Automatic Software Testing Via Mining Software Data

ZHENG, Wujie

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
September 2011

Thesis/Assessment Committee

Professor Patrick Pak Ching LEE (Chair)

Professor Michael Rung Tsong LYU (Thesis Supervisor)

Professor Ada Wai Chee FU (Committee Member)

Professor Shing Chi CHEUNG (External Examiner)

Abstract of thesis entitled:

Automatic Software Testing Via Mining Software Data

Submitted by ZHENG, Wujie

for the degree of Doctor of Philosophy

at The Chinese University of Hong Kong in September 2011

Software testing has been the primary way for assuring high quality of software systems. In software testing, testers need to design a set of test cases, including test inputs and expected outputs, to cover most of the code and find most bugs before releasing the software. It is challenging to generate such effective test cases manually for complex software systems nowadays. Automatic software testing reduces the laborious human effort in testing. However, the existing test input generation and test oracle generation techniques often require manually provided specifications, which may not be available.

In this thesis, we propose to mine specifications from various kinds of software data, such as the source code, the execution trace, and the existing outputs, to generate test inputs and to verify test outputs automatically.

First, we mine relevant APIs from source code to guide random unit-test generation. Given a method under test (MUT), a key component of object-oriented unit-test generation is to find method-call sequences that create and mutate desired inputs, including the receiver and the arguments. We present a MUT-aware sequence recommendation approach to improve the effectiveness of random object-oriented unit-test generation. Our approach mines relevant APIs for the MUT, i.e. methods that may mutate object fields accessed

by the MUT, from the source code. It then recommends short sequences that contain some relevant APIs as the inputs of the MUT. In this way, our approach generates test inputs that have a high chance of exploring most behaviors of the MUT.

Second, we mine potential rules of control branches and variables from execution traces of unverified tests to select tests for result inspection. Our approach collects branch coverage and data value bounds at runtime, and then mines implication relationships between branches and constraints of data values as potential test oracles after running all the tests. Our approach then selects only tests that violate the mined test oracles for result inspection. Experimental results show that our approach can more effectively reduce the number of tests for result inspection while revealing most of the faults.

Third, we mine test oracles of Web search engines from existing inputs (Web queries) and outputs (search results). It is labor-intensive to judge the relevance of search results for a large number of queries, and these relevance judgments may not be reusable since the Web data change all the time. To address this problem, we propose to mine implicit relationships between queries and search results, e.g., some queries may have fixed top 1 result while some may not, and some Web domains may appear together in top 10 results. We define a set of properties of queries and search results, and mine frequent association rules between these properties as potential test oracles. Experiments on major search engines show that our approach mines many high confidence rules that help to understand search engines and detect suspicious search results.

論文題目： 基於軟件數據挖掘的自動軟件測試

作者 ： 鄭吳傑

學校 ： 香港中文大學

學系 ： 計算器科學及工程學系

修讀學位： 哲學博士

摘要 ：

軟件測試是保證軟件系統的質量的主要技術。在軟件測試中，測試人員需要設計一個測試用例集合，包括測試輸入和期望結果，以在軟件發佈之前覆蓋大多數源代碼并找出大多數缺陷。對於現在的複雜系統來說，要手動生成這樣有效的測試用例集合是很有挑戰性的。自動軟件測試能較少人們在測試中所需要的大量勞動。但是，已有的自動測試輸入生成和測試結果評判技術往往需要手工提供的軟件規範，而這些規範往往難以獲得。

本論文提出了從軟件數據中挖掘軟件規範的辦法，來自動生成測試樣例和進行自動結果評判。可挖掘的軟件數據包括源代碼，運行記錄，和已有的輸出結果，等等。

首先，我們從源代碼中挖掘相關的函數來指導隨機單元測試的生成。給定一個被測函數，面向對象的單元測試生成中的一個主要步驟是找到能生成和修改被測函數參數的函數調用系列。我們提出了一種函數系列推薦算法來提高面向對象的隨機單元測試的效果。我們首先從代碼中挖掘出被測函數的相關函數，也就是那些可能修改被測函數訪問到的變量的函數。然後我們推薦使用那些包含這些相

關函數的短的函數系列來生成被測函數的輸入。採用這種辦法，生成的測試輸入能有更大的概率探索被測函數的不同行為。

其次，我們從未驗證的測試的執行記錄中挖掘控制分支和數據變量的潛在規則，用於選擇測試進行結果評判。我們先收集程序運行時的分支覆蓋率和數據值邊界。然後我們挖掘分支之間的蘊涵規則以及數據值的隱式限制，來作為潛在的結果評判標準。最後我們選出那些違背了挖掘出的評判標準的測試來進行手動的檢查。實驗結果表明，我們的方法能在發現大多數原測試集發現的缺陷的同時，有效的減少手動檢查的測試數目。

最後，我們從已有的測試輸入(Web 查詢)和輸出(搜索結果)中挖掘中 Web 搜索引擎的結果評判標準。對大量查詢的搜索結果進行人工評判需要大量的人力勞動，而且由於 Web 數據的持續的變化，這些評判結果很可能無法重用。爲了解決這個問題，我們提出挖掘查詢和搜索結果之前的隱式的規則來作為結果評判標準。例如，某些查詢有固定的排名第一的搜索結果而有些查詢則沒有。又例如某些 Web 域名在搜索結果中排名前十的時候，另一些域名也在同樣的查詢的搜索結果中排名前十。我們定義了一組查詢和搜索結果的屬性，然後挖掘這些屬性之前的關聯規則來作為結果評判標準。在主要的搜索引擎上面進行的實驗表明，我們的方法挖掘了很多高置信度的規則，這些規則能夠幫助我們理解搜索引擎并自動檢測出可疑的搜索結果。

Acknowledgement

I would like to express my greatest thanks to my supervisor, Prof. Michael R. Lyu, for his advice, understanding, and encouragement. He taught me a lot on research, and always provided insightful suggestions to my work. Without his support, this thesis would not have been possible. I also appreciate all the support and help from my mentor, Prof. Tao Xie. His great vision and enthusiasm to the research of software testing are very impressive and encouraging.

I would also like to thank my thesis committee members, Prof. Patrick Lee, Prof. Ada Fu, Prof. Shing Chi Cheung, and Prof. Elisa Baniassad for their helpful comments and suggestions on my work. I thank Prof. Steven Hoi for his guidance and discussions on the data mining techniques.

I would like to thank my colleagues and friends. I want to thank Qirun Zhang for his great help to my work. Thank Xiaoqi Li and Hao Ma for the enjoyable collaborations. Thank Yangfan Zhou, Jianke Zhu, Hongbo Deng, Xin Xin, Xinyu Chen, Zibin Zheng, Haiqin Yang, Junjie Xiong, Haixuan Yang, Kaizhu Huang, Zenglin Xu, Chao Zhou, Yilei Zhang, Kang Yu, and many others for their encouragement and kind help.

I am grateful to my mother, my wife, and my younger brother for their unlimited love and strong support.

To my family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Software permeates our daily life. Software failures can lead to serious consequences in safety-critical systems as well as in normal business. A recent report by National Institute of Standards and Technology found that software failures cost the US economy about $60 billion every year [97]. Therefore, it is critical to improve software reliability to ensure long-term software operations without failures [86].

Software testing has been the primary way for improving software reliability. Software testing typically includes three steps, generating test inputs, running test inputs, and verifying actual outputs (or properties). However, with the ever-growing size and complexity of software systems [60], software testing is becoming more and more challenging and costly. It costs billions of dollars and accounts for about 50% the cost of software development [92].

Automatic software testing reduces the laborious human effort in testing. Many test input generation tools [28, 36, 38, 63, 74, 103] can generate test inputs automatically from some kinds of specifications, such as finite state machines that model software's behaviors, or context-free grammars that de-

scribe the structure of inputs. Many testing frameworks can execute test inputs automatically. For example, JUnit [13] and googletest [5] can execute unit test inputs of Java and C++ programs, respectively. Selenium [15] and Watir [18] can execute test inputs that interact with Web applications. In regression testing, test selection techniques can reduce the execution time by executing only a small subset of tests that preserve code coverage or reveal faults previously. There are also some tools that can generate test oracles, i.e., expected test outputs/properties, to verify actual outputs automatically. Model-based testing tools can generate expected software states of a given set of actions as expected properties. Valgrind [95] can check memory errors using built-in memory usage models as test oracles.

However, the existing test input generation and test oracle generation techniques often require manually provided specifications, which may not be available or may be incomplete. For example, for unit testing, the specification of the inputs of each method is often missing due to the large amount of manual effort required. It is even more difficult to provide specifications for deriving the expected outputs of test inputs, except for state-based systems or for specific properties such as memory errors. When there are a large set of test inputs (e.g., automatically generated test inputs), or when the expected outputs change all the time (e.g., for Web search engines), it could be very costly to verify actual outputs manually.

This thesis investigates solutions to the problem of automatic software testing without manually provided specifications. Our research focuses on mining specifications from various kinds of software data to generate test inputs and to verify test outputs automatically. We mine various kinds of specifications from source code, dynamic execution traces, and test input/outputs, targeting at different context of software testing.

In this chapter, we briefly describe the main steps and techniques in au-

tomatic software testing, present the techniques in mining software data, and describe the contributions and the proposed approaches of the thesis, define the scope of the research in the thesis, and list the organization of the thesis.

1.2 Automatic Software Testing

Automatic software testing reduces the laborious human effort in the three steps of software testing: test input generation, test execution, and test output inspection. We next briefly describe the main techniques in these three steps.

