesis has been published in [110-114]. It has the following three major contributions.

- **A theoretical model for recovering the input validation implemented in software systems [110-114].** Based on the empirical studies of many software systems, we present invariant and empirical properties that characterize the implementation of input validation. We then present an algorithm for the automated recovery of input validation information from program source code. The information recovered provides the basis for the maintenance and testing of input validation.
- **An approach for the understanding and maintenance of input validation [110, 111].** We introduce a variant of control flow graph called validation flow graph for representing the overall structure of the input validation implemented in a program. Based on the recovered input validation information and the validation flow graph captures, we present a set of program slicing rules to aid the understanding of the input validation implemented in a program.

We extend the technique of decomposition slicing and introduce effect-oriented decomposition slicing which yields only the portion of the program that relates to the implementation of input validation. As a result, changes of the source code for implementing input validation can be made with minimum ripple effects.

- **Novel methods for testing input validation [112-114].** We introduce the method of input validation partition which partitions the paths through the control flow graph of a program into different classes according to the code characteristics of input validation. We classify each class in the input validation partition into valid classes and invalid classes, from which valid and invalid input conditions can be extracted from the source code. Based on the input validation partition and the extracted input conditions, we then propose two test coverage criteria for white-box input validation testing. The two coverage criteria can also be used to measure the adequacy of any test suite for testing input validation.

We also propose the techniques for the integrated white-box and black-box input validation testing. The extracted input conditions can be used to verify and reconcile against the input conditions defined in the specification. They can also be used aid the black-box test case selection and the design of white-box test cases. As a result, both the specification and the code characteristics of input validation can be covered, and a more accurate and thorough testing of input validation can be achieved.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides the background required. It introduces program analysis techniques and program slicing techniques, discusses the fundamental software testing techniques and gives an overview of empirical research in software engineering.

Chapter 3 presents the method for the recovery of input validation. In this chapter, we first introduce the analysis model and the properties for characterizing input validation implemented in a system. We then present an automated method for the recovery of input validation from program source code. A prototype tool is presented to demonstrate the feasibility of the method. Finally, statistical validation of the empirical properties and two case studies are presented to evaluate the proposed approach.

Chapter 4 presents the techniques for the maintenance of input validation. In this chapter, we first introduce a variant of control flow graph called validation flow graph for representing the structure of the input validation implemented in a program. Based on the input validation information recovered from the source code and the validation flow graph, we then propose the techniques of VFG-guided slicing to aid the understanding of input validation, and the techniques of effect-oriented decomposition slicing to aid the modification of input validation. Finally, we present the prototype tool and report the experimental evaluation of the proposed approach.

Chapter 5 proposes the techniques for input validation testing. In this chapter, we first introduce the methods to compute the input validation partition and

extract valid and invalid input conditions from program source code. We then discuss the methods of code-based input validation testing and its integration with specification-based input validation testing. Finally, we present the prototype tool and report the experimental evaluation of the proposed techniques.

Chapter 6 compares the work presented in this thesis with related work. The use of program analysis techniques for feature recovery and the use of program slicing techniques for software maintenance are introduced. The techniques of input validation testing are also discussed.

Finally, Chapter 7 summarizes the contributions of this thesis and suggests the directions for future research.

Chapter 2

Background

This chapter provides the background required in areas of program analysis, program slicing, and software testing. Section 2.1 gives an overview of the program analysis techniques and reviews the terminologies used in control dependence and data dependence analysis. Section 2.2 introduces different program slicing techniques. Section 2.3 provides the background on software testing techniques and introduces the techniques for test data generation. Finally, Section 2.4 reviews the empirical research in software engineering.

2.1 Program Analysis

Program analysis is a technique that analyzes the behaviors of software systems. The classical use of program analysis had been targeted towards the optimization of compilers. Now it has been widely used in numerous software engineering tasks such as program understanding, design recovery, and software testing.

Program analysis can be categorized as static and dynamic program analysis. Static program analysis is a technique that analyzes the behaviors of computer programs without actually executing the programs. The analysis is performed directly on program source code without making assumptions about the input of the system. The analysis results are therefore valid for any input submitted to the program.

In contrast to static program analysis, dynamic program analysis is the analysis of program properties that hold for one or more executions by examination of the running program, usually through program instrumentation [107]. Dynamic program analysis cannot prove that a program satisfies a particular property, but it can detect violations of properties and provide useful information to programmers about the behavior of their programs. In general, static and dynamic program analysis are viewed as complementary techniques to each other in terms of the following aspects [5]:

- Static program analysis must abstract over the program in order to ensure the termination of the analysis, thus may lose important information; while dynamic analysis has the benefit of examining the concrete domain of program execution. However, dynamic program analysis can suffer from significant run-time overhead.
- Static program analysis works well at checking system properties that clearly map to source code constructs, but works poorly at checking the implications of code or inexpressible properties. Dynamic program analysis has the potential to discover semantic dependencies at a distance in the code by examine the program execution paths.

- The correctness of the results reported by static program analysis may not be strong due to the imprecise understanding of the semantics of the code. Particularly, the problem of infeasible paths can plague static program analysis. Dynamic program analysis, on the other hand, may produce incorrect result due to insufficient number of program execution.

Program analysis is used to infer characteristics of software system which typically involves system properties such as control dependencies, data dependencies, invariants, reliability or anomalous behaviors [135]. Such information supports various software engineering activities, such as test data generation [56, 66, 173, 186], program understanding [154, 171], dynamic execution profiling [26], coverage analysis [15, 41, 155, 182], and impact analysis [4, 61, 138]. Our work presented in this thesis uses static program analysis techniques to obtain extensive control dependence and data dependence information in software systems for analyzing the code behavior of input validation. Next, we shall provide the background on the techniques of control flow and data flow analysis.

2.1.1 Control Dependence Analysis

In general, the analysis control flow of a program is performed through constructing a model whose states are closely related to locations in program source code [14, 145, 161]. Control flow of a program with single procedure can be represented as an intraprocedural control flow graph, often abbreviated as control flow graph. The following definitions are commonly used when analyzing the control flow of a program.

A **control flow graph (CFG)** of a program is a directed graph $G = (N, E)$, in which N contains a set of nodes and $E = \{(n, m) \mid n, m \in N\}$ contains edges that connecting the nodes. N contains two special nodes, the entry node and the exit node, representing the entry to and the exit from the program, respectively.

A node in a CFG of a program could represent a basic block which contains uninterrupted consecutive sequence of statements in the program, or it could represent a statement in the program [161]. In our research, we follow the later representation. Hence, each node in a CFG of a program represents a statement in the program and each edge in the CFG represents possible flow of control between two statements in the program. Figure 2.1 shows an example program that reads numbers and computes the sum and the product of all positive numbers and Figure 2.2 shows its CFG.

The exit node in a CFG of a program has no out-going edge; a **predicate node** in a CFG has two out-going edges, labeled true and false; and all other nodes in a CFG have single out-going edge. The two out-going edges at a predicate node are called **branches**. The **predicate** at a predicate node is evaluated during program execution to determine which branch is to be traversed. A **simple predicate** is a Boolean variable or a relational expression possible with one or more NOT ($\sim$) operators. The predicate at a predicate node in a CFG may consist of a number of simple predicates composed by Boolean operators. The **branch predicate** is the condition under which the branch is traversed. In Figure 2.2, node 4 and node 5 are predicate nodes; T and F represents true and false respectively for the branches at each of the predicate node. For example, the

branch predicate for the branch (5, 6) is “ $n > 0$ ”, and the branch predicate for the branch (5, 8) is “ $\text{NOT}(n > 0)$ ”.

```

1  n = read();
2  sum = 0;
3  prod = 0;
4  while (n!=EOL) {
5    if (n > 0) {
6      sum = sum + n;
7      prod = prod * n;
8    }
9    n = read();
10 }
11 write(sum);
12 write(prod);

```

Figure 2.1 An example program

Figure 2.2 The CFG of the example program

A **path** in a CFG $G = (N, E)$ is a sequence of nodes $\langle n_1, n_2, \dots, n_k \rangle$ such that $k \geq 2$, where $(n_i, n_{i+1}) \in E$ for $1 \leq i \leq k-1$. A **complete path** is a path in a CFG that starts at the entry node and ends at the exit node. A path is called a **feasible path** if there is an input that causes the execution of the path; otherwise, the path is called an **infeasible path**.

Let x and y be two nodes in a CFG of a program. Node x **dominates** node y if and only if every path from the entry node to y in the CFG contains x . Node x **immediately dominates** node y if x is the closest dominator of y on any path from the entry node to y in the CFG. A **dominator tree** can be constructed such that the children of a node in the tree are all immediately dominated by the node.

Let x and y be two nodes in a CFG of a program. Node x **post-dominates** node y if and only if every path from y to the exit node in the CFG contains x . Node x **immediately post-dominates** node y if x is the closest post-dominator of y on any path from y to the end node in the CFG. A **post-dominator tree** can be constructed such that the children of a node in the tree are all immediately post-dominated by the node.

Both dominate and post-dominate relations are transitive. In the CFG shown in Figure 2.2, node 1 dominates all other nodes in the CFG as every path from the entry node to each of the other nodes contains node 1. On the other hand, node 10 post-dominates all other nodes in the CFG as every path from each of the other nodes to the exit node contains node 10. Node 6 does not dominate node 8 because the path (entry, 1, 2, 3, 4, 5, 8) does not contain node 6. In addition, node 6 does not post-dominate node 5 because the path (5, 8, 4, 9, 10, exit) does not contain node 6.

Let x and y be two nodes in a CFG of a program. Node y is **control dependent** on node x if and only if the following two conditions hold:

- 1) There exists a path p from x to y such that y post-dominates every nodes after x in p .

2) y does not post-dominates x .

In general, if node y is control dependent on node x , x must be a predicate node. The definition of control dependent can be interpreted as follows: let x be a predicate node a CFG of a program. If a branch at x always leads to node y , whereas the other branch at x does not lead to node y , then y is control dependent on x [45].

The transitive property of control dependence is inherent. The transitive control dependence can be defined based on a sequence of control dependence edges. Let x and y be two nodes in a CFG of a program. Node y is **transitively control dependent** on node x if there exists a sequence of nodes $(n_0 = x, n_1, n_2, \dots, n_k = y)$ in the CFG such that $k \geq 1$ and n_j is control dependent on n_{j-1} for all $j, 1 \leq j \leq k$.

In the CFG shown in Figure 2.2, node 5 is control dependent on node 4. Node 8 is also control dependent on node 4. However, node 9 is not control dependent on node 4 as it post-dominates node 4. Node 6 and node 7 are control dependent on node 5. Hence, node 6 and node 7 are transitively control dependent on node 4.

For programs with multiple procedures, **interprocedural control flow graph (ICFG)** can be used to represent the programs. An ICFG of a program consists of a set of CFGs that are inter-connected, one for each procedure in the program. Each procedure call in a program is represented by two nodes in an ICFG of the program, namely, a **call node** and a **return node**. A **call edge** connects a call node in a procedure to an entry node of the called procedure, while a **return edge**

connects an exit node of the called procedure to the return node of the calling procedure. Hence, a call node transfers the control flow to the called procedure through a call edge and a return node transfer the control flow back to the calling procedure through a return edge.

Various methods have been proposed to compute intraprocedural [22, 44, 57, 160] and interprocedural control dependencies [76, 115, 160, 161]. Intraprocedural control dependence analysis examines the control flow within individual procedure and ignores the transfer of control flow due to function calls. Ferrante et al. has proposed a method to determine the control dependences through construction of post-dominator tree of a control flow graph [44, 57]. Computing of post-dominator tree is equivalent to the construction of dominator tree of the reversed control flow graph which can be computed quickly using existing algorithm [108]. Subsequently, control dependence relations can be derived. Interprocedural control dependence analysis examines the control flow within each procedure and the interactive control flow among the procedures. Sinha et al. [161] have proposed two approaches for computing interprocedural control dependences. The first approach computes precise interprocedural control dependences by constructing interprocedural inlined flow graph (IIFG). An IIFG copies the CFG of the called method at each all site. As the size of IIFG grows exponentially with the size of the program and infinite for recursive calls, this approach can be too expensive to be used in practice. The second approach summarizes the control dependences that exist in different context and obtains a conservative estimate of control dependences. This approach is very efficient but at the cost of some precision.

2.1.2 Data Dependence Analysis

Data dependence analysis [2, 58, 77] gathers information about the possible set of values computed at various points in a computer program. In general, data dependence analysis computes the definitions and uses of variables of interest at a given point in a program. For example, a definition of a variable may affect the output values of the statements that use the variable. On the other hand, an output value of a statement that uses several variables would be affected by definitions of those variables.

A **definition** of a variable is an expression that modifies the variable. A **use** of a variable is an expression that references the variable without modifying it. Let x and y be two nodes in a CFG of a program. A **definition-use pair (du-pair)** with respect to a variable v in the program is a pair of nodes (x, y) , where x is a statement that defines v and y is a statement that uses v , and there is a path p from x to y in the CFG of the program along which v is not redefined. The path p is called a **definition-clear (def-clear) path** with respect to the variable v . If (x, y) is a du-pair in a program, node y is **data dependent** on node x .

Similar to the control dependence relation, the data dependence relation is also inherently transitive. Let x and y be two nodes in a CFG of a program. Node y is **transitively data dependent** on node x if there exists a sequence of nodes $(n_0 = x, n_1, n_2, \dots, n_k = y)$ in the CFG such that $k \geq 1$ and n_j is data dependent on n_{j-1} for all j , $1 \leq j \leq k$.

In the example shown in Figure 2.1, node 1, node 2 and 3 are definitions of the variable n , sum and $prod$ respectively. Node 6 is a use of the variable sum and n ;

it also redefines the variable *sum*. Node 7 is a use of the variable *prod* and *n*; it also redefines the variable *prod*. Node 8 redefines the variable *n*. A def-clear path in the CFG with respect to the variable *n* is (1, 2, 3, 4, 5, 6); hence, node 6 is data dependent on node 1. Path (6, 7, 8, 4, 9) is a def-clear path with respect to the variable *sum*; hence, node 9 is data dependent on node 6. Therefore, node 9 is transitively data dependent on node 1.

Reaching definition is a canonical example of data flow analysis, which statically determines the locations of definitions that may reach a particular point in the program source code. It can be used to determine **use-definition chain (ud-chain)** that consists of a use of a variable and all the definitions of that variable that can reach that use without any other intervening definitions. Similarly, it can be used to determine **definition-use chain (du-chain)** that consists of definition of a variable and all the uses reachable from that definition without any other intervening definitions. The identification of ud-chain and du-chain is an important step in live variable analysis which is another canonical example of data flow analysis. Live variable analysis is also referred as liveness analysis. A variable is “live” at a particular point in the program if its value at that point will be used in the future. Hence, liveness analysis calculates for each given point in a program the variables that are “live” at the exit of each given point.

Various methods have been proposed to compute intraprocedural and interprocedural data flow [1, 2, 7, 49, 72, 78, 140]. Intraprocedural data flow analysis considers the data flow within a procedure. It assumes some approximations about the definitions and uses of reference parameters and global variables at call sites. A fundamental method to perform data flow analysis is to

compute the definitions and uses of program variables which are then propagated throughout the program using a CFG representation [1, 2]. The method is then extensively used and extended in various work on data flow analysis. Interprocedural data flow analysis computes the data flow information both within and outside the procedure in which analysis on functions calls and reference parameter are conducted. Based on the method of intraprocedural data flow analysis, Horrald et al. [77] have proposed an effective method to compute interprocedural du-chain and ud-chain through iterative computation of intraprocedural data flow chains. The method is then enhanced [160] to accommodate exception handling constructs. Reps et al. [149] have shown that a large class of interprocedural data flow analysis problems that have finite states can be transformed into graph reachability problem, and precisely solved in polynomial time. Demand-driven approaches [50, 81] have also been discussed to reduce the time and space overhead in conventional interprocedural data flow analysis.

2.2 Program Slicing

Program slicing is an important application of program analysis. Typically, a program performs a large set of functions. Rather than trying to comprehend or verify all the functionalities as a whole, a more efficient way is to focus on a selected function one at time, with the goal to identify which parts of the program are relevant for that particular function. Program slicing provides such kind of support.

Program slicing is a program decomposition technique that transforms a large program into a smaller one that contains only statements relevant to the computation of a chosen set of variables or functions at some chosen point in a program. It plays very important role in various software engineering activities and it has been successfully used in program understanding, software maintenance, debugging and testing [13, 25, 62, 72, 91, 116, 154]. Surveys of program slicing and its application can be found in the literature [23, 169]. Program slicing essentially reduces the amount of data that has to be analyzed in order to comprehend a program or parts of its functionality. It has also been used in model construction to reduce the total number of states in a model.

Program slicing was firstly introduced by Weiser [176] in 1981. A **slice** is an executable subset of program statements that preserves the original behavior of the program with respect to a subset of variables of interest and at a given program point. The original notion of slice claimed by Weiser was based on the deletion of statements of the program that are irrelevant to the selected variables. Since then, various notions of program slices have been proposed, as well as a number of methods to compute slices. This diversity is mainly due to the different slicing requirements of different applications. Major types of slices are introduced in the next.

2.2.1 Static Slicing Techniques

In Weiser's approach, only statically available information is used for computing slices; hence, this type of slice is referred to as a static slice. Based on Weiser's definition, the slice is computed according to a **slicing criterion** $C = (x, V)$, where

x is a program point in program P and V is a subset of variables in P . Given a slicing criterion C , the slice consists of all statements in P that potentially affect variables in V at position x . Program slice on the slicing criterion C is a subset of program statements that preserves the behavior of the original program at the program point x with respect to the program variables in V . That is, the values of the variables in V at program point x are the same in both the original program and the slice.

In general, there are two forms of slice can be constructed: backward slice or forward slice. Backward slice contains the statements of the program which can have some effect on the slicing criterion, whereas forward slice contains those statements of the program which are affected by the slicing criterion. The original slicing approach introduced by Weiser is in a form of backward slicing. Bergeretti and Carré was the first to define a notion of a forward static slice in [17]. In backward slicing, slices are computed by gathering statements and control predicates by way of a backward traversal of the program, starting at the slicing criterion; a forward slice consists of all statements and control predicates dependent on the slicing criterion [170]. Informally, A statement being “dependent” on the slicing criterion if it satisfied one of the following two conditions:

- 1) The values computed at that statement depend on the values computed at the slicing criterion.
- 2) The values computed at the slicing criterion determine the fact if the statement under consideration is executed or not.

Backward and forward slices are computed in a similar way in a sense that the former requires tracing dependences in the backward direction while the latter requires tracing dependences in the forward direction. Both of the two approaches are very useful in practice.

Weiser's algorithm computes slices of a program from its CFG. Slices are computed by finding consecutive sets of relevant or indirectly relevant statements, according to data flow and control flow dependencies [170]. Table 2.1 summarizes the computation of the slice of the program shown in Figure 2.1 with slicing criterion $(9, \text{sum})$ by Weiser's algorithm. The column Def gives the set of variables defined at each node, Ref gives the set of variables used at each node and RelSet gives the set of variables relevant to the computation of *sum* at line 9. The column Infl gives the set of nodes that are control dependent on current node. Variables referenced in a control predicate are indirectly relevant if one of the nodes that control dependent on the control predicate is relevant.

Table 2.1 Computation of the slice w.r.t. criterion $(9, \text{sum})$

Node	Statement	Def	Ref	Infl	RelSet
1	$n = \text{read}()$	$\{n\}$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\text{sum} = 0$	$\{\text{sum}\}$	$\emptyset$	$\emptyset$	$\{n\}$
3	$\text{prod} = 0$	$\{\text{prod}\}$	$\emptyset$	$\emptyset$	$\{\text{sum}, n\}$
4	while $(n \neq \text{EOL})$	$\emptyset$	$\{n\}$	$\{5, 8\}$	$\{\text{sum}, n\}$
5	if $(n > 0)$	$\emptyset$	$\{n\}$	$\{6, 7\}$	$\{\text{sum}, n\}$
6	$\text{sum} = \text{sum} + n$	$\{\text{sum}\}$	$\{\text{sum}, n\}$	$\emptyset$	$\{\text{sum}, n\}$
7	$\text{prod} = \text{prod} * n$	$\{\text{prod}\}$	$\{\text{prod}, n\}$	$\emptyset$	$\{\text{sum}, n\}$
8	$n = \text{read}()$	n	$\emptyset$	$\emptyset$	$\{\text{sum}, n\}$
9	$\text{write}(\text{sum})$	$\emptyset$	$\{\text{sum}\}$	$\emptyset$	$\{\text{sum}\}$
10	$\text{write}(\text{prod})$	$\emptyset$	$\{\text{prod}\}$	$\emptyset$	-

The final slice of this program is shown in Figure 2.3 in which all computations that involve the variable *prod* are excluded. Note that Weiser's algorithm does not include the statement at node 9 in the slice.

```

1  n = read();
2  sum = 0;
3  prod = 0;
4  while (n!=EOL) {
5      if (n > 0) {
6          sum = sum + n;
7          prod = prod * n;
8      }
9      n = read();
10 }
11 write(sum);
12 write(prod);

```

Figure 2.3 A slice of the program w.r.t. criterion (9, sum)

An alternative method for computing static slices was suggested by Ottenstein and Ottenstein [140], who restate the problem of static slicing in terms of a reachability problem in a **program dependence graph (PDG)** [57]. A PDG is a directed graph with vertices represents statements and control predicates, and edges represent either data or control dependences. The slicing criterion is identified with a vertex in the PDG, and a slice corresponds to all PDG vertices from which the vertex of interest can be reached. Various program slicing approaches then utilize the modified or extended versions of PDGs as their underlying program representations [24, 60, 93, 134]. The most notable one is done by Horwitz et al., who extended the PDG based algorithm to compute

interprocedural slices through a **system dependence graph (SDG)** [80]. Their SDG based slicing algorithm accounts for procedure calling contexts and it demonstrates to be more accurate than the original interprocedural slicing algorithm proposed by Weiser [176].

2.2.2 Other Slicing Techniques

Korel and Laski introduced a major extension of program slicing, called dynamic program slicing [101-103]. Although the term “dynamic program slicing” was first introduced in [21], dynamic slicing may very well be regarded as a non-interactive variation of Balzer’s notion of “flowback” analysis [6, 170]. The dynamic slicing approach not only utilizes static source code information, but also dynamic information from program executions on some program input. In dynamic slicing, only the dependences that occur in a specific execution of the program are taken into account.