1.2.1 Test Input Generation

Many test input generation tools [28, 36, 38, 63, 74, 103] can generate test inputs automatically from some kinds of specifications. Given a specification of the input parameters and the interesting values of each parameter, combinatorial test input generation techniques [36, 38] can generate test inputs that cover specified combinations of given parameter values. Given a finite state machine that describes some (usually functional) aspects of the software system under test, model-based testing techniques [28, 63] can generate test inputs in the form of traces: sequences of actions. A test suite is a collection of related traces (test inputs). Given a context-free grammar that specifies the input syntax of the software under test, such as compilers, grammar-based testing techniques [103, 74] can generate sentences (test inputs) of a given non-terminal (often the root element of the grammar) by traversing the grammar using either top-down approaches or bottom-up approaches. However, the specifications are often not available or may be incomplete.

Random testing uses the random mechanism to explore the possible input space, with few specifications about the inputs. For example, given the number of characters of a command line input, random testing approaches can

generate the characters one by one randomly. For unit testing, the inputs of a method is limited by the type system, and thus the effectiveness of random testing could be much higher. Many approaches [34, 37, 100, 101, 115] have been proposed to generate unit tests for object-oriented programs with different heuristics for exploring the input space. Although random testing is easy to apply, it may generate a large number of meaningless test inputs.

There are also much work on symbolic execution [119, 49, 112, 128], which generates tests based on the source code. Basically, the symbolic-execution approaches explores the control paths of the program, collects path conditions of the paths, and solves the constraints to create test inputs. Recent advances of powerful constraint solvers [70, 14, 40] have made it relatively easy to solve such path conditions. However, there are still two challenges of successful applications of symbolic execution: the difficulty of environment modeling (to understand the invocation of external calls for collecting path conditions) and the large space of control paths.

1.2.2 Test Execution

Many test frameworks can execute test inputs automatically. For example, JUnit [13] and GoogleTest [5] can execute unit test inputs of Java and C++ programs, respectively. For testing graphical user interfaces (GUIs), specific testing frameworks such as QuickTest Professional [6], Selenium [15], Watir [18] are required for running test inputs automatically.

In regression testing, test selection techniques can reduce the execution time by executing only a small subset of tests that have the highest probability of revealing faults. Test case prioritization and test suite minimization approaches share similar objectives with somewhat different handling of the test suites. Orso et al. [98] presented a technique for Java programs that selects every test case in a regression test suite that may behave differently

in the original and modified versions of the software, and yet scales to large systems. Rothermel et al. [106] and Elbaum et al. [43] evaluated a set of prioritization techniques for regression testing, focusing on the goal of increasing the likelihood of revealing faults earlier in the testing process. Hsu and Orso [58] proposed a test-suite minimization framework based on integer linear programming. Their approach can get optimal solutions for various minimization problems that involve any number of criteria.

1.2.3 Test Output Inspection

For inspecting the correctness of actual test outputs, we may generate test oracles, either expected outputs or expected properties, from some kinds of specifications. In model-based testing [28, 63], we can generate not only the sequences of actions from the finite state machines, but also the expected states after each action sequence. These states can be used as test oracles to check whether the implementation passes or fails a test case. Metamorphic testing [90, 30] relates multiple input-output pairs via well-defined relations, called metamorphic relations. A metamorphic relation is an existing or expected relation (some kind of specification) over a set of distinct inputs and their corresponding outputs, e.g. $sin(x) = sin(x + 2\pi)$ for the method $sin()$. These metamorphic relations can serve as test oracles to check a set of inputs and outputs. There are also some general specifications about the execution of software programs. For example, it is generally true that a program should not crash or hang for any given input [122]. The memory usage rules can also be used to check memory errors during execution of software programs [95].

There are often not suitable test oracles to check the functional correctness of general applications. To address this problem, the test selection for result inspection techniques can be applied to select a small subset of tests that are most likely to reveal the faults. The tests can be selected in the

way of maximizing some coverage criteria, such as code coverage criteria [59] or specification coverage criteria [31]. Dickinson et al. [41] used clustering analysis to partition executions based on structural profiles, and employed sampling techniques to select tests from clusters for result inspection. There are various approaches that mine operational models based on dynamic invariant detection for test selection. Xie and Notkin [127] developed an operational violation approach called Jov for unit-test selection and generation. They mined dynamic invariants using Daikon [45] from a set of manually written passing unit tests and selected newly generated tests that violated the dynamic invariants for result inspection.

1.3 Mining Software Data

There are a lot of valuable information in the software data. Many researches have mined the software data using different data mining techniques to support various software engineering tasks. The software data that are mined include source code, execution traces, bug reports, code revision history, documentations, mailing lists, etc. The data mining techniques that are used include classification, regression, clustering, frequent itemset mining, and so on. The software engineering tasks that are helped include automatic documentation, static defect analysis, debugging, maintenance, etc. An overview of the work in this direction can be found in the tutorial by Hassan and Xie [55].

There are many approaches that mine API usage patterns from the source code, which are then used for documentation or static defect analysis. Engler et al. [44] proposed an approach to extract programming rules using programmer-specified rule templates such as “function a must be paired with function b”. Li and Zhou [80] developed a tool named PR-Miner that uses frequent itemset mining to extract implicit API usage patterns from source

code without any specific templates. Zhong et al. [135] proposed a tool named MAPO that combines the frequent subsequence mining technique with the clustering technique to mine code snippets with respect to programming contexts. Thummalapenta and Xie [113] developed a tool named PARSEWeb that uses Google code search engine [10] to collect relevant code snippets and then mines the returned code snippets to find code snippets for a given target object type.

There are many approaches that mine specifications or locate bugs based on execution traces. Ernst et al. [45] proposed to mine dynamic invariants from passing tests and develop a tool named Daikon. Daikon defines a set of templates for generating candidate dynamic invariants, e.g. a precondition `x < y` at entry to a procedure f. The candidate invariants are evaluated on dynamic traces and a candidate is immediately discarded once it is violated. Jones et al. [68] develop the Tarantula tool that provides a mapping from testing results to the bug-relevant probabilities of statements and visualizes them in the source-code display. Liblit et al. [81] propose a low-overhead sampling infrastructure for gathering predicate information and adopt regularized logistic regression to locate non-deterministic bugs.

There are also many approaches that mine fault-prone module prediction modules based on bug history. Complexity measures including lines of code, McCabe complexity, Halstead complexity, etc. have been widely used as bug predictors [75, 88]. Ostrand et al. [99] presented a case study and suggested that revision history was informative for fault prediction. Nagappan [93] uses Principal Component Analysis to combine the complexity measures and the fault and modification history measures. In particular, their study suggested that predictors obtained from one project can also be significant for new, similar projects. Kim et al. [71] proposed an approach called change classification that determine whether a new software change is more similar to prior

buggy changes, or clean changes.

It is also beneficial to mine other software data. Murphy-Hill et al.[91] studied how programmers refactor using four data sets spanning more than 240,000 tool-assisted refactorings. Ruthruff et al. [107] built logistic regression models that predict whether a static analysis warning is accurate and actionable. Their experience with the static defect analysis tool FindBugs [3] showed that the models based on metrics from the warnings and implicated code are effective in reducing the spurious false positive warnings of FindBugs.

1.4 Contributions and Approaches

Automatic software testing reduces the laborious human effort in the three steps of software testing: test input generation, test execution, and test output inspection. Our research focuses on automatic test input generation and test output inspection. Ideally, if there are specifications for valid inputs and the expected outputs, test input generation and test oracle generation can be done easily. However, such specifications are often not available. There are two possible reasons for the missing of specifications. First, it could be costly to write specifications for a large number of methods or interfaces. This fact poses difficulties for automatic unit testing or integrated testing. Second, it is often difficult to write general specifications to generate the expected output of a specific input. For some software systems such as network protocols, finite state machines can be used to map a specific input (an action sequence) to the expected state. However, for many other software systems, the expected outputs are more complex and the mapping from an input to the expected output is hard to describe.

The contribution of this thesis is to improve the effectiveness of automatic software testing by mining specifications from software data. Although there

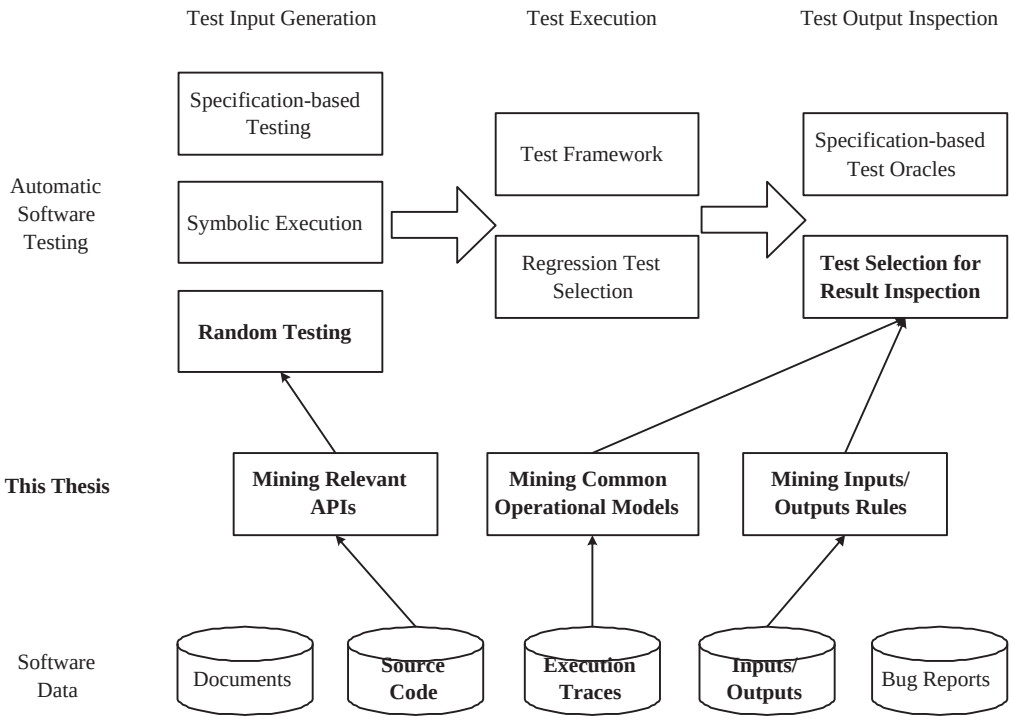


Figure 1.1: Overview of the work in this thesis

has been some existing work on mining specifications [23, 45, 56], these approaches do not target at using the mined specifications to help automatic software testing. In this thesis, we consider the data available in different scenarios of software testing, and propose new approaches to mine specifications from these data. The mined specifications are then used to generate test inputs and to verify test outputs. Figure 1.1 presents an overview of the work in this thesis. We have mined relevant APIs, common operational models, and input/output rules from the source code, execution traces, and existing inputs/outputs, respectively. We have used the mined specifications to help the tasks of random unit-test generation and test selection for result inspection. All the proposed approaches have been implemented and evaluated on

a set of software programs [132, 134, 133].

The main contributions of this thesis can be further described as follows:

1. **Mining Relevant APIs to Guide Random Unit-Test Generation**

   In this work, we mine relevant APIs from source code to guide random unit-test generation. Given a method under test (MUT), a key component of object-oriented unit-test generation is to find method-call sequences that create and mutate desired inputs. Previous work cannot find desired sequences effectively due to the large search space of possible sequences. We present a MUT-aware sequence recommendation approach, called RecGen, to improve the effectiveness of random object-oriented unit-test generation. RecGen mines relevant APIs for the MUT, i.e., methods that may mutate object fields accessed by the MUT, from the source code. It then recommends short sequences that mutate object fields accessed by the MUT to generate the inputs. We have implemented RecGen in Java and evaluated it on three libraries. The results show that RecGen can improve the code coverage and fault-revealing capability over previous random testing tools.

2. **Mining Common Operational Models to Select Tests for Result Inspection**

   In this work, we mine common operational models from execution traces of unverified tests to select tests for result inspection. In automatic testing, especially test generation in the absence of specifications, a large amount of manual effort is spent on test-result inspection. Test selection can reduce this effort by selecting a small subset of tests that are likely to reveal faults. A promising test selection approach is to dynamically mine operational models as potential test oracles and then select tests that violate them. Existing work adopting this approach mines opera-

tional models from passing tests using dynamic invariant detection. In this work, we propose to mine common operational models, which are often but not always true in all observed traces, from a (potentially large) set of unverified tests. Specifically, our approach collects branch coverage and data value bounds at runtime and then mines implication relationships between branches and constraints of data values as potential operational models after running all the tests. Our approach then selects tests that violate the mined common operational models for result inspection. We have evaluated our approach on a set of programs, compared with previous code-coverage-based, clustering-based, dynamic-invariant-based, and random selection approaches. The experimental results show that our approach can more effectively reduce the number of tests for result inspection while revealing most of the faults.

3. **Mining Test Oracles of Web Search Engines from Existing Inputs and Outputs**

   In this work, we mine test oracles of Web search engines from existing inputs (Web queries) and outputs (search results). Web search engines have major impact in people's everyday life. It is of great importance to test the retrieval effectiveness of search engines. However, it is labor-intensive to judge the relevance of search results for a large number of queries, and these relevance judgments may not be reusable since the Web data change all the time. In this work, we propose to mine test oracles of Web search engines from existing search results. The main idea is to mine implicit relationships between queries and search results, e.g., some queries may have fixed top 1 result while some may not, and some Web domains may appear together in top 10 results. We define a set of properties of queries and search results, and mine frequent association rules between these properties as test oracles. Experiments on major

search engines show that our approach mines many high confidence rules that help to understand search engines and detect suspicious search results.

1.5 Scope

The approaches presented in this thesis focus on automatic software test input generation and (pseudo) test oracle generation. We do not target at automatic test execution or other software testing activities such as test case management and maintenance.

Although the proposed approaches are designed to be general for a variety of software bugs, in this thesis we limit our scope to functional correctness or program robustness, but not other quality attributes such as performance and security.

1.6 Outline

The rest of this thesis is organized as follows.

- Chapter 2 reviews the background and related work of automatic test input generation, test selection for result inspection, and mining software data. The test adequacy criteria, which are used to guide test generation and to evaluate test suites, are also discussed.
- Chapter 3 describes the proposed techniques of mining relevant APIs from source code to guide random unit-test generation.
- Chapter 4 presents the proposed techniques of mining common operational models, including control rules and data rules, from execution traces of unverified tests to select tests for result inspection.

- Chapter 5 describes the proposed techniques of mining test oracles of Web search engines from existing inputs (Web queries) and outputs (search results).
- Chapter 6 summarizes this thesis and proposes some directions to be explored in future work.

□ **End of chapter.**

Chapter 2

Background and Related Work

In this chapter, we first discuss the main test adequacy criteria that are used to guide test generation and to evaluate test suites. We then describe the main techniques of automatic test input generation and test selection for result inspection.

2.1 Test Adequacy Criteria

A test adequacy criterion provides a measurement of test-suite quality and can be used to guide test generation [136]. There are mainly three kinds of coverage criteria: mutation coverage, code coverage, and specification based coverage.

2.1.1 Code Coverage

Code coverage describes the degree to which the source code of a program has been tested. Various kinds of code coverage criteria have been proposed, such as control-flow coverage criteria [59] and data-flow coverage criteria [47].

The main control-flow coverage criteria includes:

1. **Function coverage**: The percentage of functions that have been called

by the test suite.

2. **Line coverage**: The percentage of lines that have been executed by the test suite.

3. **Decision coverage**: The percentage of edges (such as branches in IF and CASE statements) that have been executed by the test suite.

4. **Condition coverage**: The percentage of possible results of boolean sub-expressions (each boolean sub-expression can be evaluated to true or false) that have been evaluated by the test suite.

5. **Path coverage**: The percentage of paths in the program that have been explored by the test suite. In practice, the number of possible paths in a non-trivial program can be very large, and there is no need to achieve 100% path coverage.

Different from control-flow coverage criteria that are based on the control-flow graph, data-flow coverage criteria are based on the definitions and uses of variables (more exactly, memory locations) [61]. A definition (def) of a variable is that a variable is assigned a value. A use of a variable is that the value of the variable is used (referenced). A def-use (DU) association for a variable is a pair consisting of a def and a use of the variable, such that there is a control-flow path in the code from the def to the use on which there is no intermediate redefinition or un-definition of the variable. A test case exercises a particular def-use association if the test case executes the def and subsequently execute the use. The DU coverage is then the percentage of possible def-use associations that are exercised by the test suite.

2.1.2 Mutation Coverage

The number of faults revealed by a test suite is also a good indicator of the quality of the test suite. However, it is in general impossible to know

the exact number of faults in a software program. To evaluate the fault-revealing capability of a test suite, people often use mutations of the original program with seeded faults [24, 46, 110]. Typical mutation operators include statement deletion, operator replacement, variable replacement, and so on. A test suite is said to kill a mutant if the test suite detects the fault in the mutant. Mutation coverage is then the percentage of faults detected by the test suite.

2.1.3 Specification-based Coverage

Specification based coverage criteria specify the percentage of testing requirements identified in a specification that have been exercised by the test suite [28, 63, 74, 103]. This kind of coverage criteria are highly related to the specification languages used, such as finite state machines, context-free grammars, or more complex specifications defined using advanced specification languages.

Model-based Coverage Criteria

In model-based testing [28, 63, 118], the specification refers to a finite state machine (FSM) that describes the systems' states and the changes of states. Several model-based coverage criteria can then measure how well a test suite covers the model. Some common coverage criteria include:

1. **State coverage**: The percentage of FSM states that are visited during the test execution.

2. **Transition coverage**: The percentage of FSM transitions that are traversed during the test execution.

3. **Transition-pairs coverage**: The percentage of pairs of adjacent transitions in the FSM that are traversed during the test execution.

4. **Paths coverage**: The percentage of paths that are traversed during the test execution. The all-paths criterion corresponds to exhaustive testing of the control structure of the FSM.

Among them, state coverage is a simple and popular coverage criterion. However, the resulting test suite may not test many behaviors. Full transition-pairs coverage and all-paths coverage are much expensive to achieve or even impossible. Transition coverage is often a suitable target to aim for when generating tests from FSM models.

Grammar-based Coverage Criteria

The inputs of many software programs are well structured, such as compiler inputs, files of a specific format. The structure of these inputs can be specified using context-free grammars [74]. Several grammar-based coverage criteria can then measure how well a test suite covers the grammar.

Purdom proposed to use rule coverage as a criterion for testing grammars [103]. A test case is said to cover a grammar rule if that rule is used at least once in deriving that test case. While this criterion is intuitive, experimental studies suggest that it cannot well assure fault detection capability [57]. To address this problem, Lammel proposed to use context-dependent rule coverage, where the context in which a rule is covered is taken into consideration [74]. The definitions of more kinds of grammar-based coverage criteria can be found in [77].