The strength of static slicing is that the computation of a static program slice is relatively fast compared to dynamic slicing, as no program execution is required. Its major drawback shows when static slicing has to make conservative assumptions with respect to their run-time contribution on program with conditional statements or dynamic constructs like polymorphism, pointers, aliases, etc; hence, in most of the cases, the slices produced by static slicing is larger compared to the ones produced by dynamic slicing. Dynamic slicing, on the other hand, leads to a smaller program slices and a more precise handling of dynamic and conditional language constructs. However, the computation of dynamic slices

incurs a high run time overhead due to the required recording of program executions and analysis of every executed statement.

Researchers have investigated in the techniques that reduce the size of the statically computed slices while retain the advantages of static slicing. Conditioned slicing [29, 76, 83] computes a subset of a program that preserves the behavior of the original program with respect to a slicing criterion and some specific conditions. The method allows not only the variables of interest to be specified, but also the initial conditions of interest to be specified. It statically computes the slices while involving dynamic information. Amorphous program slicing [25, 73] removes the limitation to statement deletion as the only means of simplification while retaining the semantic property that a slice preserves and produces slices that are considerably smaller than their syntax-preserving counterparts.

Hybrid approaches, where a combination of static and dynamic information is used to compute program slices are also described in the literature. Venkatesh is the first to attempt to define a hybrid slicing method called Quasi static slicing [172]. The need for hybrid slicing arises from applications where the value of some input variables is fixed while the behavior of the program must be analyzed when other input values vary. Gupta et al. propose hybrid slicing algorithms [69, 70] that integrates dynamic information from a specific execution into static slice analysis. The algorithms take advantage of both static and dynamic slicing properties in a way that, it uses static information to lower the run time overheads, and uses dynamic information for more accurate handling of dependencies. The hybrid slice produced thus is more precise than the static slice and less costly than the dynamic

slice. Rilling et al. [153] also introduce a framework to compute the static, dynamic and hybrid slices.

2.3 Software Testing

Software testing has been a very active area of software engineering research. Testing is the most commonly used method for verification of system functionalities and detection of faults so as to ensure the correctness of the system. Various testing strategies have been proposed in the literature [3, 28, 40, 43, 48, 92, 125, 130, 183]. These testing strategies can be broadly classified into black-box testing and white-box testing [85].

2.3.1 Black-box Testing Techniques

Black-box testing is specification-based. It is also referred as functional testing. In black-box testing, test cases are developed through the study of the software specification in order to verify the behavior and functionality of the software. Hence, black-box testing does not require knowledge on the internal structure and implementation of a program.

In principle, to derive a complete test suite from the specification, all possible combinations of input conditions should be considered. In practice, however, this method is too expensive to be used with limited testing resources. Hence, software testers often use their intuition and professional experiences to make certain assumptions and select a set of test cases for testing [34, 74, 131]. Many testing strategies have been developed to aid the test case design. Best known strategies

include equivalence class testing, boundary value testing and cause-effect testing [14, 131].

Equivalence class testing is a testing technique that develops a set of test cases through partitioning the input domain of a program into classes of data. This technique derives test cases by using one element from each class in the partition. Since elements in the same class are supposed to be handled identically in the program, the key to equivalence class testing is the choice of equivalence classes. Well-formulated equivalence classes can greatly reduce the potential redundancy in the designed test cases. Software testers often make the choice by second-guessing the likely implementation and the possible functional manipulations that must be presented in the implementation [85, 131].

Boundary value testing is often used to complement the equivalence class testing. It is a testing technique that generates test cases at the limits of the equivalence classes as errors are most likely to occur at the extreme ends of the input ranges. Hence, in boundary value testing, test cases need to be designed for each input range at the minimum, at the maximum, just below the maximum and just above the maximum.

Among all the black-box testing strategies, **cause-effect testing** is the only strategy that considers combinations of causes of system behavior [131, 141]. Cause-effect testing uses cause-effect graph and decision table as tools to design test cases. First, causes and effects need to be identified from specification. Cause-effect graphs are then used to identify the logic network between causes. Eventually, the decision table is built up from the graphs specifying a set of rules

on the input conditions and the actions to be performed, and subsequently a test case is designed from each rule in the table. Common arguments against the cause-effect testing method include the following two. Firstly, the identification of causes and effects may be tedious and time-consuming because of their vague description in informal specification. Secondly, since the method considers all possible combinations of causes, the size of the test suite produced could be excessively large. However, researchers have also argued that a system domain expert can often easily identify causes and effects even from an informal specification and it is worth the cost for a rigid testing [90, 141].

An advantage of black-box testing is that the test cases designed from the specification are independent of how the software is implemented; therefore, in case the implementation method changes without changing of functionalities, the test cases remain valid and useful [85]. In addition, test case can be designed as soon as the specifications are complete; hence, test case design can be conducted in advance of, or in parallel with the implementation of the software, thereby saving the overall project time. However, black-box test cases may contain significant redundancies that are infeasible to be executed due to the unknown implementation information. Furthermore, it is very difficult for black-box test cases to detect extraneous program behaviors that are not defined in the specification.

2.3.2 White-box Testing Techniques

White-box testing is code-based. It is also referred as structural testing. In white-box testing, test cases are designed based on the information derived from

program source code. Code coverage criterion is often associated with white-box testing to determine how paths in a program can be selected for testing and the adequacy of a test suite. The problem can be defined as given a program, design test cases to exercise the least set of paths in the program to meet a selected coverage criterion. In general, the stronger the coverage criterion, the more paths are needed to be exercised.

A large variety of coverage criteria have been proposed. Typical strategies include statement coverage, branch coverage, condition coverage and path coverage [14, 131]. **Statement coverage** requires that each statement in a program must be executed at least once. **Branch Coverage** requires that each branch of each predicate node in a CFG of a program must be executed at least once, i.e. the predicate at each predicate node must be evaluated to both true and false. As a predicate may consist of a number of simple predicates, **condition coverage** requires that each simple predicate in each predicate node in a CFG of a program must be evaluated to both true and false. Multiple condition coverage is more rigorous than condition coverage. For each predicate node in a CFG of a program, it requires that each combination of the truth values of the simple predicates in the predicate node must be exercised at least once. **Path coverage** requires that every possible path through a CFG of a program must be exercised at least once. Data flow coverage is a variation of path coverage that examines the sub-paths from variable definitions to subsequent references of the variables throughout the program [39, 136].

Among above-mentioned coverage criteria, statement coverage is the weakest, whereas multiple condition coverage and path coverage are much stronger criteria.

However, due to the huge number of paths in a program, path coverage is usually difficult or very costly to achieve. Many strategies have been proposed in path selection to reduce the cost on achieving path coverage. The best known path-based testing criterion is **basis path coverage** proposed by McCabe [123]. An independent path is any path through a CFG of a program that contains at least one edge that has not been traversed before the path is selected. A **basis set** is a set of linearly independent paths through a CFG that can be used to construct any other paths through the CFG. A path in the basis set is called a basis path. The number of basis paths in a program can be computed using **cyclomatic complexity** which provides a quantitative measure of the logical complexity of a program. If test cases can be designed to exercise all the basis paths in a program, every statement in the program will be exercised at least once and every branch in the CFG of the program will also be exercised at least once. Therefore, basis path coverage subsumes statement coverage and branch coverage.

As the loop construct could make the number of paths in a program infinite, **Ct coverage criterion** is introduced by Bently and Miller [16] to limit the number of loop iterations. Ct coverage criterion defines a minimum iteration count k , so that a manageable set of paths can be obtained. For example, Ct $k = 1$ coverage includes all the non-iterative paths and iterative paths that with iteration counts less than or equal 1; Ct $k = 0$ coverage identifies all the non-iterative paths and iterative paths that do not repeat any loop.

White-box testing is important to the early detection of errors in program source code during software development. The testing coverage criteria provide effective measurements on the adequacy of the testing. However, as white-box test

cases are derived program source code, white-box testing may not detect the omitted program behaviors that are defined in the specification but not implemented in code.

2.3.3 Test Data Generation

Software testing is a very labor-intensive and expensive process. It can account for up to 50% of the total cost of software development [131]. The effort and time required to create and execute test cases grow exponentially with the size of source code. As a result, many systems have many paths that had never been tested. Some problems in the system only reveal themselves after months of or years of running, in a specific circumstance. In addition, testing requires execution of the code, which can create significant practical problems. Also, the diagnosis of failures in test cases is difficult and time consuming.

Through the years, researchers and practitioners have conducted various researches on the automation of software testing in order to improve software testing process. Test data generation is one of the most challenging steps in an automated testing process, which automatically generates input data for a given program with a given condition. If the test data could be automatically generated, the cost of software development would be significantly reduced. A number of test data generation methods have been proposed [27, 38, 56, 64, 66-68, 97-100, 125, 165]. Ferguson and Korel [56] divided the major test data generation methods into three classes: 1) random, which randomly generates test data, 2) path-oriented, which generates test data to force through a selected path, and 3) goal-oriented,

which generates test data to reach a selected statement. Among them, path-oriented test data generation is the strongest and the most well researched.

Test data generation methods can be implemented statically or dynamically. Static test data generation is also referred as test data generation that uses symbolic execution [38, 148]. Executing a program symbolically means the using of variable substitution instead of actual values. The idea of the approach is then end up with an expression in terms of input variables. However, symbolic execution has many disadvantages. First, it requires plenty of computer resources. Secondly, it cannot handle dynamic data structure or function calls where information is only known at run-time. Lastly, it needs a symbolic evaluator for a particular programming language which requires a great amount of effort.

Dynamic test data generation [99] is based on actual execution of the program under test; hence, it can handle programs with pointers, arrays and function calls. In the method that dynamically generates test data, first, a random input is selected and the values of variables in the program are then known at any time of the execution. Next, through analyzing the program flow and using search algorithms, the method could eventually find a suitable input after iterative refinement. Korel has proposed a function minimization search algorithm for test data generation, which based on dynamic data flow analysis and backtracking [97]. He has also enhanced the approach to generate test data for programs with procedures [98]. However, the method that use backtracks for iterative refinement is quite expensive. It potentially requires a large number of iterations before a suitable input can be found. Furthermore, if an infeasible path is selected, the method may require a significant computational effort before the path is abandoned.

Gupta et al. have proposed an iterative relaxation method for test data generation [66]. It is a hybrid method of static and dynamic test data generation. It combines symbolic reasoning with dynamic execution, thereby mitigates problems inherited from either approach. The method also gives a promising solution for the infeasible paths problem. The method is then extended later to generate test data that exercises a selected branch in a program [67].

2.4 Empirical Research in Software Engineering

Empirical research has been playing an important role in software engineering. It has been widely used in disciplines such as medicine, physics and social science, and recently, more and more software engineering researchers have started to explore into the empirical methodologies to solve software engineering problems [157].

Empirical methods that are commonly used in software engineering research include case studies and experiments. Rather than using large samples to examine a limited number of variables, case study methods focus on a single instance or event and provide in-depth examination. While a case study is an observational study, an experiment is a controlled study [185]. In our research, experiments are used for validating hypotheses and discovering program properties during theory realization. Usually, a hypothesis refers to a provisional idea whose validity requires evaluation. In order to either confirm or disprove a hypothesis, we designed experiments, collect the data, and perform statistical validation of the hypothesis. A confirmed hypothesis may become part of a theory, and a disproved hypothesis needs to be modified. Often, through observations in the experiment, a

hypothesis can be further refined even if it has been confirmed. A useful hypothesis should enable predictions of the outcomes of an experiment or the observation of a phenomenon.

Controlled experiments are also used in our research for evaluating the proposed approach. A controlled experiment can be performed when it is difficult to control all the conditions in an experiment. Hence, a controlled experiment allows researchers to limit the number of conditions under which the observations can be made. In the design of experiment, independent variables are variables that can be manipulated and controlled during the experiment, while dependent variables are variables that are observed to change in response to the independent variables [179].

The validity of an experiment is usually classified into two types of validity: internal validity and external validity [42, 71]. An experiment has internal validity when the changes of the independent variables changes indeed cause the changes of the dependent variables within the experimental setting. An experiment has external validity when the results of the experiment can be generalized and applied to setting outside of the experiment. In many experimental designs, there is a trade-off between internal validity and external validity. For instance, an experiment in a highly controlled situation is likely to have high degree of internal validity; however, this may also limit the generalizability of the findings.

Empirical studies in software engineering have been existed for many years; however, the empirical methods are not very widely disseminated among developers and researchers [87]. Many surveys have been conducted on empirical

software engineering research and find that a large percentage of papers published in software engineering research has little or no empirical validation. Zelkowitz and Wallace [185] surveyed 612 papers in software engineering research and found that about 33% of the papers completely exclude experimentation. However, the results show that the percentage of papers that report empirical validations is gradually increasing over the years, from 63.6% in 1985 to 80.6% in 1995. Lately, Sjoberg et al. [163] conducted a survey on 5453 papers published from 1993 to 2002 in leading software engineering journals and conferences to examine the use of controlled experiments in software engineering. A well designed controlled experiment usually requires a great amount of effort and resources. The results show that only 1.9% of papers report the use of controlled experiments in their research.

Recently, more and more researchers have realized the importance of using empirical methods in software engineering research. There has been an increasing focus on using empirical methods to solve software engineering problems. Basili et al. [8] suggests to build a body of knowledge for organizing common families of studies. As isolated experiment will not lead to a larger body of knowledge, replication of experiments is emphasized by the authors for building the body of knowledge in an incremental manner. Perry et al. [144] also emphasize on the importance of replication of experiments as it can drastically improve the credibility of the empirical studies if other researchers can reproduce the results. Based on the guidelines for medical research and experiences in software engineering research, Kitchenham et al. [94] propose a set of guidelines for improving the quality of empirical studies in software engineering. The guidelines

can assist the researchers and reviewers in designing, conducting and evaluating empirical studies.

Chapter 3

Automated Recovery of Input Validation

Input validation is essential and very important to a large class of systems and usually forms a major part of data-intensive systems. In those systems, input validation controls the accuracy of the input submitted to the systems and ensure that only valid input is accepted to raise external effects. Currently, the design and implementation of input validation are usually carried out by a large population of application developers but not by a small group of highly specialized software experts. In particular, the recovery of input validation faces the following challenges.

First of all, in many cases, after the software systems have been deployed, the documents may become out of the date or do not exist at all. Hence, those documents would not be trustable and the developers are not sure about whether the input validation information obtained from them is accurate and up-to-date.

On the other hand, software systems often contain a large number of modules each of which contains hundreds or thousands of lines of code. In general, it is not obvious which parts of the code in a software system implement a specific feature. Hence, manual recovery of the input validation implemented in a program would be a tedious and time-consuming process. Furthermore, as human errors are unavoidable, developers may not be confident that the manually recovered input validation information is complete and accurate.

Therefore, the recovery of input validation is a challenging task and there is a need for an automated approach and a tool to aid such activity. Through empirical studies, we have discovered the properties for characterizing the implementation of input validation. Based on that, we propose a method for the automated recovery of input validation from program source code. All the empirical properties have been statistically validated. Case studies have also been conducted to evaluate the proposed method.

This chapter is organized as follows. Section 3.1 introduces the analysis model including the properties for characterizing input validation implemented in software systems. Section 3.2 proposes a method for the automated recovery of input validation from program source code. Section 3.3 presents the prototype tool, reports the statistical validation of the empirical properties and the case studies conducted to evaluate the proposed method. Finally, Section 3.4 concludes this chapter.

3.1 The Analysis Model

Definition 1 – Effect Node. In the CFG of a program, a node f is called an effect node if f raises external effect.

Definition 2 – Input Node. In the CFG of a program, a node u is called an input node if u reads an input directly from the external environment.

For example, a node that updates database is an effect node, and a node that reads a string directly from console is an input node. An input read from an input node u is called **the input received from u** . Figure 3.1 shows a simplified program that adds user inputted items to the shopping cart. Figure 3.2 shows the CFG of the program. In the CFG, node 4 and node 5 are input nodes, and node 15 is the only effect node in the program.

Through the empirical study, we observed that a type of nodes in a program is critical to the implementation of input validation. Next, we shall introduce an invariant property for the identification of such node.

Property 1 – Validation Node. Let u be an input node in a program. A predicate node d is a validation node of u if the following two conditions hold:

- 1) d is transitively data dependent on a node that references to the input received from u .
- 2) There is a path from d to an effect node that is transitively control dependent on d , and there is also a path from d to an exit node that does not contain any effect node.

```

...// declarations of parameters
1 boolean isExpired = checkTimeout();
2 if (isExpired)
3     showError ("You need to login again.");
  else {
4     String[] prodID = getID();
5     String[] prodCount = getCount();
6     if (prodID == null || prodCount == null)
7         out.print("Invalid prodID and prodCount.");
      else {
8         for (int i=0; i < prodID.length(); i++) {
9             String id = prodID[i];
10            String count = prodCount[i];
11            if (!toInteger(count))          // convert to integer
12                count = DefaultCount;
13            if (isExist (id)) { //Check id in PRODUCT table
14                sql = "insert into Basket (id, count)
15                    values (" + id + ", " + count + ")";
16                executeUpdate(sql);
17            }
18        }
19    }
20 }

```

Figure 3.1 A simplified shopping program

Figure 3.2 The CFG of the shopping program

Property 1 is directly implied from the fact that a validation node should check the validity of the input submitted and decide whether the input should be used to raise any external effect. In this context, we say that an effect node f is **influenced** by an input node u if f is transitively control dependent on a validation node of u , or f is transitively data dependent on a node that references to the input received from u .

Definition 3 – Validation Chain. Let d be a validation node in the CFG G of a program. The sequence of predicate nodes in G $(n_0, n_1, n_2, \dots, n_{k-1}, n_k=d)$ in which n_j $(1 \leq j \leq k)$ is control dependent on n_{j-1} forms a validation chain at d .

Note that d itself is included as the last node in the validation chain. For example, in the CFG of the shopping program shown in Figure 3.2, node 6 is a validation node for both input nodes, node 8 and node 13 are validation nodes of node 4. Node 2 is not a validation node because it is not data dependent on any input received from node 4 or node 5. It is not difficult to verify that that $(2, 6)$ is the validation chain at node 6, $(2, 6, 8)$ is the validation chain at node 8, and $(2, 6, 8, 13)$ is the validation chain at node 13.

Next, we shall present the three empirical properties we have discovered for characterizing the input validation implemented in a program. The first property states a necessary condition for implementing input validation in a program. The second and the third property are formed based on the concept of validation chain, for the inference of the decisions implemented in a program to accept or reject an input.

Property 2 – Necessary Property for Input Validation. Let u be an input node in a program P . If the input received at u is validated in P , then there exists at least one validation node of u .

Property 3 – Sufficient Property for Accepting or Rejecting Input. Let u be an input node in a program P and let D_u be the set of validation nodes of u . The validation chain at each validation node in D_u implements a decision in P to accept or reject the input received at u .

Property 4 – Necessary Property for Accepting or Rejecting Input. Let u be an input node in a program P and let D_u be the set of validation nodes of u . Any decision in P to accept or reject the input received at u is implemented by a validation chain at a validation node in D_u .

Property 2 is based on the fact that a program should properly process both expected and unexpected input, so that only valid input should be accepted for raising effects while invalid input should be rejected. Clearly, we can use Property 2 to infer whether a program fails to implement input validation. By using Property 3 and Property 4 together, we can find all the decisions to accept or reject input implemented in a program by locating the validation chains at each validation node in the program. From Property 3, all the decisions found are correct; from Property 4, there is no other decision in the program to accept or reject input. All three properties have been statistically validated through hypothesis testing. The results of the statistical validation will be reported in Section 3.3.2.

For example, in the CFG of the shopping program shown in Figure 3.2, as discussed earlier, nodes 6, node 8 and node 13 are validation nodes in the program. From Property 3, we infer that the validation chain (2, 6) at node 6 implements a decision such that if the session is not expired but the *prodID* or the *prodCount* is null, then the input should be rejected. The validation chain (2, 6, 8) at node 8 implements a decision such that if the session is not expired, *prodID* and *prodCount* are not null, but the length of *prodID* is equal to or less than 0, then the input is rejected. The validation chain (2, 6, 8, 13) at node 13 implements a decision such that if the session is not expired, *prodID* and *prodCount* are not null, the length of *prodID* is greater than 0 and the id received exists in the database, then the input is accepted and processed accordingly; otherwise, it is rejected. Property 4 can be easily verified from the shopping program shown in Figure 3.1 that the three decisions represented by the validation chains are exactly the ones implemented in the program to accept or reject the input, and there are no other such decisions exists in the program.

The presented analysis model characterizes the input validation implemented in the program. The model can effectively aid the comprehension and the maintenance of the input validation. It also provides the basis for code-based input validation testing.

3.2 A Method for the Recovery of Input Validation

Based on the analysis model discussed in the previous section, we now present the proposed methods for the automated recovery of the model from program source code. For each program in a system that accepts input and raises external effects,

first, the CFG of the program is constructed. Next, the input validation information are identified and extracted from the source code. Figure 3.3 gives an algorithm for the automated recovery that comprises three steps.

Extract_IV_Info (Program P)

Input: The source code of a program P .

Output: 1) The set U of input nodes and the set F of effect nodes.
 2) The set D_u of validation nodes for each input node u in U .
 3) The set of M of validation chains.

Step 1. Construct the CFG G of P and identify the set U of input node and the set F of effect nodes.

Step 2. For each input node u in U , compute the set D_u of validation nodes of u as follows. For each predicate node d in G , if the following three conditions are satisfied, include d in the set D_u :

- 1) d is influenced by the input received at u
- 2) There is an effect node in F that is transitively control dependent on d .
- 3) There is a path in G from d to the exit node that does not contain any effect node in F .

Step 3. Compute the set M of validation chain as follows:

- 1) Compute the union D of validation nodes $\bigcup_{u \in U} D_u$.
 - 2) For each validation node d in D , locates the set of predicate nodes on which d is transitively control dependent on, and include it in M .
-

Figure 3.3 An algorithm for the recovery of input validation

First, the CFG G of the program is constructed. Usually, source code specific tools can be used to perform static analysis, from which the control dependency, data dependency and other program characteristics can be identified. Next, the set of input nodes U and effect nodes F need to be identified. An effect node represents an output state or an action that is critical and major to a program. For example, in database applications, the statements that perform database

transactions or write to a file are obviously more critical to the ones that output warning messages. The identification of effect nodes would not be difficult to domain experts or developers who understand the program well. Hence, once the types of effects in a program have been decided, the effect nodes can be automatically identified by program analysis tools through analyzing the standard programming interfaces. The input nodes can then be identified in a similar way. Next, for each input node u in the program, the set of validation nodes of u is compute through examining the three criteria specified in Step 2. The criteria can be verified through data dependence and control dependence analysis or with the use of program slicing. Subsequently, the validation chain at each validation node is computed.

The model recovered through the three steps provides essential information on the input validation implemented in a program. Such information can help developers to quickly locate the feature in the program. It can also aid the developers in identifying the validation decisions and understanding the structure of the input validation implemented.

3.3 Evaluation

3.3.1 The Prototype Tool

To support the study of the feasibility and effectiveness of the proposed method, we have developed a prototype tool called **IVRecovery** in Java programming language. The implementation of the tool is based on the Java Architecture for

Bytecode Analysis (JABA) developed by Georgia Institute of Technology [82]. The architecture of the tool is shown in Figure 3.4.

Figure 3.4 The architecture of the prototype tool IVRecovery

JABA is a software analysis library that analyzes Java programs at the bytecode level. It is extensible API implemented in Java and is designed to installed on Sun Solaris operating system. JABA library is developed for use in software analysis

experimentation and as a foundation for further analysis. In order to use JABA, the programs must be prepared in a specified way. First, JABA requires that the files being analyzed are compiled with the debugging option of Java (`javac -g`).

Secondly, JABA reads a resource file to obtain all the information that is relevant to analyzing a program. The resource file can be generated automatically using Perl script provided in JABA package. By specifying the directory where the source files are stored, the script recursively examines all sub-directories for class files and builds a resource file. The resource file follows a predefined format that specifies values for the following fields, and then parsed by JABA.

- `ProgramName` – name of the project.
- `SourceClassPath` – source of the files to be analyzed.
- `ClassPath` – similar to the `CLASSPATH` environment variable used by the Java Runtime Environment. It is a colon-separated list of directories where JABA expects to find the classes to be analyzed, and library classes referenced by the analyzed classes.
- `ClassFiles` – a comma-separated list of classes that are analyzed for a subject. The names are fully-qualified class names, relative to the directories specified in the `ClassPath` field. Each package name of a class should be separated with a `'/'` character.

As JABA is a library, we need to write a driver to invoke its functionalities and query the analysis information. By analyzing the compiled files of Java (class files), JABA provides extensive analysis on the specified programs with information on control dependencies and data dependencies. It can also build various flow graphs

such as control flow graph, control dependence graph, dominator graph, etc. The graphs constructed can be exported into either XML or dot format. The file in dot format can be visualized by the graph layout tool supported by Graph Visualization Software (Graphviz) [65]. A typical use of JABA to build a control flow graph of a program is presented in Appendix Section 1.

Each node in a CFG produced by JABA contains the information on the type of the node, the source line number, the containing method, the definition or use and many other attributes relate to the method call (if there is any). Based on that, the Input Validation (IV) Analyzer then extracts the input validation information from the program according to the method discussed in Section 3.2. Through analyzing the standard programming interfaces, the IE-node Identifier locates all the input nodes and effect nodes in a program. The Dependency Analyzer analyzes the control dependencies and data dependencies between two specific nodes using the information provided by JABA API.

Based on the dependencies information, the V-node identifier examines each predicate node and the associated paths, and subsequently identifies the validation node for each input node in the program. For each predicate node d in a program, it examines whether d is a validation node of an input node u through the following three criteria: 1) The evaluation of the predicate at d must be influenced by the input received from u . 2) There is an effect node that is transitively control dependent on d . 3) There is a path in G from d to the exit node that does not contain any effect node in F .

The V-chain Identifier then identifies the validation chain at each validation node based on the control dependencies information as follows: For each validation node d in the program, the predicate nodes on which d are transitively control dependent are identified through control dependency analysis. According to the order identified in reverse, they form the validation chain at d .

3.3.2 Statistical Validation

We have conducted a hypothesis testing to statistically validate the three empirical properties with the aid of the prototype tool. The testing is based on binomial test [126] which computes the statistic z as follows:

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

where n is the sample size for the test, X is the number of cases that support the alternative hypothesis H_1 , and p is the hypothesized value for the proportion of cases in the population that support H_1 .

For the hypothesis testing of each of the three properties, the null hypothesis H_0 states that the property holds for less than 99 percent of the cases, and the alternative hypothesis H_1 states that the property holds for equal or more than 99 percent of the cases. Therefore, $p = 0.99$ in our test. Choosing 0.05 as the type I error probability, we get z-score 1.645. Hence, if the z-score calculated from a sample is greater than 1.645, we reject H_0 ; otherwise, we accept H_0 .

Our sample for the hypothesis testing was drawn from the industrial projects, open source systems and student projects. The details of the sample drawn are listed in Table 3.1.

Table 3.1 Sample for hypothesis testing

	Systems	LOC	No. of Sample Cases
1	Contest System	59151	53
2	Hospital E-Services	41358	31
3	Roomba [164]	4212	16
4	JavaLibrary [164]	12180	16
5	Smacs [164]	13462	24
6	Bugtracker [164]	1755	13
7	JspShop [175]	8131	29
8	ART [164]	12841	33
9	NMS [175]	3617	12
10	StudentRecord [175]	2847	14
11	SubjectRegister	6854	13
12	FriendsMatch	14365	23
13	EasyBooking	4194	15
14	ResaleOnline	11540	27
15	PersonalBlog	1639	9
16	FlowerShop	2082	11
	TOTAL	200228	339

The first and second systems were collected from industrial projects. The first system is to support contest management for team, district or agency in an insurance company. The second system is to support patient management in a hospital. The third to the tenth systems were collected from the open-source.

These eight open source systems were downloaded from Soureforge [164] and Webmaster [175]. The eleventh to the sixteenth systems were collected from student projects which were developed by senior computer science students. Our sample contains programs in various stages of development. The two systems collected from the industry have been fully tested and deployed. The open-source systems were supposed to be tested, but the degree of testing is unknown. For systems collected from student projects, only some initial unit tests had been carried out.

We analyzed the 16 systems through the use of part of the prototype system discussed in Section 5. A sample case is an independent function unit that accepts input from external environment to raise effects, thus its CFG could be an intra-procedural or an inter-procedural CFG. As shown in Table 3.1, there were altogether 339 sample cases collected from those systems. To verify Property 2, for each sample case, we computed the validation nodes and carefully examined the input validation implemented in the program. To verify Property 3 and Property 4, for each sample case, we computed the validation chains at each validation node and examined the decisions implemented; we also conducted manual check on whether there were validation decisions in the program that are not covered by the validation chains computed.

The results collected from all the sample cases gave affirmative support to the three properties. Applying formula (1), the z-score calculated is 1.85 ($p = 0.99$ and $X = n = 339$); hence, we reject H_0 and conclude that Property 2, Property 3 and Property 4 hold for at least 99 percent of the cases at 5 percent level of significance.

Though all three properties hold for all the sample cases, the hypothesis testing is needed because theoretically a program can implement the input validation in infinite number of ways and the three properties do not necessarily hold. When conducting empirical study, there are always some limitations. The results of the binomial tests are influenced by the choice of the sample. Much effort has been made to make sure that sample used in the test is randomly selected from a large pool of software systems of different size, at various stages of development, and from different sources. Even though, the sample is still far from covering all possible program behaviors in various application domains which is impossible to achieve with limited time and resources. This may pose threats to the generalization of the results.

3.3.3 Case Study on the Existence of Input Validation

A case study was conducted on 8 open source systems, which are also used in the hypothesis testing. All the 8 systems are supposed to be tested. As discussed earlier, Table 3.1 shows the number of programs in those systems that accept input from external environment to raise effects. Based on Property 2, we then did a general check on the number of programs in these systems in which input validation is not implemented.

The 8 systems are briefly described below. Roomba is a web-based room booking system for small to medium-sized hotels. Smacs is a web-based facility for the management of casual staff in an organization. JavaLibrary is an electronic library in which registered users can browse books, journals, megazines, etc. Bugtracker is a web-based tool for bug reporting and bug tracking in software

development. Art is a query and reporting tool for information sharing on the web. JspShop is a commercial online shopping system. NMS is a news management system for updating and editing news releases. StudentRecord provides web-based educational facility for teachers and students to manage their courses and records.

The prototype tool was used to process the 8 systems. The summary of the analyzed results is shown in Table 3.2.

Table 3.2 Results of the case study

Systems	# Total programs	# Input programs	# Programs w/o IV
Roomba [164]	59	16	7
JavaLibrary [164]	79	16	0
Smacs [164]	209	24	8
Bugtracker [164]	25	13	0
JspShop [175]	93	29	0
ART [164]	88	33	5
NMS [175]	48	12	9
StudentRecord [175]	41	14	0
Total	642	157	29

The column *# Input Program* gives the number of programs in each system that processes input submitted to raise effects, and the column *# Programs w/o IV* gives the number of programs in which input validation is not implemented. In all, in the 8 systems, there are 157 programs that accept input from external environment to raise effects, which is 24% of the total number of programs. Among them,

there are 29 programs in which input validation is not implemented, which accounts for 18%.

The figures show that input validation is an important feature in many software programs; however, implementation of input validation is often neglected by programmers. Through manual investigation of those programs in which input is not validated, it is not surprising to see that most of them could cause potential problems such as unexpected runtime exceptions, security vulnerability, or violation of database integrity. Two typical examples are shown in Figure 3.5 and Figure 3.6.

```
// Roomba - bookings/saveBooking.jsp
...
1  boolean newBooking = false;
2  if ((getParam("newBooking", request).equals("true")))
3      newBooking = true;
4      String id = getParam("id", request);
5      String customerid = getParam("customerid", "0", request);
6      String roomid = getParam("roomid", "0", request);
...
7  if (newBooking) {
8      sql = "INSERT INTO ROOMBA Bookings(customerid,
           roomid, ...) VALUES (" + customerid + ", " + roomid +
           ... + ")";}
9  else {
10     sql = "UPDATE ROOMBABookings SET" +
           "customerid = " + customerid + ", " + "roomid = "
           + roomid + ... + "WHERE (id = " + id + ")";}
11  DBConnector.executeUpdate(sql);
...
```

Figure 3.5 Code extracted from Roomba System

```

// Smacs - insertTestingCard.jsp
...
1  <jsp:useBean class="casualstaffhr.TestingCard" id="bean"/>
...
2  if ( request.getParameter( "number" ) != null ){
3      bean.setNumber( request.getParameter( "number" ) );
    }
...
4  if ( request.getParameter( "description" ) != null ){
5      bean.setDescription(
        request.getParameter("description") );
    }
6  bean.create();
...

```

Figure 3.6 Code extracted from Smacs System

The code segment in Figure 3.5 is extracted from the *Roomba* system, in which input are accessed at line 2, 4, 5 and 6, and the system database is updated at line 11. The program reads user input and uses them directly in database insertion or modification without any checking on the input values, and this can cause two problems. One is that an invalid input can result in Java SQL runtime exception if the value does not comply to its entry type defined in the database schema (e.g., inserting a string into a database where an integer is required). The other one is that the program is very easy to be attacked by SQL injection which will cause security issues.

The other example shown in Figure 3.6 is the code segment extracted from the Smacs system. In the code, *bean* is an instance of the class *TestingCard* as defined at line 1. The program checks the input data at line 2 and 4 and set the corresponding field of *bean* if the value is not null. However, though a domain checking is performed on the input, the method *bean.create()* at the line 6 will always

be executed regardless of the results of the checking. This program can confuse the users who provide the invalid input. It can also cause the insertion of invalid records into the database.

3.3.4 Case Study on the Recovery of Input Validation

We have conducted another case study to evaluate the effectiveness of the proposed method for the recovery of input validation. The case study was performed on the open source system Smacs which is also used in the case study discussed earlier. There are altogether 80 files in the system and the size of the source code is 13462 LOC. There is a report available that describes the development of the system but the design documentations of the system are very limited.

Senior software engineers and final year students voluntarily participated in this case study. Among them, we selected four students who scored A in their software engineering project so as to minimize the impact of a particular strong participant or a particular weak one; we also selected two engineers that have more than three years' working experience. Each participant was given four hours to study the Smac system before the case study commences. We also conducted two-hour training to them on the concept of input validation.

In this case study, two engineers and two students were asked to manually recover the input validation implemented in a system. The Java development tool Eclipse [54] had been set up to aid them in exploring the source code. Each participant was required to perform the following tasks independently:

- 1) Locate all the files in the system that implements input validation.
- 2) Identify the effects raised in the system upon accepting input from external environment.

As discussed earlier, there are 8 programs in Smacs that do not implement the input validation (as shown in Table 3.2). By using the prototype tool, we also checked that there are 18 decisions to accept the input and 61 decisions to reject the input submitted in the system.

We recorded the time each participant spent in this case study and also examined the correctness of each task performed. The results are shown in Table 3.3.

Table 3.3 Results of the case study

Participant	No. Files Located	No. Effects Identified	Time Spent (minutes)
Engineer 1	49	35	120
Engineer 2	53	38	135
Student 1	50	34	180
Student 2	42	29	165
IVRecovery	59	41	<1

Not surprisingly, compare to the students, senior software engineers spent much less time on the task and the information recovered by them is more complete. All the participants commented that the task was very tedious and time consuming, not to mention that they had spent several hours to study the system before the case study starts. On the other hand, the prototype tool spent less than

1 minute on a Pentium 4 PC to analyze the system. It identified altogether 59 files that implement input validation and 41 effects raised in the system upon accepting external input. In addition, 79 decisions were automatically extracted from the system, 18 of which are for the acceptance of input and the rest are for the rejection of input.

The results clearly show that the recovery of the input validation feature implemented in a system is a highly non-trivial task. Even for the software engineers who have more than three years' experience, a great amount of time and effort are needed on understanding the system and the input validation feature implemented, yet, the information cannot be completely and accurately recovered through manual investigation of the source code.

3.4 Conclusion

In this Chapter, we have introduced an analysis model that characterizes input validation. A method has been proposed for the automated recovery of input validation from program source code. We have also presented some empirical properties for implementing input validation in a program. All the empirical properties have been statistically validated.

Two case studies have been carried out to examine the importance of input validation and to evaluate the effectiveness of the proposed method. The results of the first case study suggest that software developers should not overlook the input validation feature because improperly implemented input validation could be the cause of many problems such as unexpected runtime exceptions, security

vulnerability, or violation of database integrity and seriously deteriorate the quality of the software. Input validation should be properly designed, implemented and verified during the software development process. The results of the second case study show that the manual recovery of input validation could be very tedious and time-consuming, yet accurate information on input validation cannot be fully recovered. The proposed method, as implemented in the prototype tool, can effectively recover input validation information from the source code in a systematical way. Such information provides the basis for aiding the understanding and maintenance of input validation. It can also be used to support white-box input validation testing.

Chapter 4

Maintenance of Input

Validation

Many of the software systems have been developed and need to be maintained for years. During the course of the time, as the requirements change and functionalities evolve, the system documents become too complicated to be understood or not trustable because the developers are not sure that all the changes have been updated. Furthermore, the original software engineers who developed the system may not be available any more, or their view is outdated due to changes made by others.

Maintenance activities include modifying the code to fix bugs, adding new requirements, or enhancing the structure of the code. Those changes are usually continually made to the systems after they are deployed. Maintenance is a difficult activity and the effort needed to make a change is often out of proportion to the magnitude of the change.

All of the aforementioned create great challenges for the maintenance of the input validation implemented in software systems. When an update of the input validation implemented in a program occurs, the developers have to understand how it was implemented before they can initiate the modification. In Chapter 3, we have proposed the method for the automated recovery of input validation from program source code. Based on the recovered input validation information, we propose in this chapter the techniques that can be used to aid the understanding and maintenance of input validation. We first propose a variant of control flow graph, called validation flow graph (VFG) for representing the overall structure of the input validation implemented in a program. We then propose the VFG-guided slicing techniques that can be used to facilitate the understanding of input validation. Based on the technique of decomposition slicing, we also propose the effect-oriented decomposition slicing which can be used to aid the modification of input validation. A case study has been conducted to evaluate the proposed techniques.

The rest of this chapter is organized as follows. Section 4.1 introduces the validation flow graph. In Section 4.2, techniques to aid the maintenance of input validation are proposed. Section 4.2.1 introduces VFG-guided slicing and Section 4.2.2 discusses the effect-oriented decomposition slicing. Section 4.3 presents the prototype tool and reports the results of the case study. Finally, Section 4.4 concludes this chapter.

4.1 Validation Flow Graph

In Chapter 3 Section 3.1, we have discussed the effect node (definition 1), input node (definition 2), validation node (property 1) and validation chain (definition 3) in a control flow graph of a program. Next, we shall introduce a variant of control flow graph, called validation flow graph which provides a structural overview of the input validation implemented in a program. The formal definition of the validation flow graph is given below:

Let $G = (N, E)$ be the CFG of a program where N and E are its set of nodes and edges respectively. Let N' be the subset of nodes in G that include the following types of nodes:

- 1) The entry node and the exit node.
- 2) All the input nodes.
- 3) All the effect nodes.
- 4) Node n in G such that a node in N' is control dependent on n .

Let E' be the set of edges that connects the nodes in N' . For each unordered pair of nodes (n, m) in N' , if there is a path in G from n to m without passing through another node in N' , an edge (n, m) in is included in E' . It is formally defined as follows.

$$E' = \{(n, m) \mid n, m \in N', n \neq m \text{ and there exists a path } (n, n_1, \dots, n_k, m) \text{ in } G \text{ such that } k \geq 0 \text{ for all } j, 1 \leq j \leq k, n_j \notin N'\}$$

The flow graph $H = (N', E')$ is called the **validation flow graph (VFG)** of the program, where N' and E' are its set of nodes and edges respectively.

The following property holds in H :

Let $N' = (n_0=entry, n_1, \dots, n_{k-1}, n_k=exit)$. For each $0 \leq j \leq k-1$, if (n_j, n_{j+1}) is an edge in G , it is also an edge in H ; otherwise, if there is a path from n_j to n_{j+1} in G , it is also an edge in H .

Figure 4.1(a) and Figure 4.1(b) gives the CFG and VFG of the shopping program (shown in Figure 3.1) discussed in Chapter 3, respectively. As discussed earlier, node 4 and node 5 are input nodes and node 15 is the only effect node in the program. Since node 15 is control dependent on node 13, node 13 should be included in the VFG of the program. Through control dependency analysis, it is easy to verify that node 8, 6, and 2 should also be included in the VFG. Though at node 11 the program checks whether the variable *count* can be converted to an integer, it does not control the execution of node 15, thus node 11 is not be included in the VFG of the program.

(a) The CFG of the shopping program

(b) The VFG of the shopping program

Figure 4.1 Comparison of the CFG and the VFG of the shopping program

Clearly, the structure of the VFG shown in Figure 4.1(b) is much simpler compared to the CFG shown, yet the VFG captures the most critical information on the input validation implemented in the program.

4.2 Techniques for the Maintenance of Input Validation

4.2.1 VFG-guided Slicing

The techniques introduced in this section are based on program slicing. Through data flow and control flow analysis, program slicing decomposes a program and extracts the code segments that influence or are influenced by the concerned variables in the program. Program slicing has been widely used program understanding and software maintenance [23, 46, 169].

During the maintenance of the input validation implemented in a program, software developers often need to know how an input submitted to the program is used, how an effect in the program is executed, and how the validations are conducted in the program. Over the time, as the requirements change and functionalities evolve, it is often that the input to be read, the external effect to be raised, or the decisions to accept or reject input in the program need to be updated. To aid software developers in such activities, we propose next the method of **VFG-guided slicing** which augments the traditional program slicing with the input validation information recovered from program source code. The method consists of three major steps:

- 1) Identify the slicing criterion.
- 2) Select slicing algorithms.
- 3) Perform program slicing.

Let H be the VFG of a program P . Next we shall present and elaborate the VFG-guided slicing guidelines.

Rule 1: An input node u in H suggests a forward slicing with respect to the criterion (u, v) , where v is the variable defined at u . The computed slice consists of all the statements in P that might be affected by the input read at u .

Rule 2: An effect node f in H suggests a backward slicing with respect to the criterion (f, V) , where V is the set of variables referenced at f . The computed slice consists of all the statements in P that might affect the execution and computation of f .

Rule 3: A validation node d in H suggests a backward slicing with respect to the criterion (d, V) , where V is the set of variables referenced at d . The computed slice consists of all the statements in P that might affect the execution of d or the evaluation of the conditions in d .

By following Rule 3, we can further examine the type of validation performed at each validation node. As defined by Tai [166], a simple predicate is a Boolean variable or a relational expression possible with one or more NOT ($\sim$) operators. In general, a predicate node consists of a number of simple predicates composed by Boolean operators. In information systems, the type of checking carried out by a simple predicate in a validation node can be classified into the following types:

- 1) Control Check: The predicate is not influenced by any input nodes, but references to some system predefined variables. For example, the checking of logged in status or system timeout.
- 2) Domain Check: The predicate is only influenced by a single input node. This type of checking is commonly used to ensure the value of an input falls within a specific range.
- 3) Intra-Input Check: The predicate is influenced by more than one input nodes. It is often used to check the consistency between different input data submitted to a program.
- 4) Input-Database Check: The predicate is influenced by some input nodes, and also references to some data retrieved from system database.

Through applying Rule 1 and Rule 2, it is easy to see that an input node usually influences more than one effect node and an effect node is usually influenced by more than one input nodes. Therefore, Rule 1 and Rule 2 may not be sufficient to help developers in understanding the complex relations between multiple input nodes and multiple effect nodes. In such cases, program chopping [104, 150] can help. A chop $C(x, y)$ of a program is computed with respect to the source statement x and a target statement y in the program, and (x, y) is called chopping criterion. The chop $C(x, y)$ consists all the statements in P that account for the influences of the statement x onto the statement y . In an intraprocedural case, a chop is basically the intersection of a backward slice at y with a forward slice at x [104]. Compared to program slicing, program chopping provides a more focused approach to reveal the transitive dependency from one statement (the source) to another (the sink) throughout the program.

Rule 4: Let $S(u, v)$ be the slice computed through Rule 1, where u is an input node in H . Let $S(f, V)$ be the slice computed through Rule 2, where f is an effect node in H . If node f is contained in $S(u, v)$ or node u is contained in $S(f, V)$, the chop $C(u, f)$ can be computed as

$$C(u, f) = S(u, v) \cap S(f, V).$$

The computation of the chop $C(u, f)$ can follow one of the following two methods:

- 1) Compute the forward slice $S(u, v)$ first, and select an effect node f of interest. Next, perform backward slicing with respect to (f, V) during which only statements contained in $S(u, v)$ are included in $C(u, f)$.

2) Compute backward slice $S(f, V)$ first, and select an input node t of interest.

Next, perform forward slicing with respect to (u, v) during which only statements contained in $S(f, V)$ are included $C(u, f)$.

The method of VFG-guided slicing helps the developers to identify the statement of interest and suggests appropriate slicing algorithm, and the program slices can then be computed accordingly. The techniques discussed provide effective ways to decompose the program into smaller pieces with respect to the input nodes, validation nodes and effect nodes in a program, which facilitate the understanding and maintenance of input validation. By applying those techniques interactively and iteratively, the complex relations among the input read, validations performed and effects raised in a program can be sorted out; the impact of making modifications can also be evaluated.