Specification-language-based Coverage Criteria

While the context-free grammar can specify the syntax of many kinds of inputs, it cannot specify many implicit constraints that are context-sensitive, such as the scope rules of many programming languages. Many specification languages are designed to specify such constraints [20, 31]. Based on the ex-

pressions of a specification language, many coverage criteria can be defined. For example, Chang and Richardson [31] developed customized coverage criteria for the following constructs of Sun Microsystems' Assertion Definition Language (ADL): conditional expressions, implication expressions, relational expressions, equality expressions, normally expressions, group expressions, unchanged expressions, and quantified expressions.

2.2 Test Input Generation

2.2.1 Random Test Input Generation

Random testing uses the random mechanism to explore the possible input space, with few specifications about the inputs. For example, given the number of characters of a command line input, random testing approaches can generate the characters one by one randomly. But such random exploration could generate a large number of meaningless test inputs. For unit testing, the inputs of a method is limited by the type system, and thus the effectiveness of random testing could be much higher. Csallner and Smaragdakis [37] developed an automatic robustness testing tool named JCrasher for Java code. JCrasher uses a parameter-graph to represent the parameter space of the program's methods, where there is an edge from type A to method m if m can be used to produce a value of type A. For each method m declared by a given class C, JCrasher generates test inputs based on the parameter graph randomly.

It is possible to improve the effectiveness of random testing using static analysis. Thummalapenta et al. [115] proposed an approach named MSeqGen to leverage the information of how method calls are used for sequence generation. MSeqGen extracts sequences that are relevant to the receiver or arguments of a MUT from code bases. It then uses those sequences to

assist random test generation and systematic test generation. Although this approach is a promising one, its performance relies on API client code, which may not be available in testing code under development or may not be sufficient enough for recommending methods or sequences for most of the MUTs.

On the other hand, there have been many approaches that improve the effectiveness of random testing using dynamic analysis. Pacheco and Ernst [100] developed a tool named Eclat that takes execution feedback into consideration. Eclat generates a method sequence randomly and classifies it as normal operation, fault-revealing, or illegal based on dynamic invariants mined from sample executions. Only the inputs classified as normal operation are used for generating new sequences afterwards. Pacheco et al. [101] extended Eclat and developed another tool named Randoop. Randoop further prunes duplicate sequences and uses API contracts, e.g., o.equals(o) should return true, as oracles to identify illegal sequences. Ciupa et al. [34] proposed an adaptive random testing approach named ARTOO for object-oriented software. ARTOO defines some measure of object distance. It then generates candidate inputs randomly, and at every step selects from them the one that is furthest away from the already used inputs. Evolutionary testing [102, 117] uses genetic algorithm to generate new test cases based on the coverage information of previously executed test cases. In this approach, test cases are described by chromosomes, which include information on which objects to create, which methods to invoke and which values to use as inputs. The genetic algorithm then mutates them with the aim of maximizing a given coverage measure.

2.2.2 Symbolic Execution

Symbolic execution [49, 112, 119, 128] is a type of automatic white-box testing technique. Basically, the symbolic-execution approaches explores the control paths of the program, collects path conditions of the paths, and solves the

An example program

```
  int m(int x, int y) {
1   if(x>5)
2     if(x*y=10)
3       printf("find it");
4   return 1;
  }
```

Paths	Path Conditions	Inputs
1, 4	!(x>5)	x=0, y=0
1, 2, 4	x>5 && !(x*y=10)	x=6, y=0
1, 2, 3, 4	x>5 && x*y=10	x=10, y=1

Figure 2.1: An example of symbolic execution

constraints to create test inputs. Figure 2.1 shows an example program, the three control paths in it, the path conditions, and some possible inputs by solving the path conditions. For example, for the control path that consists of Lines 1 and 4, we can collect the path condition $!(x > 5)$ automatically. Recent advances of powerful constraint solvers [70, 14, 40] have made it relatively easy to solve such path conditions, and may return x=0, y=0 as the input (there may be many inputs that satisfy the constraints and the solvers would randomly return one of them). We can get the inputs for exercising the other two control paths similarly.

A major challenge of symbolic execution is that it needs to understand every statement, including the invocation of external calls, so as to collect the path conditions. However, it is difficult and costly to model the behaviors of all methods, especially when the source code of many methods is unavailable, for large or complex programs. To address this problem, Godefroid et al. [49] proposed a tool named DART that incrementally generates test inputs by combining concrete and symbolic execution. DART generates path constraints during a concrete execution. DART then modifies the path

constraints and solves them, if feasible, to generate further test inputs that would direct the program along alternative paths. If it is not feasible to solve the modified constraints, DART simply substitutes the symbolic values in the constraints with random concrete values. Similarly, Sen et al. [112] developed a tool named CUTE to extract and solve constraints generated for a program that has dynamic data structures using pointer operations.

2.2.3 Specification-based Test Input Generation

When there are specifications about the inputs, behaviors, and outputs of the software systems, specification-based testing techniques can generate test inputs based on these specifications. We next describe three kinds of such test generation techniques: combinatorial test input generation, model-based test input generation, and grammar-based test input generation.

Combinatorial Test Input Generation

Combinatorial test input generation is to generate test inputs that cover some kinds of combinations of given parameter values [36, 38, 50, 73]. The assumption is that many failures result from an interaction between components. The input of combinatorial testing tools is a specification that describe the input parameters and the interesting values of each parameter. Given some combination requirement, techniques from experimental design are used to generate a small number of test inputs that satisfy the given requirements.

The most common combination requirement is the pair-wise coverage (also known as 2-wise). 100% pair-wise coverage requires that every possible pair of interesting values of any two parameters are included in some test case. A natural extension of pair-wise (2-wise) coverage is t-wise coverage, which requires every possible combination of interesting values of t parameters be included in some test case in the test suite. t-wise coverage is formally defined

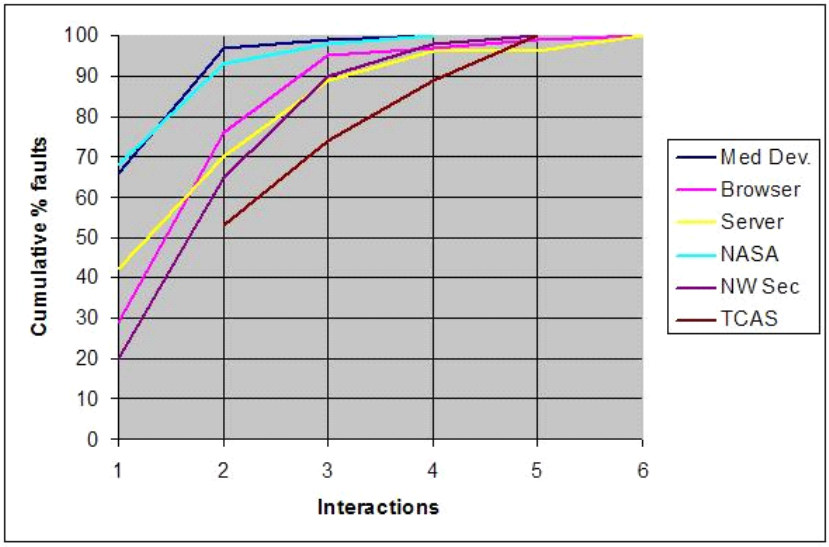


Figure 2.2: Number of parameters involved in triggering software faults

by Williams and Probert [124]. A special case of t-wise coverage is N-wise coverage, where N is the number of all parameters of the inputs. N-wise coverage requires all possible combinations of all interesting values of the N parameters be included in the test suite. As shown in Figure 2.2[1], empirical evidence suggests that most failures are triggered by the interaction of two parameters, while many of the rest can be triggered by the interaction of three or four parameters. Therefore, it is often sufficient to test the interactions of two to four parameters.

Note that the same test case may cover more than one unique combination of values. Most combinatorial testing techniques exploit this properties to generate a small number of tests that satisfy the given combination requirement heuristically. Basically, the more a test input covers combinations of values that have not been covered previously, the more likely it will be selected into the test suite [50]. In practice, there may be some constraints of the interactions between certain parameters. Many tools [38] also allow the

[1]This figure is adopted from www.csrc.nist.gov/groups/SNS/acts/ftfi.html.

testers to specify these constraints and prune invalid combinations during the test generation.

Model Based Test Input Generation

Model-based testing generates test inputs based on a given model that describes some (usually functional) aspects of the software system under test. The model is often (translated to) a finite state machine, which describes the possible actions and states of the system. Based on a given model, model based testing techniques can generate tests in the form of traces: sequences of actions. A test suite is a collection of related traces (test inputs).

Depending on the complexity of the software under test and the corresponding model, the number of possible traces (test inputs) can be very large. For finding appropriate test cases, i.e. traces that refer to a certain model-based coverage requirement, the exploration of the model has to be guided. Many test generation techniques have been proposed for this purpose [63, 118].

The generated tests, however, are often in the abstract level and could not run against the implementation of the software directly. A test harness, which is called a test stepper [63], is used to connect the abstract tests to the implementation.