4.2.2 Effect-oriented Decomposition Slicing

The problem of ripple effects is critical in software maintenance activities. For instance, if a validation decision for executing an effect needs to be modified in a program, the changes to the code may affect other effects to be taken place or cause potential inconsistencies in other parts of the program – those are the ripple effects of the modification that requires much effort to investigate and revalidate.

The decomposition slicing technique is introduced by Gallagher [59, 62] for eliminating the ripple effect induced by the modification of software programs. A decomposition slice $S(v)$ captures all relevant computations on the variable v in a program, and is independent of any program point. A brief summary of the terms follows. Two decomposition slices are **independent** if they have no statements in

common. A decomposition slice is **strongly dependent** on another slice if it is a subset of the latter. A decomposition slice is **maximal** if it is not strongly dependent on any other. A statement is **dependent** if it is included in more than one decomposition slices; otherwise, it is **independent**. A variable is *dependent* if it is the target of a dependent assignment statement; it is independent if all assignments statements that define the variable are independent statements. A **complement** of a decomposition slice is defined as the statements in the original program minus all the independent statements in the slice.

In general, modifications of a program can take three forms: addition, deletion and change of the code. A change can be treated as a deletion followed by an addition. Hence, Gallagher proposes to determine only the statements that can be deleted and the forms of statements that can be added in a decomposition slice. The following set of guidelines is proposed to ensure that any modification of decomposition slices must keep its complement intact [62].

- 1) Independent statements may be deleted from a decomposition slice.

Independent statements do not affect data flow or control flow in the complement; hence, they can be removed from a decomposition slice without affecting the complement.

- 2) Assignments statements that target independent variables can be added anywhere in a decomposition slice. Independent variables are unknown to the complement. Therefore, inclusion of assignment statements that involve independent variables cannot affect the computation of the complement.

- 3) Logical expressions and output statements maybe added anywhere in a decomposition slice. Inclusion of output statements or evaluation of logical expressions will not affect the computation of the slice, thereby cannot affect the complement. However, care must be taken when adding statements that are control dependent on the logically expressions as they may interfere with the complement.
- 4) New control statements that surround any dependent statement will cause the complement to change. Newly added control statements will be included in the complement as the dependent statements are included in both the slice and the complement. Therefore, any control statements that control the dependent statements will be included in both the slice and the complement.

As we are interested in the input validation feature, we borrow the concept of decomposition slicing and introduce the **effect-oriented decomposition slice**, which is the union of the decomposition slices taken at the effect nodes on the variables referenced at those nodes. Let F be a set of effect nodes that are influenced by some input nodes in a program. The effect-oriented decomposition slice $S(F)$ is defined as follows:

$$S(F) = \bigcup_{f \in F} \bigcup_{v \in Var(f)} S(v)$$

where $Var(f)$ is the set of variables referenced at an effect node $f \in F$, and $S(v)$ is the decomposition slice taken on a variable $v \in Var(f)$.

The effect-oriented decomposition slice with respect to all the effect nodes in a program yields only the portion of the program that relates to the implementation of the input validation, with unrelated statements deleted and dependent statements restricted from modification. It can be constructed to serve as a reference for any modification to the code that implements the input validation feature in a program. According to Property 2 and 3 discussed in Chapter 3, a decision in a program P to accept or reject input always corresponds to a validation chain at a validation node in P . Since the input validation information has been recovered from the source code of P , to modify a decision in P for accepting input and executing some effects, the validation chain that implement the decision and all the related code segments can be identified through examining the VFG of P and performing VFG-guided slicing on P . Then, any change of the code should follow the guidelines given by Gallagher as discussed earlier. This is to make sure that any modification of the code that implements the input validation will not affect other components in P .

Upon completion of the modification, the modified slice can be merged back into the complement according to the following algorithm proposed by Gallagher [62]. In this algorithm, the unchanged dependent statements are used to guide the reconstruction of the modified program. It is an editing process that is viewed as a modification to the entire program.

- 1) Order the statements in the original program according to the line number.

A program slice and its complement can then be identified based on the sequence of statement numbers from the original program. The sequence

numbering of the slice and its complement are referred to as the slice sequence and the complement sequence, respectively.

- 2) For deleted statements, delete the sequence numbers from the slice sequence. Since only independent statements are deleted, the numbers are not in the complement sequence.
- 3) For 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 $F = \min(\text{int}(P+1), M)$. Theoretically, the new statement can be inserted at sequence number $(F+P)/2$.
- 4) The merged program is obtained by merging the modified slice sequence values into the complement sequence.

4.3 Evaluation

4.3.1 The Prototype Tool

We have developed a prototype tool **IVMaintenance** extended from the tool **IVRecovery** introduced in Section 3.3. The architecture of the **IVMaintenance** is shown in Figure 4.2.

Figure 4.2 The architecture of the prototype tool IVMaintenance

As mentioned in Section 3.3, the **IVRecovery** recovers the input validation information from program source code. Based on the CFG and the information recovered, the VFG Constructor produces a VFG for each program that accepts external input to raise effects. In addition, the component generates a list of suggested slicing guidelines according to the slicing rules discussed in Section 4.2.1.

An algorithm for constructing a VFG from a CFG of a program is given in Figure 4.3. The basic idea is to remove all the unnecessary nodes from the corresponding CFG.

ConstructVFG

Input: 1) The CFG G of a program.
 2) The set U of input nodes
 3) The set F of effect nodes.

Output: The VFG of the program.

Begin

Step 1. Delete from G all the nodes except the predicate nodes, the input nodes and the effect nodes as follow.

for each node n in G , do

if n is not a predicate node, then

if n is not contained in U and F , then

delete n from G by adding an edge between each predecessor of n and the successor of n .

Step 2. Delete from G all the unnecessary predicate nodes as follows.

1) Compute the set S of predicates on which an input node or an effect node is transitively control dependent on.

2) **for each node n in G , do**

if n is a predicate node,

if n is not contained in S ,

delete n from G based on the following two cases:

i) if n is a loop construct, identify the edge d that is not a back edge, that add an edge between each predecessor of n and the destination of d .

ii) Otherwise, add an edge between each predecessor of n and the destination of any outgoing edge of n .

End

Figure 4.3 An algorithm for the construction of VFG

The algorithm consists of two steps. As a predicate node has two outgoing edges, we first keep all the predicate nodes and delete all other nodes that are not input nodes or effect nodes. A non-predicate node has maximum two incoming edges and one outgoing edges. According to the definition of VFG, the predicate nodes on which an input node or an effect node is transitively control dependent on need to be included. Hence, after the first step, a predicate node needs to be

deleted only if there is no node in the graph is control dependent on it. Consequently, only two possible types of predicate nodes need to be deleted: 1) a predicate node that has an outgoing edge directly pointing to itself (loop construct), and 2) a predicate node that has two outgoing edges point to the same destination. As such, the unnecessary predicate nodes can be deleted according to the situations.

The IV Slicer is a user interactive component for performing different types of slicing discussed in Section 4.2. Based on the slicing guidelines generated by the VFG Constructor, users select a slicing criterion and a slicing algorithm for a program, and the IV Slicer will compute the slice accordingly. The Basic Slicer works directly on the CFG generated by the IVRecovery and implements backward and forward slicing algorithms. The source code for implementing the backward slicing algorithm based on JABA is presented in the Appendix Section 2. By using the input validation information, the EOD Slicer computes the effect-oriented decomposition slice of the selected program. It also produces a report about the dependencies among the program statements to aid the maintenance activities.

4.3.2 Case Study on the Maintenance of Input Validation

With the aid of the prototype tool, we have conducted a case study to examine the feasibility and effectiveness of the proposed techniques in aiding the maintenance of input validation. The case study was performed on the open source system Smacs [164]. There are altogether 80 files in the system and the size of the source code is 13462 LOC. There is a report available that describes the

development of the system but the design documentations of the system are very limited.

Senior software engineers and final year students voluntarily participated in this case study. Among them, we selected four students who scored A in their software engineering project so as to minimize the impact of a particular strong participant or a particular weak one; we also selected two engineers that have more than three years' working experience. Each participant was given four hours to study the Smac system before the case study commences. We also conducted two-hour training to them on the concept of the input validation.

The prototype tool had identified that there are 8 programs in Smacs that do not implement input validation, 18 decisions to accept input and 61 decisions to reject input received in the system. Hence, in this case study, we prepared 20 requirement change requests: 8 for adding new validation decisions for the programs in which input validation is not implemented, 8 for deleting existing validation decisions and another 4 for modifying existing validation decisions which involve both adding and deleting activities. The participants were randomly divided into two groups, each of which consists of one engineer and two students. All the participants in Group 2 were given three hours training on the proposed techniques and the usage of the prototype tool.

The participants in Group 1 were asked to modify the source code of the system according to the change requests purely based on their experience, while the ones in Group 2 were asked to do so using the proposed techniques with the

aid of the prototype tool. Each participant was required to do the assigned task independently in three steps:

- 1) Locate the source files that are to be modified;
- 2) Modify the source code;
- 3) Test the validity of the modifications.

We recorded the time each participant spent and also verified the modification of the code for the 20 change requests. The number of changes correctly made by each participant and the time spent are shown in Table 4.1.

Table 4.1 Results of the case study

Group	Participant	No. of Valid Changes	Time Spent (hours)
1	Engineer	20	7.5
	Student 1	15	6.2
	Student 2	13	6.5
2	Engineer	20	3.5
	Student 1	19	4.4
	Student 2	20	4.5

It is interesting to see that the engineer in Group 1 spent more time on the task than the students did, but he managed to make all the changes correctly. The engineer commented that he actually spent most of the time on analyzing the source code for change impact and also on testing the related functions in the system after modifications. On the other hand, the participants in Group 2 completed the task in a much shorter time. From the figures shown, it is clear that

the work quality and efficiency of the participants in Group 2 is much higher compared to those in Group 1. This results show that the proposed techniques is useful and effective in aiding the understanding and maintenance of input validation for both experienced and inexperienced software developers.

4.4 Conclusion

Maintenance of input validation requires the software developers to understand the implementation and perform the modification with minimum ripple effects. In this chapter, we have introduced a variant of control flow graph called validation flow graph for representing the input validation implemented in a program. Based on the input validation information recovered from the source code and the validation flow graph, we have proposed the techniques of VFG-guided slicing to aid the understanding of input validation. We have also proposed the effect-oriented decomposition slicing that decomposes a program with respect to the input and effect. It can be used to aid the maintenance of input validation.

We have conducted a case study to evaluate the effectiveness of the VFG-guided slicing and the effect-oriented decomposition slicing in aiding software maintenance activities on input validation. The results show that with the aid of the prototype tool, the efficiency and the quality of the work on modifying the input validation implemented in a program can be greatly improved. In the case study, the group of software engineer and the students who used the proposed method spent less time to complete the tasks and performed more valid changes compare to the group that performed the task manually.

Chapter 5

Testing of Input Validation

As input validation is vital to the robustness of a system, having an effective method for testing input validation is important for software quality assurance. Software testing techniques can be broadly classified into white-box testing and black-box testing. The former refers to the structural testing technique that designs test cases based on the information derived from source code. The latter refers to the functional testing technique that designs test cases based on the information obtained from the specification. In general, white-box and black-box testing are considered complementary to each other. Many researchers have emphasized that, to test software more accurately, it is required to cover both specification and code behaviors [63, 85].

Currently, most existing methods for testing input validation are specification-based. Hence, to test input validation more accurately and thoroughly, it is desirable to have a systematic way for code-based input validation testing and if

possible, a method that integrates code-based and specification-based input validation testing.

We have discussed in Chapter 3 the method for the automated recovery of input validation from program source code. Based on the information recovered, this chapter further explore the code behaviors of the input validation implemented in software programs and propose novel methods for testing input validation. We first propose a method that partitions the paths through the CFG of a program based on the code characteristics of input validation and a method to extract valid and invalid input conditions from program source code. Based on that, we then propose two test coverage criteria for code-based input validation testing and a method for the integrated white-box and black-box input validation testing. Controlled experiments have been conducted to evaluate the proposed methods.

The rest of this chapter is organized as follows. Section 5.1 introduces a theory on the input validation partition and the extraction of input conditions. Section 5.2 presents an algorithm to compute the input validation partition and extract valid and invalid input conditions from program source code. Section 5.3 discusses the methods of code-based input validation testing and its integration with specification-based testing. Section 5.4 describes the prototype tool and reports the experimental evaluation of the proposed approach. Finally, Section 5.5 concludes the chapter.

5.1 Extraction of Code Behavior on Input Validation

In Chapter 3 Section 3.1, we have discussed the effect node (definition 1), input node (definition 2) and validation node (property 1) in a control flow graph of a program. As the code behavior on input validation is path-based, based on the validation nodes identified in a program, we can partition the paths through the CFG of the program into different classes for code-based input validation testing. As theoretically the number of paths through a CFG is infinite, it is naturally to base on a set of paths that satisfies a path coverage criterion for the path partition. Basis path coverage [174] testing is the most commonly used technique for path-based testing. It identifies a subset of paths through the CFG of a program in which each predicate node in the program is evaluated to both true and false. Next, we shall establish our method based on the basis path coverage criterion.

Let u be an input node in the CFG of a program. Let S be a set of paths through the CFG that satisfies basis path coverage criterion. We then partition the paths in S in a way that paths are put into the same class if they satisfy the following two conditions:

- 1) They contain the same set of validation nodes of u .
- 2) They follow the same branch at each of the above-mentioned validation nodes.

Hence, the set of validation nodes and the branch traversed at each of these nodes form a path partition criterion. The resulting partition is called the **input validation partition** of u .

In many database applications, input data are submitted through a form that is comprised of many data items. A program that processes the form submitted can access the input data at any point in the program. If the number of data items submitted is large, the number of input nodes in the program is potentially large. Hence, the analysis of input validation implemented in those programs at the input node level may not be cost effective. In that case, the input validation partition can be performed **at program level** as follows:

Let S be a set of paths through the CFG of a program that satisfies basis path coverage criterion. We partition the paths in S in such a way that paths that pass through the same set of validation nodes of any input node and take the same branch at each of these nodes are put in the same class. The resulting partition is then called the input validation partition of the program.

Figure 5.1 shows a partial specification of a credit card validation program that reads details of credit card information and performs some basic checks on the validity of the credit card information. Figure 5.2 shows a possible implementation of the program and Figure 5.3 shows the CFG of the program. In this program, node 1, 4, 9 and 12 are four input nodes and node 15 is the only effect node where the transaction occurs. It is easy to verify that node 2, node 5 and node 7 are validation nodes of node 1, node 5 is also a validation node of node 4, node 10 is a validation node of node 9 and node 13 is a validation of node 12.

From the CFG shown in Figure 5.3, the set of paths that satisfies the basis path coverage criterion can be computed: $\{(entry, 1, 2, 3, exit), (entry, 1, 2, 4, 5, 6, exit),$

(entry, 1, 2, 4, 5, 7, 8, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15 exit)}.

Table 5.1 shows the paths in the input validation partition of each of the four input nodes in the program and the corresponding partition criterion. There are 4 classes in the input validation partition of node 1, three classes in the input validation partition of node 4, node 7 and node 10 respectively. Note that some classes in the input validation partition of an input node may not contain any validation nodes of that input node. This is indicated as “N.A.” under the column “Partition Criterion” in the table.

Figure 5.1 Partial specification of a credit card validation program


```
CCValidate ()
Begin
1  read CardNumber;
2  if (!LuhnMod10(CardNumber)), then
3      showError ("Invalid credit card number");
  else
4      read CardType;
5      if (GetType(CardNumber) != cardType), then
6          showError ("Card type does not match card number");
      else
7          if (IssuingCountry(CardNumber) != "SG"), then
8              showError ("Issuing country is invalid");
          else
9              read ExpiryDate;
10             if (ExpiryDate.before(CurrentDate)), then
11                 showError ("Card has expired");
            else
12                 read Name;
13                 if (Name.trim() == ""), then
14                     showError ("Name is not entered");
                    else
15                     CCTransact();
End
```

Figure 5.2 The pseudocode for the credit card validation program

Figure 5.3 The CFG of the credit card validation program

Table 5.1 Input validation partition

Input Node	Partition Criterion	Paths in the Input Validation Partition
node 1	2T	{ (entry, 1, 2, 3, exit)}
	2F, 5T	{ (entry, 1, 2, 4, 5, 6, exit)}
	2F, 5F, 7T	{ (entry, 1, 2, 4, 5, 7, 8, exit)}
	2F, 5F, 7F	{ (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15 exit)}
node 4	N.A.	{ (entry, 1, 2, 3, exit)}
	5T	{ (entry, 1, 2, 4, 5, 6, exit)}
	5F	{ (entry, 1, 2, 4, 5, 7, 8, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15 exit)}
node 9	N.A.	{ (entry, 1, 2, 3, exit), (entry, 1, 2, 4, 5, 6, exit), (entry, 1, 2, 4, 5, 7, 8, exit)}
	10T	{ (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit)}
	10F	{ (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15 exit)}
node 12	N.A.	{ (entry, 1, 2, 3, exit), (entry, 1, 2, 4, 5, 6, exit), (entry, 1, 2, 4, 5, 7, 8, exit)}, (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit)}
	13T	{ (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit)}
	13F	{ (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15, exit)}

In majority of cases, an input received from an input node is validated by checking it against some conditions using predicate nodes before any effect is raised. A condition that must be satisfied before the use of the input to raise an effect is a valid condition of the input. Conversely, a condition to deny the use of the input to raise an effect is an invalid condition of the input.

Let u be an input node in the CFG of a program. A path through the CFG along which the input received from u satisfies the valid condition of u is called an **acceptance path** of u ; otherwise, the path is called a **rejection path** of u .

Intuitively, a rejection path of u will deny the use of the input to raise an effect, thus will not lead to any effect node influenced by u . On the other hand, an acceptance path of u may or may not lead to an effect node influenced by u because whether the effect should be raised may also depend on the validity of other input data. For example, in the CFG shown in Figure 5.3, the path (entry, 1, 2, 4, 5, 7, 8, 9, exit) is a rejection path of node 7 along which the card is expired, and the path (entry, 1, 2, 4, 5, 7, 8, 10, 11, 12 exit) is an acceptance paths of node 1 along which the card number is correct.

Based on input validation partition, we shall present an invariant property that serves as a basis for the recovery of input conditions.

Property 1 – Valid and Invalid Class. Let u be an input node of a program. Let Ω_u be the input validation partition of u . For each class K in Ω_u , if there is a path in K that contains an effect node influenced by u , then all the paths in K are acceptance paths of u ; in this case, K is called a valid class of u . Otherwise, all the paths in K are rejection paths of u ; in this case, K is called an invalid class of u .

Let K be a class in Ω_u . From the definition of input validation partition, the conjunction of branch predicates at all the validation nodes of u that any path in K follows is identical. We denote it as C_K and call it the **input condition** of class K . Clearly, the checking of the input received from an input node u against a condition to decide whether an effect should be raised is carried out by the validation nodes of u . This directly implies that the input condition of a valid class is a valid condition of the input received from u . The input condition of an invalid class is an invalid condition of the input received from u . Next, we shall summarize it in the following invariant property.

Property 2 – Valid and Invalid Input Conditions. Let Ω_u be the input validation partition of an input node u . Then, $VC_u = \{\text{the input condition of } K \mid K \in \Omega_u \text{ and } K \text{ is a valid class of } u\}$ and $IC_u = \{\text{the input condition of } K \mid K \in \Omega_u \text{ and } K \text{ is an invalid class of } u\}$ are the sets of valid and invalid input conditions respectively of the input received from u , that are implemented in the program.

In the example shown in Figure 5.3, since node 15 is the only effect node in the program, the invalid classes and valid classes of each input node can be easily identified, and consequently the invalid and valid conditions of each input node can be computed. Table 5.2 gives the valid and invalid input conditions computed for each input node.

Table 5.2 Valid and invalid input conditions

Input Node	Input Validation Partition		Valid and Invalid Input Conditions	
node 1	Invalid class	{ (entry, 1, 2, 3, exit) }	Invalid Condition	! LuhnMod10(CardNumber)
		{ (entry, 1, 2, 4, 5, 6, exit) }		NOT(!LuhnMod10(CardNumber))&&GetType(CardNumber) != CardType
		{ (entry, 1, 2, 4, 5, 7, 8, exit) }		NOT(!LuhnMod10(CardNumber))&&NOT(GetType(CardNumber) != CardType) &&IssuingCountry(CardNumber) != "SG"
	Valid class	{ (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15 exit) }	Valid condition	NOT(!LuhnMod10(CardNumber))&&NOT(GetType(CardNumber) != CardType) &&NOT(IssuingCountry(CardNumber) != "SG")
node 4	N.A.	{ (entry, 1, 2, 3, exit) }	N.A.	N.A.
	Invalid class	{ (entry, 1, 2, 4, 5, 6, exit) }	Invalid condition	GetType(CardNumber) != CardType
	Valid class	{ (entry, 1, 2, 4, 5, 7, 8, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15 exit) }	Valid condition	NOT(GetType(CardNumber) != CardType)

node 9	N.A.	{ (entry, 1, 2, 3, exit), (entry, 1, 2, 4, 5, 6, exit), (entry, 1, 2, 4, 5, 7, 8, exit)}	N.A.	N.A.
	Invalid class	{ (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit)}	Invalid condition	<i>ExpiryDate</i> .before(<i>CurrentDate</i>)
	Valid class	{ (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15 exit)}	Valid condition	NOT(<i>ExpiryDate</i> .before(<i>CurrentDate</i>))
node 12	N.A.	{ (entry, 1, 2, 3, exit), (entry, 1, 2, 4, 5, 6, exit), (entry, 1, 2, 4, 5, 7, 8, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit)}	N.A.	N.A.
	Invalid class	{ (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit)}	Invalid condition	<i>Name</i> .trim() == ""
	Valid class	{ (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15, exit)}	Valid condition	NOT(<i>Name</i> .trim() == "")

5.2 A Method for the Extraction of Input Conditions

Figure 5.4 gives an algorithm for the computation of input validation partition and the extraction of valid and invalid input conditions for each input received in a program. For the analysis of input validation for programs that have a large number of input nodes, the algorithm can be easily modified for the computation of input validation partition and extraction of input conditions at program level.

The algorithm shown in Figure 5.4 comprises two steps. In Step 1, the function `Extract_IV_Info` discussed in Section 3.2 (shown in Figure 3.3) is used for the construction of the CFG of the program, and the recovery of the input nodes, the effect nodes, and the validation nodes for each input node in the program.

In Step 2, for each input node u in U , the input validation partition of u is computed and subsequently the set of valid and invalid input conditions of u are computed. First, a set $\mathcal{S}$ of paths is computed under the basis path coverage. Next, for each input node u in U , the input validation partition Ω_u of u is computed through identifying the sequence of validation nodes and the branches traversed in each path in $\mathcal{S}$. Subsequently, each class in Ω_u is examined to determine whether it is a valid or invalid class and the sets of valid or invalid input conditions are computed respectively.

Extract_Input_Conditions

Input: The source code of a program P .

Output: For each input node u in P , the input validation partition Ω_u of u , and the sets VC_u and IC_u of valid and invalid conditions of input received from u respectively.

Begin

Step 1. Call **Extract_IV_Info** (P) to obtain the following information:

- 1) The CFG of the program P
- 2) The set U of input nodes
- 3) The set F of effect nodes
- 4) The set D_u of validation nodes for each input node u in U .

Step 2. Compute the input validation partition Ω_u and the set VC_u/IC_u of valid/invalid input conditions for each input node u in U through the following sub-steps:

2.1 Compute a set S of paths under the basis path coverage.

2.2 For each input node u in U , compute the input validation partition Ω_u of u as follows.

- 1) For each path p in S ,
 - i) Identify the partition criterion Q that consists of the sequence of validation nodes of u in p and the branch traversed at each of these nodes.
 - ii) Include Q in set Ω_u if it has not been included yet.
 - iii) Add p into the class of paths that links to the criterion Q .
- 2) For each class K in Ω_u ,
 - i) if there is a path in K that contains an effect node influenced by u , Q is marked as a valid class of u . Extract the valid condition and include it in VC_u .
 - ii) Otherwise, Q is marked as an invalid class of u . Extract the invalid condition and include it in IC_u .

End

Figure 5.4 An algorithm for the extraction of input conditions

As input validation is a major feature in a system, the valid and invalid conditions of input should be obtainable from some form of system documentation. Therefore, the invalid and invalid input conditions extracted from code can be used to check against the corresponding conditions defined in the

specification for any discrepancy to be revealed. In addition, the extracted input conditions provide the testers with true implementation information that supplements the specification, and hence can be used to aid the test case selection in black-box testing

5.3 Techniques for the Testing of Input Validation

Based on the input validation partition and the extracted input conditions, we first propose two coverage criteria for white-box input validation testing. We then discuss the use the extracted input validation information in aiding black-box input validation testing.

5.3.1 White-box Input Validation Testing

Based on the input validation information recovered from the source code, we can test the code behaviors on input validation that are implemented in a program to ensure the accuracy of input received by the program. Next, we shall introduce the two coverage criteria for measuring the adequacy of input validation testing.

The input validation partition of an input node u divides the paths through the CFG of a program into different classes. Each class can be further classified as a valid class or an invalid class. Input received at u through executing any path in a valid class is accepted by the program, while input received at u through executing any path in a invalid class is rejected by the program. Thus, each valid class implements a way to accept input received at u , and each invalid class implements a way to reject input received at u . Based on this, we can measure the adequacy of

a test suite for testing the input validation implemented in a program at two levels:
input node level and program level.

Let u be an input node in the CFG of a program P . Let Ω_u be the input validation partition of u . Based on the input validation partition, a test suite is adequate for testing the input received from u if at least one path in each class in Ω_u is exercised. Hence, the **path-based input validation coverage** (path-based IVC) at **input node level**, $IVC(T, u, P)$, is defined as the percentage of the classes in Ω_u from which at least one path is exercised. That is:

$$IVC(T, u, P) = \frac{CE(T, u, P)}{C(T, u, P)} \times 100\%$$

where $C(T, u, P)$ is the total number of classes in Ω_u and $CE(T, u, P)$ is the number of classes in Ω_u from which at least a path is exercised by a test case in T .

Let Ω be the input validation partition of a program P . The **path-based input validation coverage** at **program level**, $IVC(T, P)$, is defined as the percentage of classes in Ω from which at least one path is exercised. That is:

$$IVC(T, P) = \frac{CE(T, P)}{C(T, P)} \times 100\%$$

where $C(T, P)$ is the total number of classes in Ω and $CE(T, P)$ is the number of classes in Ω from which at least one path is exercised by a test case in T .

As discussed earlier, in the credit card validation program shown in Figure 5.1, Figure 5.2, node 2, node 5 and node 7 are validation nodes of node 1, node 5 is

also a validation node of node 4, node 10 is a validation node of node 9 and node 13 is a validation of node 12. There are four classes in the input validation partition of node 1, three classes in the input validation partition of node 4, node 9 and node 12 respectively. To satisfy the path-based IVC at input node level, at least one path needs to be selected from each class in the partition. Table 5.3 gives the sample set of paths to be covered for testing input validation based on the path-based IVC at each input node. The last row shows the input validation partition of the program and a test suit that satisfies the path-based IVC at program level.

In general, a predicate node in a flow graph consists of a number of simple predicates composed by Boolean operators. In the path-based IVC, the complexity of the simple predicates that constitute a validation node is not considered. Therefore, for many safety-critical applications in which the combinations of the simple predicates in a validation node is of concern, the testing of input validation based on the path-based IVC alone may not be sufficient. Next, we shall present the condition-based input validation coverage criterion.

Multiple condition coverage is a testing criterion that requires every possible combination of conditions in a predicate node is exercised at least once [14]. Based on that, we propose the **condition-based input validation coverage** (condition-based IVC) for testing input validation implemented in a program: For each validation node in a program, every possible combination of the outcomes of the simple predicates in the node is exercised at least once.

Table 5.3 Sample test suites based on path-based IVC

Input Node	Input Validation Partition	Sample Set of Paths
node 1	{ (entry, 1, 2, 3, exit) }	{ (entry, 1, 2, 3, exit), (entry, 1, 2, 4, 5, 6, exit), (entry, 1, 2, 4, 5, 7, 8, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit) }
	{ (entry, 1, 2, 4, 5, 6, exit) }	
	{ (entry, 1, 2, 4, 5, 7, 8, exit) }	
	{ (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15 exit) }	
node 4	{ (entry, 1, 2, 3, exit) }	{ (entry, 1, 2, 3, exit), (entry, 1, 2, 4, 5, 6, exit), (entry, 1, 2, 4, 5, 7, 8, exit) }
	{ (entry, 1, 2, 4, 5, 6, exit) }	
	{ (entry, 1, 2, 4, 5, 7, 8, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15 exit) }	

node 9	{ (entry, 1, 2, 3, exit), (entry, 1, 2, 4, 5, 6, exit), (entry, 1, 2, 4, 5, 7, 8, exit) }	{(entry, 1, 2, 3, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit) }
	{ (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit) }	
	{ (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15 exit) }	
node 12	{ (entry, 1, 2, 3, exit), (entry, 1, 2, 4, 5, 6, exit), (entry, 1, 2, 4, 5, 7, 8, exit) }, (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit) }	{(entry, 1, 2, 3, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15, exit) }
	{ (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit) }	
	{ (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15, exit) }	
program level	{ (entry, 1, 2, 3, exit) }, { (entry, 1, 2, 4, 5, 6, exit) }, { (entry, 1, 2, 4, 5, 7, 8, exit) }, { (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit) }, { (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit) }, { (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15, exit) }	{(entry, 1, 2, 3, exit), (entry, 1, 2, 4, 5, 6, exit), (entry, 1, 2, 4, 5, 7, 8, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 11, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 14, exit), (entry, 1, 2, 4, 5, 7, 9, 10, 12, 13, 15, exit) }

In the credit card validation program shown in Figure 5.2, as each validation node in the program has exactly one simple predicate, two test cases are required to cover the conditions at each of the validation node. If there are n simple predicates in a validation node, the number of test cases required to satisfy the condition-based IVC for testing the node is 2^n . Hence, the condition-based IVC is only applicable to the programs in which the validation nodes have a relative small number of simple predicates. The condition-based IVC is expected to be used to complement the path-based IVC and the choice depends on the time and resources available.

The two input validation coverage criteria can be used to measure the adequacy of a test suite for testing input validation. Besides that, they can be used to guide the white-box test cases design manually or automatically using existing test case generation techniques. Next, we shall elaborate the methods for test case generation.

To test input validation in a program based on the path-based IVC, at least one path in each class in the input validation partition of each input node in the program need to be exercised. Since the valid and invalid input conditions have been extracted from the valid and invalid classes respectively, test cases can be manually selected for each of such conditions. Alternatively, a path from each class in the partition can be selected for generating test cases automatically using existing path-oriented test case generation techniques [56, 66, 67, 98]. If the path selected from a class in the partition is infeasible, another path in the class is selected. The process is repeated until a feasible path in the class is located or no other selection can be made.

As the input conditions have been extracted from the program source code, test cases can be easily designed to satisfy the condition-based IVC. For each validation node d in a program, the combinations of outcomes of the simple predicates in d are computed. Then test cases to cover each of the combinations can be generated manually. Test cases can also be generated automatically using existing test case generation techniques through some transformation. For instance, Gupta et al. propose a method to generate test data that exercises a selected branch [67]. A test case to cover a combination of outcomes of the simple predicates in a validation node d can be generated through the following steps:

- 1) Replace the condition defined in d with the conjunction of the following simple predicates: For each simple predicate s in d , the predicate ($s =$ its Boolean value set in the combination).
- 2) Identify the branch at d that will be exercised by the combination.
- 3) Generate a test case to pass through the branch using the method proposed by Gupta et al. [67].

5.3.2 The Integrated Approach

Currently, most existing methods for testing input validation are black-box testing methods. In black-box testing, test cases are developed through the study of the software specification in order to verify the behavior and functionality of the software. In principle, to derive a complete test suite from the specification for testing input validation, all possible combinations of input conditions should be considered. However, for a software system with ordinary complexity, the size of

the input domain can be very large; hence, exhaustive testing of all possible combinations of input conditions is costly and impractical with limited testing resources. Therefore, software testers often use their intuition and professional experiences to make certain assumptions on the implementation and select a set of test cases for testing. However, this approach relies on chances to determine the effectiveness of the resulting test suite [34, 74, 131].

Many black-box testing methods, such as the equivalence class testing and cause-effect graph, suggest that the number of test cases can be greatly reduced by apply the combinatorial testing only on the input conditions that may affect the effects of concern, rather than considering all possible input combinations [120, 131]. Equivalence class testing partitions the input space into equivalence classes according to the input conditions and elements in the same class are supposed to be handled identically in the program. Test cases are then designed by selecting at least one element from each equivalence class in the partition. Cause-effect testing is a strategy that considers combinations of causes of system behavior [131, 141]. It uses cause-effect graph and decision table as tools to design test cases, in which input conditions form the causes and actions form the effects. A test case is designed for each rule in the decision table. To conduct specification-based input validation testing, a tester needs to develop a test suite that is supposed to cover every aspect of the input validation implemented in the system. Often, when making choices on the equivalence partitions or deciding the cause-effect relations, software testers need to second-guess the likely implementation and functional manipulations in the program in order to develop more useful test cases [85, 131].

Furthermore, if the specification is outdated or contains ambiguities information, the testing process could be even more difficult.

As all the valid and invalid input conditions for each input received in a program have been extracted from the program source code, we can use such information to aid input validation testing by covering code behaviors on input validation in black-box testing. As a result, a possible combination of white-box and black-box input validation testing can be achieved. Next, we shall discuss the method towards the integrating of white-box and black-box input validation testing.

Firstly, we can use the extracted input conditions to verify against the corresponding conditions defined in the specification. This requires two steps:

- 1) Identify the valid and invalid input conditions defined in the specification.
- 2) Verify and reconcile the extracted input conditions with the ones identified from the specification.

Depending on the formality of the specification, the identification of the input conditions from the specification can be performed either manually or automatically. For example, if the specification is formally defined using predicate calculus or state transformations, the input conditions can be easily identified by partitioning the input domain into a finite number of input classes [152]. In practice, however, it is most likely that the specification is written in naturally languages. In that case, manual identification of input conditions from the specification is required. As input validation is a major feature in a large class of

software systems, the identification of the input conditions can be carried out by employing the techniques used in equivalence class testing or cause-effect testing.

The verification and reconciliation of the extracted input conditions with the ones identified from the specification requires the testers to have a good understanding of both the specification and the source code. This process can help to uncover the ambiguities and discrepancies between the code and the specification before executing any test cases. A valid or invalid input condition X extracted from the code matches with corresponding condition/conditions in the specification if one of the following three cases holds:

- 1) There exists a corresponding input condition Y in the specification such that $X \equiv Y$.
- 2) There exists n ($n \geq 1$) corresponding input conditions $Y_1, \dots, Y_n$ in the specification such that $X \equiv Y_1 \vee \dots \vee Y_n$.
- 3) There exists a corresponding input condition Y in the specification such that $X_1 \vee \dots \vee X \vee \dots \vee X_n \equiv Y$, where X_i ($1 \leq i \leq n$ and $n \geq 1$) are other input conditions recovered from code.

In the second case, a condition specified in the specification is a sub-condition of the one extracted from the code as the condition extracted could be a conjunction of branch predicates. Clearly, test cases that exercise X also exercise all the n conditions $Y_1, \dots, Y_n$ jointly. In the third case, the code is more refined than specification probably due to the subdivision of effect types or user interfaces for reporting invalid input.

Furthermore, the extracted input conditions can aid the testers in test case design and test case selection. Since the extracted information provide the testers essential implementation information on input validation, the number of times the testers need to make assumptions on the code implementations during test case design can be greatly reduced. The testers can use the extracted information to refine the test cases already designed or select test cases more effectively, and avoid unnecessary input conditions that are not implemented in the program. In addition, by using the techniques discussed in Section 5.3.1, white-box test cases can be designed to supplement the black-box test cases for testing input validation, and the coverage metric can be used to measure the adequacy of a test suite.

In the credit card validation program discussed earlier, we have extracted the valid and invalid input conditions for each input node in the program. Table 5.4(a) shows the input conditions extracted from the code, and Table 5.4(b) shows the input conditions identified from the specification discussed earlier in Figure 5.1. The input conditions from the specification are identified using the technique from equivalence class testing [131]. The letters and numbers in the table are unique identifiers.

In this example, the input conditions (a), (c) and (d) in Table 5.4(a) appear to be combinations of branch predicates. The input conditions identified from the specification, however, has no such combinations. Hence, this is the second case as discussed earlier where the input conditions identified from the specification is a sub-condition of the one extracted from the code.

Table 5.4 Reconciliation of input conditions

(a) Input conditions extracted from code

Input Node	Valid Input Conditions	Invalid Input Conditions
node 1	NOT(!LuhnMod10(CardNumber))&& NOT(GetType(CardNumber) != CardType)&& NOT(IssuingCountry(CardNumber) != "SG")	! LuhnMod10(CardNumber) (b)
		NOT(!LuhnMod10(CardNumber)) && GetType(CardNumber) != CardType (c)
		NOT(!LuhnMod10(CardNumber)) && NOT(GetType(CardNumber) != CardType) && IssuingCountry(CardNumber) != "SG" (d)
		GetType(CardNumber) != CardType (f)
node 4	NOT(GetType(CardNumber) != CardType) (e)	
node 9	NOT(ExpiryDate.before(CurrentDate)) (g)	ExpiryDate.before(CurrentDate) (h)
node 12	NOT(Name.trim() == "") (i)	Name.trim() == "" (i)

(b) Input conditions identified from the specification

Input	Valid Input Conditions	Invalid Input Conditions
Card Number	Passes the Luhn(Mod 10) check (1)	Fails to pass the check (2)
Card Type	A type match the Card Number (3)	A type does not match the number (4)
Expiry Date	Future date (5)	Past date (6)
Holder Name	String variable (7)	Empty string (8) String with space only (9)

The verification and reconciliation of the input conditions shown in Table 5.4 reveals three interesting issues. First of all, it is not difficult to detect that there is a condition in the implementation but not in the specification as shown in the condition (a) and (d) in Table 5.4(a). The condition `(IssuingCountry(Number) != "SG")` checks the issuing country of the credit card must be Singapore but this is not required in the specification shown in Figure 5.1. This problem could be either because the specification is not up-to-date, or the condition is an extraneous one introduced during program modification.

Secondly, the invalid conditions for card holder's name is classified in to two cases as shown in the condition (8) and (9) in Table 5.4(b). This is because the testers believe that the empty string and the string with space only would be handled differently in the program. A closer look of the condition (j) in Table 5.4(a) shows that the two cases are actually handled in the same way through using the `"trim()"` function which removes the white spaces from both ends of a string.

Lastly, a careful check on the input condition (g) in Table 5.4(a) reveals that if the expiry date is equal to the current date, it is treated as a valid date. However, in the specification, the expiry date is required to be future date which should be at least one day after the current date. This ambiguity may also need to be investigated and resolved.

5.4 Evaluation

5.4.1 The Prototype Tool

To validate the proposed methods, a prototype tool **IVTesting** has been developed that is extended from the tool **IVRecovery** discussed in Section 3.3 .

Figure 5.5 gives the architecture of the **IVTesting**.

Figure 5.5 The architecture of the prototype tool IVTesting

As discussed in Section 3.3, the IVRecovery builds the CFG of a program and extracts the input validation information including the input nodes, the effect nodes and the validation nodes from the program through data flow and control

flow analysis. The IVMiner implements the second step of the algorithm shown in Figure 5.4 for the computation of the input validation partition and input conditions. It contains two sub-components: the Path Partitioner and the Condition Extractor. Given a CFG of a program, the Path Partitioner first computes a basis set that consists of a set of linearly independent paths. The number of paths in a basis set is determined by cyclomatic complexity [122]. Figure 5.6 shows the algorithm for computing a basis set.

FindBasisSet

Input: 1) A CFG G of a program
2) A node n in the CFG

Output: A set S of linearly independent paths

Begin

if n is the exit node of the G , **then**

 add current path to the set S ;

else

if n has not been visited before, **then**

 mark n as visited;

 label a default outgoing edge;

 FindBasisSet(destination of the default outgoing edge)

for each of the other outgoing edges, **do**

 FindBasisSet (destination of the edge);

else

 FindBasisSet(destination of the default outgoing edge)

End

Figure 5.6 An algorithm for the computation of a basis set

The algorithm is a modified depth-first search algorithm [84] and has been proved by Poole [146]. The search starts from the entry node of a CFG and recursively descends down all possible outgoing edges. If the node visited has

never been visited before, a default outgoing edge is selected, and the current path is split into two paths that traverse each outgoing edge, going down the default one first. This process continues until the exit node is reached, and then a path is found and added to the basis set.

The Path Partitioner then partitions the basis set according to the validation nodes and branches traversed in the paths as follows: for each path in the basis set computed, the sequence of validation nodes of an input node u contained in the path and the branch traversed at each of these nodes are identified as a partition criterion. This criterion forms a class in the partition and is included in the partition if it has not been included yet. The path is then put into the corresponding class in the partition. Upon completion of the partition, the paths in each class K in Ω_u are examined. If there is a path in K contains an effect node influenced by u , K is classified as a valid class of u and the Condition Extractor computes the set of valid conditions of the input received from u ; otherwise, K is classified as an invalid class of u and the Condition Extractor computes the set of invalid conditions of the input received from u .

The computed input conditions can then be used by the software testers for the verification and reconciliation against the specification. Any inconsistency identified is due to either a design or coding error which should be investigated and rectified. Upon completion of the reconciliation, if there is any modification of the code, the IVRecovery and IVMiner shall be re-executed to obtain the updated input validation partitions and the refined sets of valid and invalid input conditions, which are subsequently used to guide test case generation.

The Path Tracer contains a component JTracer which is developed from the open source tool JSlice [86]. JSlice is a dynamic slicing tool for Java programs which collects and analyzes an execution trace of a program for a given input. As the size of an execution trace is usually huge, JSlice uses data compression algorithm to compactly represent the execution trace; hence, dynamic slicing can be performed over the compact representation. JSlice is built on top of the Kaffe Java virtual machine [88] and is compatible with JDK 1.4. JTracer is implemented by modifying JSlice as only the feature for tracking the execution traces is needed. JTracer generates the execution trace of a test case in terms of the source line number and the corresponding class name through tracing the program execution at run time. A sample program and the execution traces obtained are presented in Appendix Section 3. Through parsing the execution traces and the set of paths in the input validation partitions, the IV Coverage Analyzer computes the degree of path-based input validation coverage at input node level and program level for a given test suite.

5.4.2 Experimental Design

We have conducted a controlled experiment to evaluate the effectiveness of the proposed methods in testing input validation. The experiment was performed on two software systems. The first one is an open source system called Smacs which downloaded from Sourceforge [164]. Smacs is a web-based application for the management of the casual staff in an organization. It can store rosters, produce pay schedules and automate the roster generation based on staff requirements and staff availability. There are altogether 80 files in the system and the size of the

source code is 13462 LOC. The second system is called ResaleOnline which is developed by senior computer science students. ResaleOnline provides a platform for school students, staff and alumni to resale their used items. There are altogether 107 files in the system and the size of the source code is 11540. Before the commencement of the experiment, we have prepared for both systems the design specifications. In addition, each system has been thoroughly tested to ensure that the system behaves in accord with its specification.

In the experiment, we seeded into each system the faults that are commonly made during the implementation of input validation. The following seven types of faults were seeded into each system:

- 1) Boolean operator fault (BOF): The changes of *AND*, *OR* and *NOT* operators in a validation node.
- 2) Relational operator fault (ROF): The changes of less than, greater than, less or equal to and greater or equal to operators in a validation node.
- 3) Variable and constant fault (VCF): A variable or constant in a validation node is replaced by another variable or constant.
- 4) Predicate insertion fault (PIF): An extraneous clause is inserted into a validation node.
- 5) Predicate omission fault (POF): A clause is omitted from a validation node.

- 6) Condition insertion fault (CIF): An extraneous validation node is inserted in to the code.
- 7) Condition omission fault (COF): A validation node is omitted from the code.

For fault type 6), a trivial *if* condition was inserted right before an effect node such that it will not affect normal program execution. For fault type 7), we chose only to remove the validation node that has one branch leading to the exit node immediately after the checking failure; the node that prints system messages was also removed. We seeded 30 faults of the above types into each system under test in a way that no fault conceals another and all faults cause observable failures. The distribution of the faults seeded in each system is given in Table 5.5.

Table 5.5 The number of faults seeded

BOF	ROF	VCF	PIF	POF	CIF	COF
6	6	6	3	3	3	3

From our analysis of functional testing techniques and further discussion with the software engineers in the industry, we come to the conclusion that the most frequently used method to test input validation is equivalence class testing supplemented with boundary value analysis. Hence, in the experiment, we compare the proposed methods with this method. The objective of the experiments is to uncover the faults seeded in each system under test using one of the following methods:

- 1) Perform equivalence class testing supplemented with boundary value analysis.
- 2) Perform white-box testing to satisfy the path-based input validation coverage (IVC).
- 3) Perform white-box testing to satisfy both the path-based IVC and condition-based IVC.
- 4) Perform integrated white-box and black-box input validation testing.