Grammar Based Test Input Generation

Grammar-based testing generates test inputs based on a given context-free grammar [74, 103]. As described above, context-free grammar can be used to specify the input syntax of many software programs, such as compilers, data files of a specific format, exchange data, etc. Given a non-terminal (often the root element) of a context-free grammar, one way of generating a sentence (test input) of the non-terminal is as follows: randomly pick a production rule, generate the sentences of non-terminals/terminals used by

the production rule, and then combine these sentences into a whole sentence. Similar to model-based test input generation, the exploration of the grammar can be guided in the way of maximizing a given grammar-based coverage criterion quickly.

One can also explore the grammar to generate sentences (test inputs) of a given non-terminal systematically. Lammel [74] proposed a bottom-up approach, called Geno, to generate sentences in the order of increasing depth. *depth* denotes the largest distance from the non-terminal to a terminal in its constructing tree. Geno first generates sentences of depth 1, and then generates sentences of depth 2 by combining sentences of depth 1, and so on. It also provides other control mechanisms, such as recursion control, for the test generation.

2.3 Test Output Inspection

2.3.1 Specification-based Test Oracle Generation

Some test oracles may be generated from specifications automatically. For example, in model-based testing [28, 63], we can generate not only the sequences of actions from the finite state machines, but also the expected states after each action sequence. These states can be used as test oracles to check whether the implementation passes or fails a test case.

A test oracle is not limited to specify the expected output (properties) of a single input. Instead of relating an output to its input, metamorphic testing relates multiple input-output pairs via well-defined relations, called metamorphic relations. A metamorphic relation is an existing or expected relation (some kind of specification) over a set of distinct inputs and their corresponding outputs for multiple executions of the target method [30]. For example, consider the mathematic method *sine*. The expected results of x

and $x + 2\pi$ are the same, i.e., $sin(x) = sin(x + 2\pi)$.

Despite the application specific specifications, there are also some general specifications about the execution of software programs that can generate or serve as test oracles. For example, it is generally true that a program should not crash or hang for any given input [122]. There are also general rules about the memory usages that can be used to check memory errors during execution of software programs [95].

2.3.2 Test Selection for Result Inspection

Test selection is to select a small subset of tests from the original test suite without reducing much of the fault-revealing capability. There are two possible objectives of test selection, to reduce the execution time and to reduce the result inspection time. In regression test selection, the main concern is often the test execution time. The existing approaches are based on static analysis, fault history, and the dynamic execution traces. In test selection for result inspection, the main concern is the time of inspecting the results, where the expected outputs are unknown or even unstable. The existing approaches are mainly based on dynamic execution traces.

Coverage-based Test Selection

The coverage-based approach attempts to cover as many elements (e.g. lines of code) of a given type as the original test suite with as few test cases as possible [76]. Selecting a minimal-size, coverage-maximizing subset of a test suite is an instance of the set-cover problem, which is often solved using a greedy approximation algorithm. On each of its iterations, the greedy algorithm selects the test that covers the largest number of elements not covered by the previously selected tests. This approach is based on the assumption that many software faults and their resulting failures can be revealed simply

by exercising such elements, regardless of other factors.

Various kinds of code coverage criteria have been proposed for test selection, such as control-flow testing criteria [59] and data-flow testing criteria [47]. Hutchins et al. [61] reported an experimental study investigating the effectiveness of control-flow and data-flow testing criteria. Their results suggested that test sets achieving high code coverage levels usually showed significantly better fault-detection capability than randomly chosen test sets of the same size. However, the results also indicated that 100% code coverage alone is not a reliable indicator of the effectiveness of a test set. Leon et al. [76] evaluated the effectiveness of complex information flow criteria, which model indirect control/data dependencies between instructions or objects, for test selection. Their results suggested that test sets maximizing complex information flow criteria revealed more faults than test sets maximizing block coverage with substantial additional cost. In some subjects, the profiles could not be generated due to memory constraints.

There also exist a number of black-box coverage criteria for test selection. In partition testing [92], a test input domain is divided into subdomains based on some criteria, and then developers can select one or more representative inputs from each subdomain. When *a priori* specifications are provided for a program, Chang and Richardson [31] used specification coverage criteria to select a candidate set of test cases that exercise new aspects of the specification.

Clustering-based Test Selection

Dickinson et al. [41] used clustering analysis to partition executions based on structural profiles, and employed sampling techniques to select executions from clusters for result inspection. The approach uses agglomerative hierarchical clustering to cluster the tests, and then selects one test from each

cluster randomly. The main assumption of this approach is that a significant number of failures are isolated in clusters of small size. Dickinson et al. [42] further proposed a failure-pursuit sampling approach to enhance the efficiency in finding failures. Moreover, Leon et al. [76] evaluated the effectiveness of clustering analysis based on complex information flow criteria. Their results suggested that the effectiveness of the clustering analysis did not depend strongly on the type of used profiling.

Behavioral-model-based Test Selection

There are various approaches that mine dynamic invariants for test selection. A dynamic invariant is a property that holds at specific program points, e.g., a precondition `x < y` at entry to a procedure f. Ernst et al. [45] proposed to mine dynamic invariants from passing tests and develop a tool named Daikon. Daikon defines a set of templates for generating candidate dynamic invariants, including preconditions, postconditions, and loop invariants. The candidate invariants are evaluated on dynamic traces and a candidate is immediately discarded once it is violated. Any new test that violates the mined dynamic invariants may reveal faults and deserve manual inspection.

Harder et al. [54] proposed the operational difference approach to select tests based on Daikon. Their approach repeatedly adds new tests if they violate the invariants of the previously selected tests. Xie and Notkin [127] developed an operational violation approach called Jov for unit-test selection and generation. They mined operational models using Daikon from a set of manually written passing unit tests and selected automatically generated test inputs that violated the operational models. Pacheco and Ernst [100] developed a similar tool named Eclat, which further distinguishes illegal and fault-revealing inputs with some strategies. Hangal and Lam [52] developed DIDUCE that extracts operational models dynamically from long-running

program executions. DIDUCE reports all detected violations at runtime and gradually relaxes invariants to allow for new behavior. Due to the limited number of existing passing tests or previously selected tests, the mined dynamic invariants of these approaches could be noisy and thus many model violations could be false positives.

There are also some approaches that build software behavioral models and then classify failures using both failing and passing tests. Haran et al. [53] built behavior models of failures in-house using random forests, so as to help classify remotely-collected execution data. Michail and Xie [89] collected bug and "not bug" reports that consist of event histories from GUI application users. They then built a distance weighted nearest neighbor learner to help users avoid bugs in GUI applications. Bowring et al. [27] classified program executions based on Markov models. They trained the models incrementally in an active-learning paradigm to help test-plan development.

Finally, Xie and Notkin [128] developed an approach for automatically identifying special and common unit tests based on algebraic models. Their approach selects a test as a special test if the test exercises a certain program behavior that is not exhibited by most of other tests. But it is applicable only in object-oriented unit testing.

2.4 Mining Software Data

2.4.1 Mining Source Code

There are many approaches that mine information from source code, which may come from the software under test, other related software programs, or online code search engines. The approaches often mine API usage patterns from the source code, which are then used for documentation or static defect analysis.

Engler et al. [44] proposed an approach to extract programming rules using programmer-specified rule templates such as "function a must be paired with function b". Li and Zhou [80] developed a tool named PR-Miner that uses frequent itemset mining to extract implicit API usage patterns from source code without any specific templates. Zhong et al. [135] proposed a tool named MAPO that combines the frequent subsequence mining technique with the clustering technique to mine code snippets with respect to programming contexts. Thummalapenta and Xie [113] developed a tool named PARSEWeb that uses Google code search engine [10] to collect relevant code snippets and then mines the returned code snippets to find code snippets for a given target object type.

Robillard [105] proposed an approach named Suade to automatically rank program elements that are potentially interesting to a developer investigating source code, based on the topology of structural dependencies in a program. Saul et al. [108] extended Suade to recommend API functions using random walks. Long et al. [85] developed a tool Altair that ranks related API methods for a given query, which is a method, according to pair-wise overlap, i.e., the share data that they both access. Inoue et al. [62] proposed an approach of ranking software components, called Component Rank, based on analyzing actual use relations among the components and propagating the significance through the use relations. Thummalapenta and Xie [114] proposed an approach named SpotWeb to detect hotspots, i.e. API classes and methods that are frequently reused, in a given framework by mining code examples gathered from the web. Zhang et al. [131] augmented the call graph with control flow analysis to capture the significance of the caller-callee linkages in the call graph, so as to retrieve the relevant APIs more effectively.

Li et al. [79] also used frequent sequence mining to detect copy-pasted code in large software including operating systems, and then detect copy-

paste related bugs. Thummalapenta and Xie [115] mined exception-handling rules, which describe expected behavior when exceptions occur, as sequence association rules in the source code.

2.4.2 Mining Execution Traces

There are many approaches that mine information from execution traces of the software. The main tasks are mining specifications and fault localization.

Ernst et al. [45] proposed to mine dynamic invariants from passing tests and develop a tool named Daikon. Daikon defines a set of templates for generating candidate dynamic invariants, e.g. a precondition `x < y` at entry to a procedure f. The candidate invariants are evaluated on dynamic traces and a candidate is immediately discarded once it is violated. Hangal and Lam [52] developed DIDUCE that extracts operational models dynamically from long-running program executions. Ammons et al. [23] proposed an approach to summarize the frequent interaction patterns in program execution as probabilistic finite state automata (PFSA). Lo and Khoo [84] improved the quality of mining results by filtering erroneous execution traces and clustering related traces. Yang et al. [129] proposed an approximate inference algorithm and other heuristics to mine interesting temporal API rules effectively. Gabel and Su [48] proposed a general specification mining framework to learn complex temporal properties from execution traces. Henkel and Diwan [56] presented an automatic tool for extracting algebraic specifications, e.g., for a stack s and an object o, `pop(push(s,o).state) .state=s`, from Java classes.