Since black-box and white-box testing techniques have been taught in the software engineering subject at our school, 12 students enrolled in the software engineering subject were selected to participate in the experiment. All the students have no prior knowledge of the system under test and all of them are familiar with Java programming. The students were evenly divided into four groups to conduct the experiment using the above four testing methods respectively. Before commencement of the experiment, the students were given two-hour training on input validation testing, the proposed techniques to be used in the experiment and the usage of the prototype tool.

The experiment procedures for using method 1, 2 and 3 respectively are based on the experiments designed for functional and structural testing discussed by the Basili and Selby [10]. There are no functional testing tools or test case generation tools available in the experiment. In method 1, the students were provided with the system specification and executable version of the system under test. They

were required to design test cases from the specification, execute the test cases, verify the testing results and record the failures observed.

In method 2 or method 3, the students were provided with the source code and the access to the prototype tool which extracts the input conditions and reports the attained level of input validation coverage. They were informed about the existence of infeasible paths and required to construct test cases based on the input conditions to achieve a coverage level as high as possible. They were then given the specification to verify the validity of test results and record the failures observed.

In method 4, the students were first asked to produce a test specification by identifying the valid and invalid input conditions purely from the specification. Upon completion, they were then given the source code and the input conditions computed by the prototype tool for the verification against the test specification. Any inconsistencies or ambiguities discovered need to be recorded. Finally, they were required to design test cases based on all the information available, execute the test cases and record the failure observed.

Five 4-hour sessions were organized for each system under test. The whole process was monitored by the author and we had made sure that each student conducted the experiment independently.

5.4.3 Threats to Validity

The experiment has two independent variables: the testing technique and the type of software. The dependent variable to be examined is the number of faults observed.

Threats to validity for this experiment are similar to those for earlier studies [89, 181]. Threats to external validity are factors which prevent the generalization of the experiment results to actual software engineering practice. These threats include the testers, programs, faults and fault densities which can only be addressed by repeated studies using different testers, programs, faults and fault densities.

Threats to internal validity are unknown factors exerting control over the dependent variables that include the following aspects:

- Instrumentation effects caused by differences in the programs under test and the type of faults seeded.
- Selection effects due to variability of the students. Basically there is no way to guarantee that the students participated in the experiment have exactly the same knowledge and capabilities.
- Maturation effects arose when the experiment proceeds as the students learn at different speeds and depths.
- Observer effects due to the experimental conditions. To minimize this effect, the experiment results are independently verified by two research staff.

5.4.4 Results and Discussions

For the students in Group 1 who performed the equivalence class testing supplemented with boundary value analysis, we verified the test suite designed and the number of faults correctly identified by each student. In addition, the prototype tool tracked the execution of each test suite and computed the path-based input validation coverage. A summary of the results is shown in Table 5.6.

Table 5.6 The percentage of faults observed using method 1

Group 1 Student	Smacs		ResaleOnline	
	%Faults	IVC	%Faults	IVC
1	53%	46%	47%	38%
2	60%	62%	59%	57%
3	57%	54%	53%	50%

The column “%Faults” gives the percentage of faults observed by each student out of the 30 faults seeded in the system. The column “IVC” gives path-based input validation coverage at the program level for each test suite. From the figures shown, we observed that none of the test suites manages to detect all the faults seeded in the systems. Since all the faults seeded relate to input validation, not surprisingly, a test suite that has higher with higher input validation coverage tends to have a better ability in detecting those faults, though no test suite gives 100% input validation coverage due to the existence of infeasible paths.. Clearly, the testing of input validation is a non-trivial task, and the input validation coverage criteria can be very useful to measure the adequacy and thoroughness of the input validation testing. We examined the types of the faults detected and found that

most of the faults detected fell under the category ROF, VCF, POF and COF. The method almost detected none of the faults of the type PIF and CIF.

For the testing using method 2, the students in Group 2 performed white-box testing based on the path-based input validation coverage criterion. The prototype tool first analyzed the program source code and computed the input validation partition of each input received in the system. Overall, there are 79 classes of paths computed from Smacs, of which 18 are valid classes and 61 are invalid classes, and there are 103 classes of paths computed from ResaleOnline, of which 29 are valid classes and 74 are invalid classes. Based on the input validation partitions computed and the valid and invalid input conditions extracted from the source code, the students in Group 2 designed test cases to cover each class in the input validation partitions in each system. The testing results are shown in Table 5.7. The figures show the improvement of accuracy achieved by method 2 in detecting the faults; however, there are still quite a number of faults undetected in each system. We observed that many of the undetected faults were those Boolean operator faults (BOF) hidden in the validation nodes that consist of more than one simple predicates.

Table 5.7 The percentage of faults observed using method 2

Group 2 Student	%Faults Observed	
	Smacs	ResaleOnline
1	63%	67%
2	73%	70%
3	67%	67%

For the testing using method 3, the students in Group 3 performed the white-box testing based on both the path-based and the condition-based input validation criteria. The prototype tool analyzed that there are altogether 64 validation nodes in the system Smacs and 54 validation nodes in the system ResaleOnline. A summary of the complexity of the predicate at those validation nodes is shown in Table 5.8.

Table 5.8 Complexity of the predicate at validation nodes

# Simple Predicates	# Validation Nodes	
	Smacs	ResaleOnline
1	33	41
2	15	12
3	3	1
4	3	0
Total	64	54

The row “#Simple Predicate” gives the number of the simple predicates in a validation node which ranges from 1 to 4 in the two systems. As the total number of validation nodes in each system is not too large, and the number of the simple predicates in a validation node is relatively small, the size of the test suite designed by each student using method 3 is manageable. The percentage of the faults observed by each student is summarized in Table 5.9. As expected, by adding the condition-based input validation coverage criterion, the student managed to design test cases that detected a number of the Boolean operator faults.

Table 5.9 The percentage of faults observed using method 3

Group 3 Student	%Faults Observed	
	Smacs	ResaleOnline
1	80%	80%
2	83%	77%
3	77%	80%

We further observed that method 3 is not very capable in detecting faults of the type POF and COF. This is a common issue of white-box testing as white-box testing derives test cases from the code. If the conditions are not implemented in the code, there would be no test cases designed to exercise them; therefore, those omitted conditions can be easily overlooked by the testers. On the other hand, as shown in method 1, faults of the type PIF and CIF are usually difficult to be detected using black-box testing as the program implements the behaviors that are not in the specification. Since black-box testing derived test cases from the specification, it is very likely that those extraneous conditions in the code would not be exercised at all.

In method 4, an integrated white-box and black-box input validation testing was performed and Table 5.10 shows the results of the testing. The students in Group 4 managed to uncover most of the faults seeded in the two systems.

It is clear that, at least in this experiment, method 2 and 3 performed better than method 1 and method 3 demonstrated significant improvement of the fault detection capability over both systems. Method 2 is applicable and effective because it focused on a subset of paths in a program that has been identified as

major and essential to implementation of the input validation. Method 3 delivered even better performance because it explored the combinations of the conditions in each validation node. Method 4 integrated the white-box input validation information into the black-box testing; hence, the performance was greatly improved.

Table 5.10 The percentage of faults observed using method 4

Group 4 Student	%Faults Observed	
	Smacs	ResaleOnline
1	93%	90%
2	87%	90%
3	97%	93%

During the experiment, all the students were aware of the time limit and managed to complete the assigned tasks in time. On average, method 4 took longer time than the other methods. This is expected as the verification and reconciliation of the code information with the specification took extra time. However, it is worth the effort as the process helped the students to uncover quite a number of faults before any test cases were executed. In particular, faults such as Boolean and relational operator fault, omission or insertion of condition in a predicate node can be effectively spotted through the reconciliation process. Usually, detection of such faults requires boundary value testing or multiple condition coverage.

We also observed that though there might not be a direct match between the input conditions extracted from the code with the ones defined in the

specification, the understanding of the input conditions was not difficult for the students. The corresponding output types and the names of the variables gave them clue. The strategy was to first identify the correspondence in terms of the modules or functions implemented, and then find out the one-to-one correspondence for each prime input condition extracted.

We further observed that during the test case design in method 4, besides the test cases designed from the test specification, students also designed additional test cases based on the extracted input conditions to supplement the black-box test cases. In this way, both the specification and the code characteristics of input validation are tested by the test suite designed. As a result, significant improvement of the accuracy and thoroughness of the testing can be achieved.

5.5 Conclusion

In this chapter, we have proposed the input validation partition method that partitions the paths through the CFG of a program into different classes according to code characteristics of input validation. Each class in the partition can be classified into valid classes and invalid classes, from which valid and invalid input conditions can be extracted from the source code. Based on input validation partition, we have proposed two test coverage criteria for white-box input validation testing. The path-based IVC requires at least one path in each class of the input validation partition needs to be exercised. Condition-based IVC requires every combination of conditions in each validation node need to be exercised at least once. The two coverage criteria can also be used to measure the adequacy of any test suite for testing input validation.

We have also proposed the techniques for the integrated white-box and black-box input validation testing. The extracted input conditions can be used to verify and reconcile against the input conditions defined in the specification for any inconsistencies and ambiguities to be revealed. In addition, the extracted input conditions can aid the test case selection in black-box testing and also aid the design of white-box test cases that supplement the black-box testing. Hence, the resulting test suite covers both the specification and the code characteristics of input validation.

Our evaluation results show that testing input validation is not an easy task and the proposed coverage criteria can serve as measurements of the adequacy of testing conducted. In the experiment, the integrated white-box and black-box input validation testing can effectively improve the accuracy and thoroughness of input validation testing. The reconciliation process can help to detect quite a number of faults before execution of any test cases. The test suite designed covers both the specification and the code characteristics of input validation; hence, significantly improves its fault detection capability.

Chapter 6

Related Work

We have proposed in this thesis the approaches for the recovery, maintenance and testing of input validation. Those approaches are developed from techniques of program analysis, program slicing and software testing. This chapter discusses and compares the proposed approaches with the related work in the areas of locating feature in program source code, using slicing for program maintenance and input validation testing.

6.1 Feature Recovery

We have proposed in this thesis an approach that locates and recovers input validation from program source code. Software programs usually have many functional features. Various program analysis techniques have been developed for locating features in program source code. Locating feature is a technique of design recovery. The technique identifies the most relevant code or design abstractions for a specific feature from a combination of program source code, domain

knowledge, and professional experiences [21, 36]. Chen and Rajlich [33] propose a semi-automatic method for locating features based on abstract system dependency graph which describes detailed dependencies in a system at the level of global declarations. The method aids programmers to navigate the graph and search for a feature's implementation. It requires that the programmers must be experienced and have pre-knowledge of the system. Eisenbarth et al. [55] propose a semi-automated method that combines static analysis, dynamic analysis and user concept analysis for locating features in source code. The method is capable to identify the code segment that specifically implements a feature, but it relies on domain-specific expertise.

There are also many approaches that use graph matching techniques to locate features in source code [49, 95, 96, 158]. In those approaches, the structure of the source code is matched with predefined patterns or abstract models of the feature, thus the construction of the patterns or the models for representing the feature is needed.

Wilde and Scully [178] introduce the software reconnaissance method that uses test case execution to locate features. The method is based on a comparison of execution traces of different test cases of a particular feature. Another approach that based on dynamic information is proposed by Wong et al. [180]. The method analyzes execution slices of test cases that implement a particular functionality. Simply stated, the method locates the parts of code that are unique to a feature by identifying the code that is executed by any test cases that show functionality of a feature but not by any that does not show the functionality of the feature. The

effectiveness of the approach depends on the selection of test cases. Poorly selected test cases will lead to inaccurate results.

Program slicing has also been used for identification of reusable functions [37, 106]. Lanubile et al. [106] introduce a technique of transform-slice for recovery of reusable functions from ill-structured programs. In their approach, a set of input variables, a set of output variables and an initial statement of the searched function need to be specified by users and added to the slicing criterion. Tan et al. [167] introduces a method for extracting functionalities from source code based on static slicing techniques that are augmented with input/output statements. The method is developed from the hypothesis that a program delivers its functionalities through its output statements. A specification driven slicing technique is proposed by Cimitile et al. [37] for extracting code fragments that implement functions. The method identifies the pre-condition and the post-condition of a function from specifications, and the slicing criterion is then defined for searching the function from program source code.

So far, there is no approach that uses any of the program analysis techniques for the recovery of the input validation has been proposed yet. One characteristic of our approach is that the proposed approach incorporates the empirical properties into program analysis for feature recovery. The method is an evolving process in nature which involves construction of hypotheses, deduction of consequences, refinement of hypotheses through observations, and statistical validation. The process is very close to the experimental program analysis procedure discussed by Ruthruff [157], in which multiple experiments are conducted to search for the properties of a program under analysis. The

experiments are conducted in sequence such that the design of later experiments is changed or refined by leveraging the findings from previous experiments. The empirical methods are commonly adopted in medicine research, and recently, more and more software engineering researchers have started to explore into the use of empirical methods to solve software engineering problems [8, 144, 157, 162].

6.2 Using Slicing for Program Maintenance

Software maintainers are often faced with the problems of understanding existing software and making changes without affecting the behavior of other parts of the program. In this thesis, we have proposed the techniques of VFG-guided program slicing and effect-oriented decomposition slicing to aid the understanding and maintenance of input validation in software systems. The ideas are based on the techniques of program slicing, chopping and the decomposition slicing.

Chopping is first introduced by Jackson and Rollins [83] in which dependence chains from a source to a sink are identified. Compared to program slicing, chopping provides a more focused approach to reveal the transitive dependency from one statement to another throughout the program. Jackson and Rollins [83] impose a restriction on chopping that the source and sink must be in the same procedure. Reps and Rosay [150] extends the form of chopping into an unrestricted interprocedural chopping. Krinke [104] presents an evaluation of various slicing and chopping algorithms. He then proposes an approach that uses barriers to further reduce the size of chopping for program understanding [105].

Chopping basically uses intersection of a backward slice and a forward slice. Variants of such set operations on slices have been proposed. Lyle [116] introduces program dicing that for finding the differences between the static slices of two variables. It is a subtraction of two slices and is often used to identify the statements likely to contain bugs when computation on some variables fails while on other variables successes. Beszedes et al. [19] introduce union slices which are the union of dynamic slices for many test cases and suggest using a combination of static and union slices for program maintenance. However, set operations on slices need to be used with care as it has been established that union of two static slices may not produce a valid slice [47], and union of two dynamic slices is also not necessarily a valid slice [72].

The decomposition slicing technique is introduced by Gallagher et al. [62] for decomposing a program into a set of components, each of which maintain the behavior of part of the original program. The technique provides partial view of a program with the unnecessary statements removed and dependent statements restricted from modification. Based on the technique of decomposition slicing, Gallagher [61] then build a tool for the maintenance of C programs. The tool automates and visualizes impact analysis by displaying the relationships among a program's decomposition slices and allows the software maintainers to visually comprehend the relationships and the impact of proposed changes. The use of decomposition slices in understanding the internal structures of Web Applications has also been discussed [151].

Recently, Tonella [171] argues that the interference dependency between two variables could be missing in decomposition slices. He extends the decomposition

slice graph with additional nodes that present weak interference between computations and calls it a concept lattice of decomposition slices. In this new presentation, a binary relationship exists between a variable and a statement if the statement belongs to the decomposition slice of the variable. Consequently, the problem of impact analysis is transformed into the problem of identifying a set of reachable nodes in the concept lattice. As all the dependencies between computations are explicitly represented, this approach can be more accurate and useful compared to the original decomposition slices for software maintenance.

6.3 Input Validation Testing

The earliest work related to input validation testing mostly focus on the automated generation of programs to test compilers [11, 12, 51, 121, 147]. For example, Purdom [147] proposes an algorithm to produce a small set of short sentences from a context free grammar for testing parsing programming and for debugging grammars. Duncan and Hutchison [51] proposes a method that uses attributed grammars to generate both input and output for testing system specifications or implementations. Maurer [121] discusses a method that uses enhanced context-free grammars for generating test cases and improving functional testing of very-large-scale integrated circuits. Those approaches do not use static program analysis and they were usually developed for a particular domain of application.

Subsequently, Beizer [14] proposes an approach for syntax testing that uses graph to specify user command. Test cases are then generated to cover the graph using coverage techniques. Marick [120] also proposes an approach to syntax testing based on some informal guidelines. However, these approaches cover only

the syntax of input which is only part of input validation. Both approaches suggest general ideas on generating test cases in syntax testing, but do not provide specific and formal methods or tools to implement the idea.

Recently, techniques that are more related to input validation have been proposed by Hayes and Offutt [78, 79, 137]. They propose the input validation analysis and testing (IVAT) technique that statically analyzes input command syntax as defined in a structured specification written in a predefined format and generates test cases for input validation testing [78, 79]. Their approach goes beyond Beizer and Marick's work by addressing the notion of searching for interface problems early in the lifecycle and then using any problems found to generate test cases for later testing. Motivated by the importance of web applications security, Offutt et al. [137] then propose the bypass testing, which is a variation of input validation testing, for testing the robustness of web applications. Through bypassing client-side checking, test cases with invalid input are created to test input validation implemented in the applications.

The proposed input validation testing techniques in this thesis share the objective of testing input validation in data-intensive systems with IVAT. However, there is a theoretical difference between the two approaches. The IVAT and bypass testing are specification-based. They generate test cases from the specification alone and do not require any information from program source code. The testing techniques proposed in this thesis are code-based. The techniques partition the paths in a CFG of a program based on the code characteristics of input validation and extract valid and invalid input conditions from code for generating test cases.

We have also proposed two input validation testing criteria for white-box input validation testing. Test coverage is one of the key measures to ensure the test quality of software systems [177, 187]. Currently, most of the white-box testing techniques are based on the structure and logic of code [14, 131, 187]. For example, branch coverage and basis path coverage are based on the control flow of a program [14, 131]. Data flow coverage is based on the definition-use association of program variables, which includes a variety of data flow artifacts such as c-use, p-use and du-paths [14, 58, 131]. Recently, many coverage criteria have been proposed for testing systems from different aspects. Memon et al. [124] proposes an event-based coverage criteria for GUI events and their interactions, in which the GUI components are represented by an event-flow graph. Based on this graph, intra-component coverage criteria are used to evaluate the adequacy of tests on events within a component and inter-component criteria are used to assess the adequacy of test sequences across components. Rountev et al. [156] present a family of coverage criteria for testing the object interactions based on reversed-engineered sequence diagrams, which are extensions of traditional coverage criteria through the use the sequences of messages in the diagrams.

In opposing to the existing coverage criteria, the input validation coverage criteria proposed in this thesis are based on a functional feature implemented a software system. In a large class of systems especially data-intensive systems, input validation reflects the key functionality of those systems. Hence, the proposed technique serves as a fault-based model that targets a key functionality – input validation – in the systems, in which test cases are generated to demonstrate the

absence of input validation faults [127]. Currently, there is no such code-based technique exists.

Our proposed testing techniques share with the structural testing techniques on the generation of test cases that satisfy a coverage criterion. It shares with the functional testing techniques on the use of some kind of model to conduct testing. Currently, most of the existing testing techniques use either structural or functional testing strategies that are solely based on code or specification respectively (e.g. [27-30, 40, 109, 118, 119, 130, 139, 168]). As both white-box and black-box testing have their own strength and limitations, researchers have pointed that neither functional nor structural testing method alone can ensure the full correctness of software systems [63, 85].

The advantages of using both black-box and white-box information in software testing have been well discussed in the literature [20, 31, 32, 35, 152, 184]. Richardson and Clake [152] introduces a partition analysis technique that applies symbolic evaluation to translate both the specification and the implementation into a functional representation. The technique then integrates testing and verification through dividing a procedure's domain into sub-domains such that elements of each sub-domain are treated uniformly by the specification and the program.

Chen et al. [31, 32] develop a formal method for testing object-oriented programs at the class and cluster levels. The method selects test cases using a black-box technique based on algebraic specifications, and then uses white-box technique to determine whether the objects resulting from executing such test

cases are observationally equivalent. Chen et al. [35, 184] also propose a path-based strategy to reduce the size of black-box test suite. In their approach, a black-box test suite covers all compatible combinations of input conditions; hence, the test suite contains a large number of test cases and is impractical to be tested in its entirety. The authors propose that the black-box test cases can be partitioned into different classes according to the paths they follow. A limited number of test cases then can be selected from each class, and the size of the test suite can be reduced.

Baydeda et al. [20] proposes a Class Specification and Implementation Graph (CSIG) that contains both black-box and white-box information. In a CSIG, each method of a class is represented by two control flow graphs; one is constructed from the implementation through control flow analysis, and the other one is constructed based on specification through building prototypes that correspond to the specification. Based on the CSIG, test cases can then be generated using structural testing techniques to cover both specification and implementation characteristics.

In our proposed approach for input validation testing, the valid and invalid input conditions recovered from program source are verified and reconciled against the corresponding conditions identified from the specification. Another method that can be used to detect faults in software specification or source code is software inspection. Techniques of software inspection has been extensively studied and discussed in the literature [7, 18, 52, 53, 117, 159].

Basili et al. [7] propose a perspective-based reading technique that is used to review software requirements specification. The technique verifies the quality of

the specification by requiring each reviewer to take the perspective of a specific stakeholder of the document which helps ensure the quality of the specification.

MacDonald et al. [117] describes the facilities of the object-oriented paradigm and the issues that may raise when inspecting object-oriented code. They suggest that the improved program documentations and the use of various programming techniques could assist in making object-oriented code easier to inspect.

Later, Dunsmore et al. [52, 53] introduce three techniques for effective and efficient inspection of object-oriented program source code: one based on a checklist which includes a list of check items to follow, one based on abstractions in which the functionality of each part is abstracted from the code, and the other one based on scenarios which are the sequences of events to trace through the code.

Our proposed approach focuses on a specific feature – input validation. As the valid and invalid input conditions are automatically extracted from program source code, the reconciliation and verification of the extracted information against the input conditions defined in the specification can be performed in a more effective way.

Chapter 7

Conclusion

7.1 Conclusion

Input validation plays an important role in a large class of software systems. It is the most effective means to enforce the accuracy of the input submitted to a system and vital for the robustness of the system. Currently, the processes for the recovery and the maintenance of input validation are tedious and time-consuming; to perform a thorough testing of input validation is also difficult. This thesis has presented the approaches that address the recovery, maintenance and testing of input validation in software systems. The proposed approaches are established based on the techniques of program analysis, program slicing and software testing augmented with empirical properties. The work has been published in [110-114].

We have developed a theory for characterizing the input validation implemented in software systems and proposed a method for the automated recovery of input validation from program source code. We have established the empirical properties to check whether a program fails to implement input

validation and to identify all the decisions implemented in a program for accepting or rejecting input. All the empirical properties have been statistically validated.

We have conducted two case studies to examine the importance of input validation and to evaluate the feasibility of the proposed method for the recovery of input validation. The first case study suggests that improperly implemented input validation could be the cause of many problems such as unexpected runtime exceptions, security vulnerability, or violation of database integrity. Those problems seriously deteriorate the quality of the software. In the second case study, the proposed method effectively recovered input validation information from the source code in less than a minute. In contrast, the process of manual recovery of input validation took very long time, yet complete and accurate information on input validation cannot be fully recovered.

Maintenance of input validation requires the software developers to first understand the implementation of input validation. We have introduced the validation flow graph for representing the overall structure of the input validation implemented in a program. Based on the input validation information recovered from the source code and the validation flow graph, we have proposed the techniques of VFG-guided slicing to aid the understanding of input validation.

The method of VFG-guided slicing contains four slicing guidelines that help the developers to identify the statement of interest and suggest appropriate slicing algorithm, thereby facilitate the understanding of the input validation implemented in a program. We have also extended the technique of decomposition slicing and proposed the effect-oriented decomposition slicing which decomposes a program with respect to the implementation of input validation. The effect-oriented

decomposition slice yields only the portion of the program that relates to input validation and minimizes the ripple effects during the maintenance of input validation.

We have conducted a case study to evaluate the effectiveness of the VFG-guided slicing and the effect-oriented decomposition slicing. In the case study, two groups of participant were required to make changes to the input validation implemented in a system, in which the proposed techniques were used by one of the groups. The results show that with use of the proposed techniques, much less time was spent on the assigned tasks, and the quality of the work done by the participants was greatly improved.

We have proposed novel testing methods to improve the thoroughness and accuracy of input validation testing. We have presented the input validation partition method that partitions the paths through the control flow graph of a program into different classes according to the code characteristics of input validation. We have also presented the method to extract valid and invalid input conditions from the source code. Based on the input validation partition and extracted input conditions, we have proposed the path-based and condition-based input validation coverage criteria to guide white-box input validation testing. The two coverage criteria can also be used to measure the adequacy of any test suite for testing input validation. We have also proposed the techniques for the integrated white-box and black-box input validation testing. The extracted input conditions can be used to verify and reconcile against the input conditions defined in the specification for any inconsistencies and ambiguities to be revealed. They can also be used to aid the black-box test case selection and the design of white-box test

cases. Hence, the resulting test suite covers both the specification and the code characteristics of input validation, and a more accurate and thorough testing of input validation can be achieved.

We have conducted a controlled experiment to evaluate the proposed testing techniques in which various faults on the implementation of input validation were seeded into the systems, and the participants were required to uncover the faults seeded using four different testing methods. The results show that the testing of input validation is not an easy task and the proposed coverage criteria can serve as measurements of the adequacy of input validation testing. In the experiment, the white-box testing based on path-based input validation coverage criterion performed better than the functional testing method. The use of condition-based input validation coverage criterion together with path-based coverage criterion delivered even better results. The integrated white-box and black-box input validation testing method had the highest fault detection capabilities and significantly improved the accuracy and thoroughness of input validation testing.

7.2 Recommendations and Future Research

The research results presented in this thesis are encouraging and promising. We have received positive feedback from research peers and domain experts. The research leads to many future possibilities.

First of all, the work presented in this thesis suggests several practices to the software engineering community. Input validation should not be overlooked, and should be properly designed, implemented and verified during the software

development process. Software project managers should recognize the importance of input validation and put it as a very important target in design and verification activities. To software developers, emphasis must be put on specifying and designing robust interfaces for handling input from different sources. To software testers, comprehensive test cases must be designed and carefully selected to cover various input conditions. In addition, project managers may require the developers to provide as many details as possible in the interface specification, facilitating the verification and reconciliation of the information recovered from the code against the ones defined in the specification.

Like any other empirical research [89, 181], the research presented in this thesis is not without limitations. Threats to external validity of this research include the testers, programs, faults and fault densities which prevent the generalization of the experiment results to actual software engineering practice. Threats to internal validity of this research include instrumentation effects caused by differences in the programs under test and the type of faults seeded, selection effects and maturation effects due to variability of the participants, and observer effects due to the experimental conditions. With limited time and resources, much effort has been made to make sure that both types of threats to validity are adequately addressed. In the future, to minimize the threats to validity further, it is recommended to conduct more studies in a systematic way using programs written in different program languages, different types of faults and fault densities, and having the participants with different levels of experiences. This could also lead to the discovery of new empirical properties which might be language-specific or fault-specific.

The prototype tools presented in this thesis are built on third party libraries and components and currently support programs written in Java programming language. The implementations can be further refined and enhanced to achieve higher precision and better user interface. Universal tools that support multiple programming languages can also be developed. This requires the design of an intermediate representation for representing the control flow graph of a program that might be written in any programming language. Algorithms can then be implemented based on the new representation. Such tools would be very useful for analyzing large-scaled real-world software applications that are often implemented in more than one programming language.

A natural extension of the work is to develop approaches for code-based equivalence classes testing. Equivalence class testing [131] is a black-box testing technique that develops a set of test cases through partitioning the input domain of a program into classes of data. As input validation is a specialized feature that handles input submitted to a system, the approach for the recovery of input validation can be generalized and extended to extract equivalence classes from the source code based on the input and output interactions in the system. Subsequently, approaches for code-based equivalence class testing can be developed to supplement black-box equivalence class testing. Covering code behavior in black-box testing has the advantage of achieving test accuracy with optimized testing effort.

Furthermore, the work presented in this thesis is established through empirical studies and focuses on a specific functional feature implemented in a program. This empirical-based feature-oriented paradigm might open a new avenue for

solving software engineering problems. Exploration on other software features would be a promising future research area.

Appendix

1. Source code for building a CFG of a program using JABA:

```

...
// Build the CFG for each method in a Java program
public class CFGBuilder extends JABADriver{
...
    // perform analysis operations after the program has
    // been parsed.
    protected void run(){
        if (program == null){
            usage();
            throw (new IllegalArgumentException());
        }
        System.out.println("Symbol table loaded...");

        //Get all the classes except the library classes
        Class[] classes = program.getClasses();
        for (int i = 0; i < classes.length; i++){

            //Get all the methods in a class
            Method[] methods = classes[i].getMethods();
            for (int j = 0; j < methods.length; j++){
                String methodName = methods[j].getName();

                //Get the CFG of the method
                CFG cfg = Factory.getCFG(methods[j]);

                //Output the CFG into XML and Dot format
                if (isNotNull(cfg)){
                    System.out.println("Constructing cfg...");
                    Utils.writeXMLFile(cfg, methodName+"-cfg.xml");
                    Utils.writeDotFile(cfg, methodName+"-cfg.dot");
                }
            }
        }
        System.out.println("\nAnalysis completed successfully");
    }
...

```

2. Source code for implementing the backward slicing algorithm based on JABA:

```

//Backward slicing algorithm to get all the relevant nodes
//of the rootNode
public ArrayList getRelNodes(Node rootNode) {
    HashSet relVars = new HashSet(); //Relevant variables
    ArrayList relNodes = new ArrayList(); //Relevant nodes

    relNodes.add(rootNode);

    //Find the index of the rootNode
    int index = 0;
    Node[] nodes = getNodes();
    for (int i = nodes.length - 1; i >= 0; i--) {
        if (nodes[i].equals(rootNode)) {
            index = i;
            break;
        }
    }

    //Add relevant variables of rootNode into the set relVars
    addRelVars(rootNode, relVars);

    //Add the chain of control dependent nodes of rootNode
    //into the set relNodes
    ArrayList controlNodes = this.getControlNodesChain(rootNode);
    if (controlNodes != null)
        relNodes.addAll(controlNodes);

    //Examine the nodes from the index-1 to 0 backward
    for (int i = index - 1; i >= 0; i--) {
        //Break out of the for loop if relVars is empty which
        //means the slicing has been completed.
        if (relVars.isEmpty())
            break;
        StatementNode n = (StatementNode) nodes[i];

        // Get the variable defined at the statement
        DefUse def = n.getDefinition();
        HasValue defVar = null;
        if (def != null)
            defVar = getVar(def);

        if (n.getType() == StatementNode.PREDICATE_NODE) {
            if (relNodes.contains(n)) {
                //Add the variables used in the predicate node
                addRelVars(n, relVars);

                // remove the variable defined from relVars
                if (defVar != null && relVars.contains(defVar))
                    relVars.remove(defVar);
            }
        }
    }
    else{

```

```

    if (defVar == null || !relVars.contains(defVar)) {
        DefUse[] uses = n.getUses();
        boolean isReferred = false;
        HasValue usedVar = null;
        for (int k = 0; k < uses.length; k++) {
            usedVar = getVar(uses[k]);
            if (usedVar != null && relVars.contains(usedVar)) {
                isReferred = true;
                break;
            }
        }
        if (isReferred) {
            if (!refNodes.contains(n))
                refNodes.add(n);
            addRelVars(n, relVars);
        }
        else if (defVar != null && relVars.contains(defVar)) {
            if (!relNodes.contains(n))
                relNodes.add(n);

            //Add the predicate node it control-dependent on
            //into the set relNodes
            Node aPredNode = (Node) controlEdges.get(n);
            if (aPredNode != null) {
                if (!relNodes.contains(aPredNode))
                    relNodes.add(aPredNode);
            }
            else
                //it is not control dependent on any nodes, thus
                //remove the variable defined from relVars
                relVars.remove(defVar);

            //Add the variables used in this node into relVars
            addRelVars(n, relVars);
        }
    }
    return relNodes;
}

// Add the relevant variables of node n into the set relVars
private void addRelVars(Node n, HashSet relVars) {
    //Get the variables used at node n
    DefUse[] uses = ((StatementNode) n).getUses();
    HasValue usedVar = null;
    for (int i = 0; i < uses.length; i++) {
        usedVar = getVar(uses[i]);
        if (usedVar != null && !(usedVar instanceof TempVariable)
            && !(usedVar instanceof FormalParameterImpl))
            if (!relVars.contains(usedVar))
                relVars.add(usedVar);
    }
}

```

3. A sample program and its execution traces:

```
1 class testProg
2 {
3     public static void main(String[] args)
4     {
5         int i=3;
6         int j=9;
7         int z=0;
8
9         if ((i-j)%2==0)
10            if (j>0 && i>0)
11                z=function1(i,j);
12            else
13                z=function2(i,j);
14
15        while(i!=0)
16        {
17            if(z<100)
18                z++;
19            i--;
20        }
21
22        System.out.println(z);
23    }
24
25    public static int function1 (int i, int j)
26    {
27        int k = function2(i,j);
28        return k-i+j;
29    }
30
31    public static int function2(int i, int j)
32    {
33        return i*j;
34    }
35 }
```

Exeuction Traces	Comments
testProg.main:5,6,7,9,10,11,	Input value i=3, j=9 and z=0 and call function1 at line 11.
testProg. function1:27,	Call function2 at line 27
testProg. function2:33,exit	Return 3*9 = 27 and exit function2
testProg. function1:28,exit	Return 27-3+9 = 33 and exit function1
testProg.main:15,17,18,19,15,17,18,19,15,17,18,19,15,22,exit	As i=3, three loops are executed and z = 36

Author's Publications

- [1] Liu, H. and Tan, H.B.K. 2006. Automated Verification and Test Case Generation for Input Validation. in Proceedings of 1st Workshop on Automation of Software Test Co-located with ICSE 2006, (Shanghai, China), 29-35.
- [2] Liu, H. and Tan, H.B.K. 2006. An approach to aid the understanding and maintenance of input validation. in Proceedings of the 22nd International Conference on Software Maintenance, (Philadelphia, PA, USA), 370-379.
- [3] Liu, H. and Tan, H.B.K. 2008. An approach for the maintenance of input validation. Information and Software Technology, 50 (5), 449-461.
- [4] Liu, H. and Tan, H.B.K. 2008. Testing input validation in Web applications through automated model recovery. Journal of Systems and Software, 81 (2), 222-233.
- [5] Liu, H. and Tan, H.B.K. 2009. Covering Code Behavior on Input Validation in Functional Testing. Information and Software Technology, 51 (2), 546-553.

Bibliography

- [1] Aho, A.V., Sethi, R. and Ullman, J.D. 1986. Compilers, Principles, Techniques and Tools. Addison-Wesley.
- [2] Allen, F.E. and Cocke, J. 1976. A program data flow analysis procedure. Communications of the ACM, 19 (3), 137-146.
- [3] Antoniol, G., Di Penta, M. and Zazzara, M. 2004. Understanding Web applications through dynamic analysis. in Proceedings. 12th IEEE International Workshop on Program Comprehension, 24-26 June 2004, (Bari, Italy), IEEE Comput. Soc, 120-129.
- [4] Apiwattanapong, T., Orso, A. and Harrold, M.J. 2005. Efficient and precise dynamic impact analysis using execute-after sequences. in Proceedings of the 27th International Conference on Software Engineering, (St. Louis, MO, USA), IEEE, 432-441.
- [5] Ball, T. 1999. The concept of dynamic analysis. in Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering, (Toulouse, France), Springer-Verlag.
- [6] Balzer, R.M. 1969. EXDAMS-extendable debugging and monitoring system. 35.
- [7] Basili, V., Laitenberger, O., Shull, F. and Rus, I. 2000. Improving software inspections by using reading techniques. in Proceedings of International Conference on Software Engineering, (Limerick, Ireland), IEEE, 836.
- [8] Basili, V., Shull, F. and Lanubile, F. 2000. Using experiments to build a body of knowledge. in Proceedings of the 3rd International Andrei Ershov

- Memorial Conference - Perspectives of Systems Informatics., (Akademgorodok, Russia), Springer-Verlag, 265-282.
- [9] Basili, V.R. 1996. The role of experimentation in software engineering: past, current, and future. in Proceedings of the 18th International Conference on Software Engineering, (Berlin, Germany), IEEE Comput. Soc., 442-449.
- [10] Basili, V.R. and Selby, R.W. 1987. Comparing the effectiveness of software testing strategies. IEEE Transactions on Software Engineering, SE-13 (12), 1278-1296.
- [11] Bauer, J.A. and Finger, A.G. 1979. Test plan generation using formal grammars. in Proceedings of the 4th International Conference on Software Engineering, (Munich, West Germany), IEEE, 425-432.
- [12] Bazzichi, F. and Spadafora, I. 1982. An automatic generator for compiler testing. IEEE Transactions on Software Engineering, SE-8 (4), 343-353.
- [13] Beck, J. and Eichmann, D. 1993. Program and interface slicing for reverse engineering. in Proceedings of the 15th International Conference on Software Engineering, (Baltimore, MD, USA), IEEE, 509-518.
- [14] Beizer, B. 1990. Software testing techniques. Van Nostrand Reinhold, New York.
- [15] Belli, F. and Dreyer, J. 1997. Program segmentation for controlling test coverage. in Proceedings of the 8th International Symposium on Software Reliability Engineering, (Albuquerque, NM, USA), IEEE Comput. Soc, 72-83.
- [16] Bently, W.G. and Miller, E.F. 1993. Ct coverage-initial results. Software Quality Journal, 2 (1), 29-47.
- [17] Bergeretti, J.-F. and Carré, B.A. 1985. Information-flow and data-flow analysis of while-programs. ACM Trans. Program. Lang. Syst., 7 (1), 37-61.
- [18] Berling, T. and Thelin, T. 2004. A case study of reading techniques in a software company. in Proceedings of International Symposium on Empirical Software Engineering, (Redondo Beach, CA, USA), IEEE Comput. Soc, 229-238.
- [19] Beszedes, A., Farago, C., Mihaly Szabo, Z., Csirik, J. and Gyimothy, T. 2002. Union slices for program maintenance. in Proceedings of

- International Conference on Software Maintenance, (Montreal, Que., Canada), IEEE Comput. Soc, 12-21.
- [20] Beydeda, S., Gruhn, V. and Stachorski, M. 2001. A graphical class representation for integrated black- and white-box testing. in Proceedings of International Conference on Software Maintenance, (Florence, Italy), IEEE, 706-715.
- [21] Biggerstaff, T.J. 1989. Design recovery for maintenance and reuse. *Computer*, 22 (7), 36-49.
- [22] Bilardi, G. and Pingali, K. 1996. Framework for generalized control dependence. in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, (Philadelphia, PA, USA), ACM, 291-300.
- [23] Binkey, D. and Gallagher, K.B. 1996. Program Slicing. *Advances in Computers*, 43, 1-50.
- [24] Binkley, D. 1993. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 2 (1-4), 31-45.
- [25] Binkley, D., Harman, M., Raszewski, L.R. and Smith, C. 2000. An empirical study of amorphous slicing as a program comprehension support tool. in Proceedings of the 8th International Workshop on Program Comprehension, (Limerick, Ireland), IEEE Comput. Soc, 161-170.
- [26] Bond, M.D. and McKinley, K.S. 2005. Practical path profiling for dynamic optimizers. in Proceedings of International Symposium on Code Generation and Optimization, (San Jose, CA, United States), IEEE, 205-216.
- [27] Bueno, P.M.S. and Jino, M. 2002. Automatic test data generation for program paths using genetic algorithms. *International Journal of Software Engineering and Knowledge Engineering*, 12 (6), 691-709.
- [28] Buy, U., Orso, A. and Pezze, M. 2000. Automated testing of classes. in Proceedings of the International Symposium on Software Testing and Analysis, (Portland, OR, USA), IEEE, 39-48.
- [29] Cardell-Oliver, R. and Glover, T. 1998. A practical and complete algorithm for testing real-time systems. in Proceedings of the 5th International

- Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, (Lyngby, Denmark), Springer-Verlag, 251-261.
- [30] Carver, R.H. and Tai, K.-C. 1998. Use of sequencing constraints for specification-based testing of concurrent programs. *IEEE Transactions on Software Engineering*, 24 (6), 471-490.
- [31] Chen, H.Y., Tse, T.H., Chan, F.T. and Chen, T.Y. 1998. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 7 (3), 250-295.
- [32] Chen, H.Y., Tse, T.H. and Chen, T.Y. 2001. TACCLE: a methodology for object-oriented software Testing at the Class and Cluster Levels. *ACM Transactions on Software Engineering and Methodology*, 10 (1), 56-109.
- [33] Chen, K. and Rajlich, V. 2000. Case study of feature location using dependence graph. in *Proceedings of 8th International Workshop on Program Comprehension*, (Limerick, Ireland), IEEE Comput. Soc, 241-247.
- [34] Chen, T.Y. and Lau, M.F. 1999. On the Completeness of a Test Suite Reduction Strategy. *Computer Journal*, 42 (5), 430-440.
- [35] Chen, T.Y., Poon, P.L., Tang, S.F. and Yu, Y.T. 2000. White on black: a white-box-oriented approach for selecting black box-generated test cases. in *Proceedings of the 1st Asia-Pacific Conference on Quality Software*, (Hong Kong, China), IEEE Comput. Soc, 275-284.
- [36] Chikofsky, E.J. and Cross, J.H., II 1990. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7 (1), 13-17.
- [37] Cimitile, A., de Lucia, A. and Munro, M. 1996. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance: Research and Practice*, 8 (3), 145-178.
- [38] Clarke, L.A. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2 (3), 215-222.
- [39] Clarke, L.A., Podgurski, A., Richardson, D.J. and Zeil, S.J. 1989. Formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15 (11), 1318-1332.

- [40] Clarke, P.J., Ding, J., Babich, D. and Malloy, B.A. 2006. A tool to automatically map implementation-based testing techniques to classes. *International Journal of Software Engineering and Knowledge Engineering*, 16 (4), 585-614.
- [41] Cohen, M.B., Dwyer, M.B. and Jiangfan, S. 2006. Coverage and adequacy in software product line testing. in *Proceedings of ISSTA Workshop on Role of Software Architecture for Testing and Analysis*, (Portland, ME, United States), ACM, 53-63.
- [42] Cook, T.D. and Campbell, D.T. 1979. *Quasi-Experimentation: Design & Analysis Issues for Field Settings*. Houghton Mifflin Company.
- [43] Csallner, C. and Smaragdakis, Y. 2005. Check 'n' crash: Combining static checking and testing. in *Proceedings of the 27th International Conference on Software Engineering*, (Saint Louis, MO, United States), IEEE, 422-431.
- [44] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13 (4), 451-490.
- [45] Cytron, R., Ferrante, J. and Sarkar, V. 1990. Compact representations for control dependence. in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (White Plains, NY, USA), ACM, 337-351.
- [46] De Lucia, A. 2001. Program slicing: methods and applications. in *Proceedings of 1st IEEE International Workshop on Source Code Analysis and Manipulation*, (Florence, Italy), IEEE Comput. Soc, 142-149.
- [47] De Lucia, A., Harman, M., Hierons, R. and Krinke, J. 2003. Unions of slices are not slices. in *Proceedings of 7th European Conference on Software Maintenance and Reengineering*, (Benevento, Italy), IEEE Comput. Soc, 363-367.
- [48] Deng, Y., Frankl, P. and Wang, J. 2004. Testing Web Database Applications. *SIGSOFT Softw. Eng. Notes*, 29, 1-10.

- [49] Devanbu, P. 1995. On "A framework for source code search using program patterns". IEEE Transactions on Software Engineering, 21 (12), 1009-1010.
- [50] Duesterwald, E., Gupta, R. and Soffa, M.L. 1997. A practical framework for demand-driven interprocedural data flow analysis. ACM Transactions on Programming Languages and Systems, 19 (6), 992-1030.
- [51] Duncan, A.G. and Hutchison, J.S. 1981. Using attributed grammars to test designs and implementations. in Proceedings of the 5th International Conference on Software Engineering, (San Diego, CA, USA), IEEE, 170-178.
- [52] Dunsmore, A., Roper, M. and Wood, M. 2003. The development and evaluation of three diverse techniques for object-oriented code inspection. IEEE Transactions on Software Engineering, 29 (8), 677-686.
- [53] Dunsmore, A., Roper, M. and Wood, M. 2002. Further investigations into the development and evaluation of reading techniques for object-oriented code inspection. in Proceedings of the 24th International Conference on Software Engineering, ACM, 47-57.
- [54] Eclipse. Open source Java development tool. <http://www.eclipse.org/>.
- [55] Eisenbarth, T., Koschke, R. and Simon, D. 2003. Locating features in source code. IEEE Transactions on Software Engineering, , 29 (3), 210-224.
- [56] Ferguson, R. and Korel, B. 1996. The chaining approach for software test data generation. ACM Transactions on Software Engineering and Methodology, 5 (1), 63-86.
- [57] Ferrante, J., Ottenstein, K.J. and Warren, J.D. 1987. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9 (3), 319-349.
- [58] Frankl, P.G. and Weyuker, E.J. 1988. An applicable family of data flow testing criteria. IEEE Transactions on Software Engineering, 14 (10), 1483-1498.