Statistical debugging applies statistical analysis on execution traces to locate the bugs automatically. The basic idea is that the more a statement or a predicate is covered in failing tests and the less in passing tests, the more possible it is to be the bug. Agrawal et al. [21] use set operations of the execution slices of different runs to determine the faulty statements. Jones

et al. [68] develop the Tarantula tool that provides a mapping from testing results to the bug-relevant probabilities of statements and visualizes them in the source-code display. Liblit et al. [81] propose a low-overhead sampling infrastructure for gathering predicate information and adopt regularized logistic regression to locate non-deterministic bugs. Liu et al. [81] propose a statistical model-based approach named SOBER that considers how frequent a predicate is evaluated as true in each run. Nainar et al. [94] enrich the predicates by adding compound predicates derived from two atomic predicates. Jiang and Su [65] mine bug predictors from atomic predicates using SVM and Random Forests, and then link them through clustering and control flow graph traversal. Baah *et al.* [25] propose a probabilistic program dependence graph that models the behaviors of predicates conditioned on their structural predecessors. Wong et al. [125] uses an RBF neural network on statement coverage to locate program bugs. Wang et al. [121] refine code coverage of test runs using control- and data- flow patterns prescribed by different fault types. Reps et al. [104] defines a path spectrum for application to the Year 2000 problem. Dallmeier et al. [39] investigate the differences of call sequences to pinpoint the defective classes.

2.4.3 Mining Other Software Data

There are many other software data that are valuable for mining, such as bug reports, code revision history, documents.

Fault-prone module prediction approaches mine the bug history to predict faulty modules. These approaches first define specific measures that may relate to fault-proneness, and then build prediction models based on these measures. Complexity measures including lines of code, McCabe complexity, Halstead complexity, etc. have been widely used [75, 88]. Basili et al. [26] validated the usefulness of the object-oriented design measures (CBO,

RFC, LCOM, DIT, NOC, and WMC) [33]. Ostrand et al. [99] presented a case study and suggested that revision history was informative for fault prediction. Nagappan [93] uses Principal Component Analysis to combine the complexity measures and the fault and modification history measures. In particular, their study suggested that predictors obtained from one project can also be significant for new, similar projects. Schroeter et al. [109] showed that import relations in Eclipse are good for predicting bugs. Kim et al. [71] proposed an approach called change classification that determine whether a new software change is more similar to prior buggy changes, or clean changes. In this manner, change classification predicts the existence of bugs in software changes.

Example work on mining other software data is listed as follows. Murphy-Hill et al.[91] studied how programmers refactor using four data sets spanning more than 13,000 developers, 240,000 tool-assisted refactorings, 2,500 developer hours, and 3,400 version control commits. Ruthruff et al. [107] built logistic regression models that predict whether a static analysis warning is accurate and actionable. Their experience with the static defect analysis tool FindBugs [3] showed that the models based on metrics from the warnings and implicated code are effective in reducing the spurious false positive warnings of FindBugs.

□ **End of chapter.**

Chapter 3

Unit-Test Generation via Mining Relevant APIs

In this chapter, we describe our work on mining relevant APIs from source code to guide random unit-test generation. Given a method under test (MUT), a key component of object-oriented unit-test generation is to find method-call sequences that create and mutate desired inputs, including the receiver and the parameters. Previous work cannot find desired sequences effectively due to the large search space of possible sequences. To address this issue, we present a MUT-aware sequence recommendation approach called RecGen to improve the effectiveness of random object-oriented unit-test generation. Unlike existing random testing approaches that select sequences without considering how a MUT may use inputs generated from sequences, RecGen analyzes object fields accessed by a MUT and recommends a short sequence that mutates these fields. RecGen first mines relevant APIs for the MUT, i.e., methods that may mutate object fields accessed by the MUT, from the source code. It then recommends short sequences that contain relevant APIs of the MUT to generate the inputs. We have implemented RecGen in Java and evaluated it on three libraries. The results show that RecGen can

improve the code coverage over previous random testing tools.

3.1 Problem and Motivation

Unit testing is one of the most commonly used techniques to assure high quality of software systems. A primary objective of unit testing is to achieve high structural coverage such as statement coverage. To this end, a method under test (MUT) needs to be executed with specific inputs. For object-oriented programs, the desired inputs including the receiver and arguments of a MUT are often objects that have specific values in their fields. As directly modifying object fields may violate class invariants, it is necessary to employ method-call sequences (in short as *sequences*) to create and mutate objects to generate desired inputs. Therefore, a key component of object-oriented unit-test generation is to find method-call sequences that generate desired inputs of a MUT.

There are two main approaches to generate desired method-call sequences: random sequence generation and bounded-exhaustive sequence generation. Random sequence generation approaches generate sequences randomly from the whole space. Csallner and Smaragdakis [37] developed a random testing tool called JCrasher to generate sequences randomly based on a parameter graph, which represents the parameter space of the program's methods. It is challenging to find desired sequences randomly since there is a very large space of possible sequences. Pacheco et al. [101] proposed a feedback-directed random test generation approach and implemented it in a tool called Randoop. Randoop guides random test generation by pruning out invalid or redundant sequences, which are classified based on the execution results, for generating new sequences. But there are still a lot of valid and unique sequences that are not relevant for achieving new code coverage of a MUT. Adaptive Random Testing (ART) [35, 83] aims to select inputs evenly across

the input space. Basically, the ART approaches define some distance for values of primitive types or objects of classes. They generate candidate inputs randomly, and then at every step select a candidate input that is the farthest away from the already used inputs. But when applied to object-oriented programs, the weights associated with object fields are usually not rigorous and they are not suitable for various MUTs. Bounded-exhaustive sequence generation approaches [111, 119, 126] generate sequences exhaustively up to a small bound of sequence length. However, generating desired objects, including the receiver and arguments, often requires longer sequences beyond the small bound that can be handled by the bounded-exhaustive approaches.

To address the challenges faced by these existing sequence-generation approaches, we propose a MUT-aware sequence recommendation approach called RecGen to improve the effectiveness of random object-oriented unit-test generation. Unlike existing random testing approaches that select sequences without considering how a MUT may use inputs generated from sequences, RecGen analyzes object fields accessed (i.e., read and write) by a MUT and recommends a short sequence that mutates these fields. The rationale is that a sequence that mutates object fields accessed by a MUT has a higher chance to explore different behaviors of the MUT. RecGen first builds the Abstract Syntax Tree (AST) from the code under test and identifies the methods that mutate object fields accessed by a MUT. These methods are called *relevant methods* of the MUT. RecGen then recommends a short sequence that consists of *relevant methods* of the MUT for generating the receiver or an argument. In particular, RecGen assigns weights to sequences based on the number of *relevant methods* they include and their sizes. The chance of a sequence to be selected is proportional to the ratio of its weight over the total weight of all sequences. Finally, for MUTs whose test generation keeps failing (i.e., the resulting sequence is duplicate or throws uncaught

exceptions), RecGen recommends a set of sequences, which cover the MUT's all *relevant methods* appearing in the candidate sequences, for each input of the MUT. Such a batch-mode recommendation further improves the chance of generating desired inputs with limited cost.

To illustrate the idea of RecGen, we present an example MUT named `openDatabase`, which belongs to the `Environment` class in Berkeley DB Java Edition [12], in Figure 3.1. The MUT calls `setupDatabase`, which subsequently calls `getAllowCreate` to check the field `allowCreate` of the `DatabaseConfig` argument. To achieve high structural coverage of the MUT (and the private methods called by it) needs various inputs. In particular, to cover the code under the true branch of `dbConfig.getAllowCreate` in the `setupDatabase` method and create a new database, the field `allowCreate` of the `DatabaseConfig` argument of the MUT should have the `true` value, which by default has the `false` value and can be mutated by invoking `DatabaseConfig.setAllowCreate`. We also need an object of `Environment` for the receiver, while the `Transaction` argument can be null and the `String` argument can be any string. An example of desired sequences is shown in Figure 3.2.

Assume that we have generated a set of sequences that produce objects of `Environment` and objects of `DatabaseConfig`. When selecting previously generated sequences for the `DatabaseConfig` argument, existing random testing tools have a low chance to select a sequence that includes the `DatabaseConfig.setAllowCreate` method. On the other hand, RecGen identifies that `DatabaseConfig.setAllowCreate` is a *relevant method* of the MUT since it mutates the field `allowCreate` that is accessed by the MUT. RecGen then recommends short sequences with the `DatabaseConfig.setAllowCreate` method and assigns large weights to them, and thus improves the chance to generate desired new sequences to create a new database. There are also other *relevant methods* of the MUT and recommending a single sequence cannot guarantee the success of sequence

```
// Environment.java
public synchronized Database openDatabase(
                      Transaction txn,
                      String databaseName,
                      DatabaseConfig dbConfig)
throws DatabaseException {
    checkHandleIsValid();
    checkEnv();
    try {
        if (dbConfig == null) {
            dbConfig = DatabaseConfig.DEFAULT;
        }
        Database db = new Database(this);
        setupDatabase(txn, db, databaseName,
                      dbConfig,
                      false,
                      false,
                      envImpl.isReplicated());
        return db;
    } catch (Error E) {
        envImpl.invalidate(E);
        throw E;
    }
}

private void setupDatabase(..., DatabaseConfig
                           dbConfig, ...)
throws DatabaseException {
    ...
    if (databaseExists) {
        ...
    } else {
        ...
        /* No database.
           Create if we're allowed to. */
        if (dbConfig.getAllowCreate()) {
            ...
        }
    }
    ...
}

// DatabaseConfig.java
public boolean getAllowCreate() {
    return allowCreate;
}

public void setAllowCreate(boolean allowCreate) {
    this.allowCreate = allowCreate;
}
```

Figure 3.1: The `openDatabase` MUT and related code

```
File home = new File(myFilePath);
EnvironmentConfig envConfig=new EnvironmentConfig();
envConfig.setAllowCreate(true);
Environment env = new Environment(home,envConfig);
DatabaseConfig dbConfig = new DatabaseConfig();
dbConfig.setAllowCreate(true);
Database newDb=env.openDatabase(null,"db",dbConfig);
```

Figure 3.2: A desired sequence of `openDatabase`

generation. If the test generation of the MUT keeps failing, RecGen recommends a set of sequences that cover the MUT's all *relevant methods*, including `DatabaseConfig.setAllowCreate`, for the `DatabaseConfig` argument. Such a batch-mode recommendation further improves the chance of generating desired sequences.