- [59] Gallagher, K., Harman, M. and Danicic, S. 2003. Guaranteed inconsistency avoidance during software evolution. *Journal of Software Maintenance and Evolution Research and Practice*, 15 (6), 393-416.
- [60] Gallagher, K.B. 2004. Some notes on interprocedural program slicing. in *Proceedings of the 4th International Workshop on Source Code Analysis and Manipulation*, (Chicago, IL, United States), IEEE Computer Society, 36-42.
- [61] Gallagher, K.B. 1996. Visual impact analysis. in *Proceedings of International Conference on Software Maintenance*, (Monterey, CA, USA), IEEE Comput. Soc. Press, 52-58.
- [62] Gallagher, K.B. and Lyle, J.R. 1991. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17 (8), 751-761.
- [63] Goodenough, J.B. and Gerhart, S.L. 1975. Toward a theory of test data selection. *SIGPLAN Notices*, 10, 493-510.
- [64] Gotlieb, A., Botella, B. and Rueher, M. 1998. Automatic test data generation using constraint solving techniques. *International Symposium on Software Testing and Analysis*, 23 (2), 53-62.
- [65] GraphViz 2007. Doty. <http://www.graphviz.org/>.
- [66] Gupta, N., Mathur, A.P. and Soffa, M.L. 1998. Automated test data generation using an iterative relaxation method. in *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, (Lake Buena Vista, FL, USA), ACM, 231-244.
- [67] Gupta, N., Mathur, A.P. and Soffa, M.L. 2000. Generating test data for branch coverage. in *Proceedings of 15th IEEE International Automated Software Engineering Conference*, (Grenoble, France), IEEE Comput. Soc, 219-227.
- [68] Gupta, N., Mathur, A.P. and Soffia, M.L. 1999. UNA based iterative test data generation and its evaluation. in *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, (Cocoa Beach, FL, USA), IEEE Comput. Soc, 224-232.

- [69] Gupta, R., Soffa, M.L. and Howard, J. 1997. Hybrid slicing: integrating dynamic information with static analysis. *ACM Transactions on Software Engineering and Methodology*, 6 (4), 370-397.
- [70] Gupta, R., Soffa, M.L. and Howard, J. 1997. Hybrid slicing: integrating dynamic information with static analysis. *ACM Transactions on Software Engineering and Methodology*, 6 (4), 370-397.
- [71] Haas, D.F. and Kraft, D.H. 1984. Experimental and quasi-experimental designs for research in information science. in *Proceedings of Information Processing & Management*, (Atlanta, GA, USA), IEEE, 229-237.
- [72] Hall, R.J. 1995. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2 (1), 33-53.
- [73] Harman, M., Binkley, D. and Danicic, S. 2003. Amorphous program slicing. *Journal of Systems and Software*, 68 (1), 45-64.
- [74] Harrold, M.J., Gupta, R. and Soffa, M.L. 1993. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2 (3), 270-285.
- [75] Harrold, M.J., Miller, R., Porter, A. and Rothermel, G. 1997. A collaborative investigation of program-analysis-based testing and maintenance. in *Proceedings of 2nd International Workshop on Empirical Studies of Software Maintenance*, (Bari, Italy), 51-56.
- [76] Harrold, M.J., Rothermel, G. and Sinha, S. 1998. Computation of interprocedural control dependence. *Software Engineering Notes*, 23 (2), 11-20.
- [77] Harrold, M.J. and Soffa, M.L. 1994. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16 (2), 175-204.
- [78] Hayes, J.H. and Offutt, A.J. 1999. Increased software reliability through input validation analysis and testing. in *Proceedings of the 10th International Symposium on Software Reliability Engineering*, (Boca Raton, FL, USA), IEEE Comput. Soc, 199-209.

- [79] Hayes, J.H. and Offutt, J. 2006. Input validation analysis and testing. *Empirical Softw. Engg.*, 11 (4), 493-522.
- [80] Horwitz, S., Reps, T. and Binkley, D. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12 (1), 26-60.
- [81] Horwitz, S., Reps, T. and Sagiv, M. 1995. Demand interprocedural dataflow analysis. in *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM, 104-115.
- [82] JABA 2006. Java Architecture for Bytecode Analysis. <http://www.cc.gatech.edu/aristotle/Tools/jaba.html>.
- [83] Jackson, D. and Rollins, E.J. 1994. A new model of program dependences for reverse engineering. *SIGSOFT Softw. Eng. Notes*, 19, 2-10.
- [84] Johnsonbaugh, R. and Schaefer, M. 2004. *Algorithms*. Pearson, Prentice Hall, Singapore.
- [85] Jorgensen, P. 2002. *Software testing : a craftman's approach*. CRC Press, Boca Raton, FL.
- [86] JSlice. 2008. Dynamic slicing tool for Java programs. <http://jslice.sourceforge.net>.
- [87] Juristo, N. and Moreno, A.M. 2003. *Lecture notes on empirical software engineering*. World Scientific Publishing Co., Inc.
- [88] Kaffe 2008. Java virtual machine. <http://www.kaffe.org/>.
- [89] Kamsties, E. and Lott, C.M. 1995. An empirical evaluation of three defect-detection techniques. in *Proceedings of the 5th European Software Engineering Conference*, (Sitges, Spain), Springer-Verlag, 362-383.
- [90] Khenaidoo, N. and Robert, L.P. 1995. Cause-effect graphing analysis and validation of requirements. in *Proceedings of the conference of the Centre for Advanced Studies on Collaborative Research*, (Toronto, Ontario, Canada), IBM Press.
- [91] Khor, S. and Grogono, P. 2004. Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically. in *Proceedings of the 19th International Conference on Automated Software Engineering*, (Linz, Austria), IEEE Comput. Soc, 346-349.

- [92] Khor, S. and Grogono, P. 2004. Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically. in Proceedings. 19th International Conference on Automated Software Engineering, 20-24 Sept. 2004, IEEE Comput. Soc, 346-349.
- [93] Kiss, A., Jasz, J., Lehotai, G. and Gyimothy, T. 2003. Interprocedural static slicing of binary executables. in Proceedings of the 3rd International Workshop on Source Code Analysis and Manipulation, (Amsterdam, Netherlands), IEEE Comput. Soc, 118-127.
- [94] Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K. and Rosenberg, J. 2002. Preliminary guidelines for empirical research in software engineering. IEEE Transactions on Software Engineering, 28 (8), 721-734.
- [95] Kontogiannis, K., DeMori, R., Bernstein, M., Galler, M. and Merlo, E. 1995. Pattern matching for design concept localization. in Proceedings of the 2nd Working Conference on Reverse Engineering, (Toronto, Ont., Canada), IEEE Comput. Soc. Press, 96-103.
- [96] Kontogiannis, K.A., Demori, R., Merlo, E., Galler, M. and Bernstein, M. 1996. Pattern matching for clone and concept detection. Automated Software Engineering, 3 (1-2), 77-108.
- [97] Korel, B. 1990. Automated software test data generation. IEEE Transactions on Software Engineering, 16 (8), 870-879.
- [98] Korel, B. 1996. Automated test data generation for programs with procedures. in Proceedings of the International Symposium on Software Testing and Analysis, (San Diego, CA, USA), ACM, 209-215.
- [99] Korel, B. 1992. Dynamic method for software test data generation. Software Testing Verification and Reliability, 2 (4), 203-213.
- [100] Korel, B. and Al-Yami, A.M. 1996. Assertion-oriented automated test data generation. in Proceedings of the 18th International Conference on Software Engineering, (Berlin, Ger), IEEE, 71-80.
- [101] Korel, B. and Laski, J. 1988. Dynamic program slicing. Information Processing Letters, 29 (3), 155-163.

- [102] Korel, B. and Laski, J. 1990. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13 (3), 187-195.
- [103] Korel, B. and Rilling, J. 1998. Dynamic program slicing methods. *Information and Software Technology*, 40 (11-12), 647-659.
- [104] Krinke, J. 2002. Evaluating context-sensitive slicing and chopping. in *Proceedings of the International Conference on Software Maintenance*, (Montreal, Que., Canada), IEEE Comput. Soc, 22-31.
- [105] Krinke, J. 2004. Slicing, chopping, and path conditions with barriers. *Software Quality Journal*, 12 (4), 339-360.
- [106] Lanubile, F. and Visaggio, G. 1997. Extracting reusable functions by flow graph based program slicing. *IEEE Transactions on Software Engineering*, 23 (4), 246-259.
- [107] Larus, J.R. and Ball, T. 1994. Rewriting executable files to measure program behavior. *Software - Practice and Experience*, 24 (2), 197-218.
- [108] Lengauer, T. and Tarjan, R.E. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1 (1), 121-141.
- [109] Liu, C.-H., Kung, D.C., Hsia, P. and Hsu, C.-T. 2000. Structural testing of Web applications. in *Proceedings of the 11th International Symposium on Software Reliability Engineering*, (San Jose, CA, USA), IEEE, 84-96.
- [110] Liu, H. and Tan, H.B.K. 2008. An approach for the maintenance of input validation. *Information and Software Technology*, 50 (5), 449-461.
- [111] Liu, H. and Tan, H.B.K. 2006. An approach to aid the understanding and maintenance of input validation. in *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, (Philadelphia, PA, USA), IEEE Comput. Soc, 370-379.
- [112] Liu, H. and Tan, H.B.K. 2006. Automated Verification and Test Case Generation for Input Validation. in *Proceedings of 1st Workshop on Automation of Software Test Co-located with ICSE 2006*, (Shanghai, China), 29-35.

- [113] Liu, H. and Tan, H.B.K. 2009. Covering Code Behavior on Input Validation in Functional Testing. *Information and Software Technology*, 51 (2), 546-553.
- [114] Liu, H. and Tan, H.B.K. 2008. Testing input validation in Web applications through automated model recovery. *Journal of Systems and Software*, 81 (2), 222-233.
- [115] Loyall, J.P. and Mathisen, S.A. 1993. Using dependence analysis to support the software maintenance process. in *Proceedings of the International Conference on Software Maintenance*, (Montreal, Que., Canada), IEEE Comput. Soc. Press, 282-291.
- [116] Lyle, J.R. and Weiser, M. 1987. Automatic program bug location by program slicing. in *Proceedings of the 2nd International Conference on Computers and Applications*, (Beijing, China), IEEE Comput. Soc. Press, 877-883.
- [117] MacDonald, F., Miller, J., Brooks, A., Roper, M. and Wood, M. 1996. Applying inspection to object-oriented code. *Journal of Software Testing Verification and Reliability*, 6 (2), 61-82.
- [118] Mandrioli, D., Morasca, S. and Morzenti, A. 1995. Generating test cases for real-time systems from logic specifications. *ACM Transactions on Computer Systems*, 13 (4), 365-398.
- [119] Mansour, N. and Salame, M. 2004. Data generation for path testing. *Software Quality Journal*, 12 (2), 121-136.
- [120] Marick, B. 1995. *The craft of software testing : subsystem testing including object-based and object-oriented testing*. PTR Prentice Hall, Englewood Cliffs, N.J.
- [121] Maurer, P.M. 1990. Generating test data with enhanced context-free grammars. *IEEE Software*, 7 (4), 50-55.
- [122] McCabe, T.J. 1982. Structured testing: A software testing methodology using the cyclomatic complexity metric. x+63.
- [123] McCabe, T.J. 1982. Structured testing: A software testing methodology using the cyclomatic complexity metric. National Bureau of Standards, Special Publication.

- [124] Memon, A.M., Soffa, M.L. and Pollack, M.E. 2001. Coverage criteria for GUI testing. in Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, (Vienna, Austria), Association for Computing Machinery, 256-267.
- [125] Michael, C.C., McGraw, G. and Schatz, M.A. 2001. Generating software test data by evolution. IEEE Transactions on Software Engineering, 27 (12), 1085-1110.
- [126] Montgomery, D.C. and Runger, G.C. 2003. Applied statistics and probability for engineers. J. Wiley, New York.
- [127] Morell, L.J. 1990. A theory of fault-based testing. IEEE Transactions on Software Engineering, 16 (8), 844-857.
- [128] MSDN 2006. Design Guidelines for Secure Web Application. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod77.asp>.
- [129] MSDN. 2003. Input Validation. in Improving Web Application Security - Threats and Countermeasures, 24-28.
- [130] Muccini, H., Inverardi, P. and Bertolino, A. 2004. Using software architecture for code testing. IEEE Transactions on Software Engineering, 30 (3), 160-171.
- [131] Myers, G.J., Badgett, T., Thomas, T.M. and Sandler, C. 2004. The art of software testing. John Wiley & Sons, Hoboken, N.J.
- [132] Nakajo, T. and Kume, H. 1991. A case history analysis of software error cause-effect relationships. IEEE Transactions on Software Engineering, 17 (8), 830-838.
- [133] Nakajo, T., Sasabuchi, K. and Akiyama, T. 1989. A structured approach to software defect analysis. Hewlett-Packard Journal, 40 (2), 50-56.
- [134] Nanda, M.G. and Ramesh, S. 2006. Interprocedural slicing of multithreaded programs with applications to Java. ACM Transactions on Programming Languages and Systems, 28 (6), 1088-1144.
- [135] Nielson, F., Nielson, H.R. and Hankin, C. 1999. Principles of program analysis. Springer, Berlin ; New York.

- [136] Ntafos, S.C. 1988. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14 (6), 868-874.
- [137] Offutt, J., Wu, Y., Du, X. and Huang, H. 2004. Bypass testing of Web applications. in *Proceedings of the 15th International Symposium on Software Reliability Engineering*, (Saint-Malo, Bretagne, France), IEEE Comput. Soc, 187-197.
- [138] Orso, A., Apiwattanapong, T., Law, J., Rothermel, G. and Harrold, M.J. 2004. An empirical comparison of dynamic impact analysis algorithms. in *Proceedings of the 26th International Conference on Software Engineering*, (Edinburgh, United Kingdom), IEEE, 491-500.
- [139] Ostrand, T.J. and Balcer, M.J. 1988. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31 (6), 676-686.
- [140] Ottenstein, K.J. and Ottenstein, L.M. 1984 The program dependence graph in a software development environment in *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments* ACM Press, 177-184
- [141] Paradkar, A., Tai, K.C. and Vouk, M.A. 1997. Specification-based testing using cause-effect graphs. *Annals of Software Engineering*, 4, 133-157.
- [142] Perry, D.E. 1987. An empirical study of software interface faults-an update. in *Proceedings of the 20th Hawaii International Conference on System Sciences*, Hawaii Int. Conference Syst. Sci, 113-126.
- [143] Perry, D.E. and Evangelist, W.M. 1985. An empirical study of software interface faults. in *Proceedings of the International Symposium on New Directions in Computing*, (Trondheim, Norway), IEEE Comput. Soc. Press, 32-38.
- [144] Perry, D.E., Porter, A.A. and Votta, L.G. 2000 Empirical studies of software engineering: a roadmap. in *Proceedings of the Conference on The Future of Software Engineering* (Limerick, Ireland), ACM, 345-355
- [145] Podgurski, A. and Clarke, L.A. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16 (9), 965-979.

- [146] Poole, J. 1995. Method to Determine a Basis Set of Paths to Perform Program Testing., 18p.
- [147] Purdom, P. 1972. A sentence generator for testing parsers. BIT (Nordisk Tidskrift for Informationsbehandling), 12 (3), 366-375.
- [148] Ramamoorthy, C.V., Ho, S.-B.F. and Chen, W.T. 1976. On the automated generation of program test data. IEEE Transactions on Software Engineering, SE-2 (4), 293-300.
- [149] Reps, T., Horwitz, S. and Sagiv, M. 1995. Precise interprocedural dataflow analysis via graph reachability. in Proceedings of the 22nd Symposium on Principles of Programming Languages, ACM, 49-61.
- [150] Reps, T. and Rosay, G. 1995. Precise interprocedural chopping. in Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, (Washington, DC, USA), ACM, 41-52.
- [151] Ricca, F. and Tonella, P. 2001. Web application slicing. in Proceedings of the International Conference on Software Maintenance, (Florence, Italy), IEEE Comput. Soc, 148-157.
- [152] Richardson, D.J. and Clarke, L.A. 1985. Partition analysis: a method combining testing and verification. IEEE Transactions on Software Engineering, SE-11 (12), 1477-1490.
- [153] Rilling, J. and Karanth, B. 2001. A hybrid program slicing framework. in Proceedings of the 1st International Workshop on Source Code Analysis and Manipulation, (Florence, Italy), IEEE Comput. Soc, 12-23.
- [154] Rilling, J. and Klemola, T. 2003. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. in Proceedings of the 11th IEEE International Workshop on Program Comprehension, (Portland, OR, USA), IEEE Comput. Soc, 115-124.
- [155] Rountev, A., Kagan, S. and Sawin, J. 2005. Coverage criteria for testing of object interactions in sequence diagrams in Proceedings of 8th International Conference of Fundamental Approaches to Software Engineering, (Edinburgh, UK), Springer-Verlag, 289-304.
- [156] Rountev, A., Kagan, S. and Sawin, J. 2005. Coverage criteria for testing of object interactions in sequence diagrams in Proceedings of 8th

- International Conference of Fundamental Approaches to Software Engineering (FASE'05), (Edinburgh, UK), Springer-Verlag, 289-304.
- [157] Ruthruff, J.R., Elbaum, S. and Rothermel, G. 2006. Experimental program analysis: a new program analysis paradigm. in Proceedings of the 2006 international symposium on Software testing and analysis, (Portland, Maine, USA), ACM Press, 49-60.
- [158] Sartipi, K. and Kontogiannis, K. 2003. On modeling software architecture recovery as graph matching. in Proceedings of the International Conference on Software Maintenance, IEEE Comput. Soc, 224-234.
- [159] Shull, F., Lanubile, F. and Basili, V.R. 2000. Investigating reading techniques for object-oriented framework learning. IEEE Transactions on Software Engineering, 26 (11), 1101-1118.
- [160] Sinha, S. and Harrold, M.j. 2000. Analysis and Testing of Programs with Exception Handling Construct. IEEE Transaction on Software Engineering, 26 (9).
- [161] Sinha, S., Harrold, M.J. and Rothermel, G. 2001. Interprocedural control dependence. ACM Transactions on Software Engineering and Methodology, 10 (2), 209-254.
- [162] Sjöberg, D.I.K., Dyba, T. and Jorgensen, M. 2007 The Future of Empirical Methods in Software Engineering Research. in Proceedings of the Future of Software Engineering, IEEE Computer Society, 358-378
- [163] Sjöberg, D.I.K., Hannay, J.E., Hansen, O., Kampenes, V.B., Karahasanovic, A., Liborg, N.-K. and Rekdal, A.C. 2005. A survey of controlled experiments in software engineering. IEEE Transactions on Software Engineering, 31 (9), 733-753.
- [164] Sourceforge. Open-source website. <http://sourceforge.net/>.
- [165] Sy, N.T. and Deville, Y. 2001. Automatic test data generation for programs with integer and float variables. in Proceedings of the 16th Annual International Conference on Automated Software Engineering, (San Diego, CA, USA), IEEE Comput. Soc, 13-21.
- [166] Tai, K.-C. 1996. Theory of fault-based predicate testing for computer programs. IEEE Transactions on Software Engineering, 22 (8), 552-562.

- [167] Tan, H.B.K. and Kow, J.T. 2001. An approach for extracting code fragments that implement functionality from source programs. *Journal of Software Maintenance and Evolution*, 13 (1), 53-75.
- [168] Taylor, R.N., Levine, D.L. and Kelly, C.D. 1992. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18 (3), 206-215.
- [169] Tip, F. 1995. A survey of program slicing techniques. *Journal of Programming Languages*, 3 (3), 121-189.
- [170] Tip, F. 1994. A Survey of Program Slicing Techniques. CWI (Centre for Mathematics and Computer Science).
- [171] Tonella, P. 2003. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering*, 29 (6), 495-509.
- [172] Venkatesh, G.A. 1991. Semantic approach to program slicing. in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (Toronto, Ont, Can), ACM, 107.
- [173] Watkins, A. and Hufnagel, E.M. 2006. Evolutionary test data generation: A comparison of fitness functions. *Software - Practice and Experience*, 36 (1), 95-116.
- [174] Watson, A.H. and McCabe, T.J. 1996. Structured testing: A software testing methodology using the cyclomatic complexity metric.
- [175] Webmaster 2006. Open Source Website. <http://code.cnzz.cn>.
- [176] Weiser, M. 1981. Program slicing. in *5th International Conference on Software Engineering*, 9-12 March 1981, (San Diego, CA, USA), IEEE, 439-449.
- [177] Weyuker, E.J., Weiss, S.N. and Hamlet, D. 1991. Comparison of program testing strategies. in *Proceedings of the Symposium on Testing, Analysis, and Verification*, (Victoria, BC, Can), ACM.
- [178] Wilde, N. and Scully, M.C. 1995. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7 (1), 49-62.

- [179] Wohlin, C. 2000. Experimentation in software engineering : an introduction. Kluwer Academic, Boston.
- [180] Wong, W.E. 2000. An integrated solution for creating dependable software. in Proceedings of the 24th Annual International Computer Software and Applications Conference, (Taipei, Taiwan), IEEE Comput. Soc, 269-270.
- [181] Wood, M., Roper, M., Brooks, A. and Miller, J. 1997. Comparing and combining software defect detection techniques: a replicated empirical study. *Softw. Eng. Notes*, 22, 262-277.
- [182] Woodward, M.R. and Hennell, M.A. 2006. On the relationship between two control-flow coverage criteria: all JJ-paths and MCDC. *Information and Software Technology*, 48 (7), 433-440.
- [183] Xie, T., Marinov, D., Schulte, W. and Notkin, D. 2005. Symstra: a framework for generating object-oriented unit tests using symbolic execution. in Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, (Edinburgh, UK), Springer-Verlag, 365-381.
- [184] Yu, Y.T., Tang, S.F., Poon, P.L. and Chen, T.Y. 2001. A study on a path-based strategy for selecting black-box generated test cases. *International Journal of Software Engineering and Knowledge Engineering*, 11 (2), 113-138.
- [185] Zelkowitz, M.V. and Wallace, D. 1997. Experimental validation in software engineering. *Information and Software Technology*, 39 (11), 735-743.
- [186] Zhang, J., Chen, X. and Wang, X. 2004. Path-oriented test data generation using symbolic execution and constraint solving techniques. in Proceedings of the 2nd International Conference on Software Engineering and Formal Methods, (Beijing, China), IEEE Comput. Soc, 242-250.
- [187] Zhu, H., Hall, P.A.V. and May, J.H.R. 1997. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29 (4), 366-427.