We have implemented RecGen in Java. RecGen consists of two components: a static analysis component, which identifies *relevant methods* of the MUTs, and a test generation component, which generates tests for the MUTs. The static analysis component is built upon the Eclipse JDT Compiler [2] and the test generation component is built upon another random testing tool Randoop [101]. RecGen accepts a set of methods under test and outputs the generated test cases in test files for JUnit. We have evaluated RecGen on three libraries: a database library Berkeley DB [12], a data structure library JDSL [11], and a scientific computation library JScience [7]. The experimental results show that RecGen can improve the code coverage over previous random testing tools [37, 83, 101]. The results also show that RecGen finds a real bug in Berkeley DB Java Edition, which has been confirmed by its developers.

The rest of this chapter is organized as follows. Section 3.2 reviews related work. Section 3.3 presents the proposed random object-oriented unit-test generation approach with MUT-aware sequence recommendation. Section 3.4 de-

scribes the empirical studies and results. Section 3.5 discusses the limitations of our approach and future work. Section 3.6 concludes the chapter.

3.2 Related Work

Our work belongs to the category of object-oriented unit-test generation and is inspired by the API recommendation work. In this section we discuss the related work on these two topics. More related work on automatic test input generation can be found in Chapter 2.

3.2.1 Object-Oriented Unit-Test Generation

Object-oriented unit-test generation contains two parts: generating test inputs and generating test oracles. Test input generation further contains two components: generating method-call sequences and generating primitive arguments in sequences. RecGen addresses the test input generation problem and focuses on sequence generation.

Method-call Sequence Generation

There are two main approaches to generate desired method-call sequences: random sequence generation and bounded-exhaustive sequence generation. Random sequence generation approaches generate sequences randomly from the whole space. Csallner and Smaragdakis [37] developed a random testing tool named JCrasher to generate sequences randomly. Pacheco et al. [101] proposed a feedback-directed random test generation approach and implemented it in a tool Randoop. Adaptive Random Testing (ART) [35, 83] approaches such as ARTGen [83] select inputs evenly across the input space. These random sequence generation approaches cannot generate desired sequences effectively from the large space of possible sequences. Bounded-exhaustive sequence generation approaches [111, 119, 126] generate sequences

exhaustively up to a small bound of sequence length. However, generating desired objects, including the receiver and arguments, often requires longer sequences beyond the small bound that can be handled by the bounded-exhaustive approaches.

Recently, Thummalapenta et al. [115] proposed an approach named MSeqGen to generate desired sequences for a MUT using the client code. MSeqGen extracts sequences that are relevant to the receiver or arguments of a MUT from the client code. It then uses those sequences to enhance random test generation and dynamic symbolic execution approaches. The performance of MSeqGen relies on the client code, which may not be available in testing code under development or may not be sufficient enough for recommending sequences for most of the MUTs. Alternatively, RecGen assists random test generation by recommending sequences based on object-field-access information of the methods in the application under test, without requiring any client code.

Primitive Argument Generation

The primitive arguments of a method-call sequence can be generated randomly or systematically via symbolic execution. Random approaches [35, 37, 83, 101] generate primitive arguments from all possible values or a set of predefined values randomly. Symbolic execution approaches [49, 111, 116, 119, 128] generate primitive arguments systematically to cover all feasible paths in the MUTs. The symbolic-execution approaches execute method sequences with symbolic parameters (unspecified arguments), builds path constraints on the parameters, and solves the constraints to create actual test inputs with concrete parameters. For large or complex programs, it is computationally intractable to precisely maintain and solve the constraints required for test generation. Dynamic symbolic execution approaches [111, 116] address this

issue by substituting the symbolic parameters in the constraints with random concrete values. RecGen generates primitive arguments randomly, yet it is possible to combine the sequence recommendation approach of RecGen with the symbolic execution approaches. We plan to investigate this approach in our future work.

3.2.2 API Recommendation

RecGen recommends sequences for a MUT based on its *relevant methods*. The idea is inspired by the API recommendation work that helps to recommend related APIs for a given method.

Some approaches mine API usage patterns from example code repositories. Engler et al. [44] proposed an approach to extract programming rules using programmer-specified rule templates such as "function a must be paired with function b". Li and Zhou [80] developed a tool named PR-Miner that uses frequent itemset mining to extract implicit API usage patterns from source code without any specific templates. Zhong et al. [135] proposed a tool named MAPO that combines the frequent subsequence mining technique with the clustering technique to mine code snippets with respect to programming contexts. Instead of finding code snippets from a repository (with a limited set of snippets), Thummalapenta and Xie [113] developed a tool named PARSEWeb that uses Google code search engine [10] to collect relevant code snippets and then mines the returned code snippets to find code snippets for a given target object type. These approaches, however, cannot be used for testing code under development, as we may not have example code available.

Some other approaches recommend API methods mainly based on structural dependencies of the library code. Mandelin et al. [87] developed a tool named Prospector that answers queries of how to create certain Java types.

It mines code snippets from both API type signatures and client code. Robillard [105] proposed an approach named Suade to automatically rank program elements that are potentially interesting to a developer investigating source code, based on the topology of structural dependencies in a program. Saul et al. [108] extended Suade to recommend API functions using random walks. Long et al. [85] developed a tool called Altair to recommend APIs for C programs. Altair recommends related API methods for a given method according to pair-wise overlap, i.e., the shared data that they both access. Our approach is inspired by Altair and we extend it to recommend sequences for a given query, which contains a method under test and its receiver or one of its arguments.

3.3 Our Approach

In this section, we present the proposed approach of random object-oriented unit-test generation. We first present the overview of the approach. We then describe the MUT-aware sequence recommendation technique in detail. Finally, we describe test oracles and test execution.

3.3.1 Overview

RecGen accepts a set of methods under test, generates test inputs, adds test oracles, and outputs the generated test cases in test files for JUnit. Alternatively, the users can specify a set of classes to test. RecGen adds all the public methods in these classes into the list of methods to test. By default, RecGen filters out subclasses of `java.lang.Exception`, since these classes are often used to handle exceptional cases and may corrupt the program states for further testing. RecGen then repeats the following steps to test the MUTs up to a given time limit.

- Select a MUT randomly.
- For each input (the receiver or arguments) of the MUT
  - If the input is a primitive value, randomly select a primitive value from a fixed pool of values.
  - If the input is an array of primitive values, randomly construct an array with a set of primitive values selected from a fixed pool of values.
  - If the input is an object, select a sequence (or a set of sequences for MUTs whose test generation keeps failing) from existing valid sequences using MUT-aware sequence recommendation.
- Generates a new sequence (or a set of sequences for MUTs whose test generation keeps failing) to test the MUT by concatenating selected sequences with the MUT at the end.
- Add test oracles to the new sequence.
- Execute the new sequence and process it according to execution results; only valid and unique sequences are used for further sequence generation.

Before describing the main steps in detail, we next explain a bit more on the mechanism of object generation in RecGen. For an input that is of a reference type, RecGen selects an existing sequence to generate an object as the input. The pool of existing sequences is initially empty. At that time only test generation for constructors or static methods may succeed. But as the test generation proceeds, new valid sequences are added to the pool of existing sequences and the size of the pool grows quickly. The number of existing sequences that can generate objects of the desired type of an input often becomes large. RecGen does not create objects of reference types in an active mode, i.e., by invoking constructors of classes recursively (Note that the

arguments of constructors may also be objects). This mechanism is adopted because if an object of a type is difficult to be generated randomly, such as the `Database` class shown in Figure 3.1, there are probably some constraints on arguments of methods that return an object of the type (including the constructors and other classes' methods). In this case, to create objects in an active mode also has a low chance of success, and could take a high cost.

3.3.2 MUT-aware Sequence Recommendation

Given a MUT and its receiver or one of its arguments, RecGen first selects previously generated sequences that produce the desired (compatible) type of objects. This selection can be done by recording the execution results of previously generated sequences. Among these candidate sequences, RecGen recommends the sequences that mutate object fields accessed by the MUT. We first describe how RecGen identifies *relevant methods* of a MUT, i.e., the methods that mutate object fields accessed by the MUT. We then describe how RecGen recommends a single sequence for most MUTs or a set of sequences for MUTs whose test generation keeps failing, based on *relevant methods*.

Identifying Relevant Methods

Various approaches [80, 85, 87, 135] recommend related APIs for a given method. Since we target at the test-generation problem, we are interested in methods that may affect the execution of a given MUT by mutating object fields accessed by the MUT. RecGen identifies *relevant methods* for the methods under test by analyzing all the code of the application under test. Basically, RecGen checks whether two methods may access the same object fields. If so, we consider the two methods relevant; otherwise, we consider them non-relevant. Let $N(f)$ denote the set of object fields that a method f

may access. The relevance between a method g and f is defined as follows.

$$relevance(f,g) = \begin{cases} 1, & if\ |N(f) \cap N(g)| > 0 \\ 0, & otherwise \end{cases}$$

$relevance(f,g)$ indicates whether g may affect the execution of f, and vice versa.

To compute the set of object fields that a method may access, RecGen first analyzes which object fields a method accesses directly in its method body. RecGen uses the Eclipse JDT Compiler [2] to build the Abstract Syntax Tree (AST) from the code under test, and then extracts the direct field-access information. A method may access an object field by calling other methods, such as the `openDatabase` method illustrated in Figure 3.1. Therefore, RecGen merges the set of object fields accessed by a given method and the methods that the given method calls. RecGen extracts the call graph of the code under test from the AST. For a method f, RecGen searches the call graph for the methods that are called by f or called by the methods called by f. Let $M(f)$ denote the search result. RecGen then merges object fields accessed directly by f or by $M(f)$ into the set $N(f)$.

RecGen identifies *relevant methods* of all the methods in the application under test and saves the results in advance. Given a MUT, RecGen queries its *relevant methods* from the saved results. This technique allows RecGen to identify *relevant methods* of the MUTs quickly during test generation.

Recommending a Single Sequence

Correlation Weights. RecGen recommends sequences for each input of a MUT based on the *relevant methods*. Basically, RecGen prefers sequences that include more *relevant methods* of the MUT and fewer other methods. Therefore, we define the correlation of a sequence to a MUT as the percentage of *relevant methods* of the MUT in the sequence. More formally, the

correlation of a sequence seq to a method f is defined as follows.

$$weight\_corr(seq, f) = \sum_{i=1}^{n} relevance(seq_i, f)/n$$

where seq_i is the ith method in the sequence and n is the total number of method calls in the sequence.

The metric indicates how likely a sequence mutates object fields that a MUT may access. We use this metric as one kind of weights of the sequences. The larger the weight is, the higher chance it has in exploring different behaviors of the MUT by using the sequence to generate the input of the MUT.

Size Weights. RecGen also prefers to use shorter sequences to generate inputs of a MUT. In many cases, it is sufficient to use short sequences, whose search space is much smaller than the space of all the sequences, to generate desired objects. However, it is still possible that long sequences are necessary to generate desired objects. Instead of setting a fix threshold of the size (i.e., length) of sequences to search, RecGen recommends shorter sequences by assigning size weights to sequences as follows.

$$weight\_size(seq, f) = 1/n$$

where n is the total number of method calls in the sequence.

By assigning large weights to shorter sequences, RecGen searches the space of short sequences more thoroughly. At the same time, a long sequence that is necessary to generate desired objects could be recommended by the correlation weights.

Overall Weights. RecGen normalizes the correlation weights and the size weights of sequences separately such that each kind of weights of all sequences sum up to 1. In particular, for the correlation weight, our approach divides the correlation weight of each sequence for a MUT f by the sum of all the correlation weights. RecGen employs the same normalization for the

size weights of sequences. RecGen then combines the weights equally. The overall weight of a sequence for a MUT f is calculated as follows:

$$weight(seq, f) = (weight\_corr(seq, f) + weight\_size(seq, f))/2$$

RecGen randomly selects a sequence to generate the input of a MUT based on the overall weights of sequences. In particular, the chance of a sequence to be selected is proportional to the ratio of its weight over the total weights of sequences.

Priority of Receiver Sequences. Finally, RecGen recommends to use the receiver sequence, i.e., the sequence that is used to generate the receiver, to generate arguments, if possible. This technique is based on the rationale that the arguments of a MUT often have correlations with the receiver. For example, consider the `Vector.remove(Object)` method in `java.util.Vector`. To cover most of the code of this method, we need to generate objects that are contained in the receiver as well as objects that are not contained in the receiver for the `Object` argument. Suppose that we have selected a sequence *seq* that generates a vector v as the receiver, and now we would like to select a sequence to generate the argument. Selecting sequences for the argument randomly has a low chance to generate an object that is contained in the receiver v. On the other hand, the sequence *seq* that creates and mutates the receiver v is more likely to produce an object contained in v. Therefore, RecGen prefers to select the receiver sequence to generate the arguments. In particular, given a MUT and an argument to generate, RecGen checks whether the receiver sequence can produce objects of the argument type. If so, RecGen selects the receiver sequence to generate the argument with a predefined probability, which is 0.5 by default.

Recommending a Set of Sequences

Although RecGen has improved the chance of generating a desired input with a single sequence, it cannot guarantee to generate desired inputs with only a few attempts for some MUTs. RecGen employs a batch-mode recommendation to further improve the chance with limited cost.

If the test generation of a MUT keeps failing, i.e., returning invalid or duplicate sequences, RecGen identifies that this MUT is difficult to test. Given an input of such a MUT, RecGen first selects previously generated sequences that produce the desired (compatible) type of objects as candidate sequences. It then recommends a set of sequences, which cover all *relevant methods* of the MUT appearing in the candidate sequences, for the input. In particular, RecGen ranks the candidate sequences in the descending order of their weights as defined earlier. From top to down, RecGen recommends a sequence if the sequence covers a *relevant method* (of the MUT) that is not covered by the previous recommended ones. For *relevant methods* that have only a boolean argument, RecGen distinguishes the values (true/false) of the argument. RecGen does not distinguish the values of other types due to the large value space. For each argument, RecGen also recommends the receiver sequences that can produce objects of the argument type. Finally, RecGen explores all the combinations of the sequences for the inputs to generate new sequences.

The set of recommended sequences cover all *relevant methods* of the MUT appearing in the candidate sequences, and thus has a high chance to generate the desired inputs. In addition, the recommended sequences are only a small portion of all candidate sequences and they cost less to run. Therefore, the batch-mode recommendation further improves the chance of generating desired inputs with limited cost. RecGen employs the batch-mode recommendation for only the MUTs whose test generation fail k times successively

(k is 5 by default).

3.3.3 Test Oracles and Test Execution

RecGen adds test oracles to the test inputs of a MUT to form test cases. A test oracle specifies the expected effect of a method on the program states including invariant properties. When a test case is executed, test oracles decide whether the test case passes or fails. RecGen employs test oracles that express generic properties of Java programs. In particular, RecGen adopts the contracts used in Randoop [101], which are listed as follows.

- `o.equals(o)` returns true
- `o.equals(o)` throws no exception
- `o.hashCode()` throws no exception
- `o.toString()` throws no exception

The programmers may specify additional test oracles, including domain-specific ones. A test oracle can be added by implementing a `Contract` interface and specifying the pre-conditions, post-conditions, and invariant properties.

When a new sequence is generated, it is executed immediately and checked with the test oracles. If the execution of a new sequence returns normally and produces new objects, the new sequence is added to the pool of existing sequences and used for further test generation. Otherwise, it is outputted in test files for JUnit but not used for further test generation.

3.4 Experiments

In this section, we present the experimental studies to evaluate the effectiveness of RecGen in unit-test generation. We first describe the experimental

setup, including subjects, measurements, compared approaches, and experimental environments. We then present the results, including code coverage and the number of generated tests, of RecGen and other approaches in unit-test generation. Finally we present an example to illustrate how RecGen helps to find real bugs in applications.

3.4.1 Experimental Setup

Subjects

To evaluate our approach, we conduct experiments on three Java libraries: a database library Berkeley DB [12], a data structure library JDSL [11], and a science computation library JScience [7].

Berkeley DB (BDB) is a computer software library that provides a high-performance embedded database. Berkeley DB can support thousands of simultaneous threads of control or concurrent processes manipulating databases as large as 256 terabytes, on a wide variety of operating systems. Berkeley DB Java Edition comprises a pure Java database.

JDSL is a data structure library in Java. It implements fundamental data structures and algorithms such as priority queues, sorting and searching algorithms, minimum spanning trees, and graph travels.

JScience is a Java library for scientific computing (math, physics, astronomy, economics, etc.). It supports symbolic calculations and analyses, quantum and natural in physics, linear algebra, different types of numbers, and many other tasks in scientific computing.

The characteristics of the three subject applications are summarized in Table 3.1.

Table 3.1: Subject programs

Library	#Classes	#Methods	#LOC
Berkeley DB	248	4371	96148
JDSL	102	1060	19636
JScience	65	1386	18290

Measurement

To evaluate the effectiveness of our approach in unit-test generation, we use two metrics. The first metric is the statement coverage, i.e., the percentage of statements that are covered by executing the generated tests. The higher the coverage is, the more thoroughly the applications are tested and the more confidence we can gain in the testing. The second metric is the number of the generated tests. This metric indicates the potential efforts required to execute and to check the results of the generated tests.

Compared Approaches

We compare RecGen with previous random test generation approaches, including JCrasher [37], Randoop [101], and ARTGen [83]. JCrasher uses undirected random testing approach, Randoop applies feedback-directed random testing approach, and ARTGen employs adaptive random testing approach.

Randoop, ARTGen, and RecGen use a "timelimit" parameter to limit the time for test generation. We run these tools on each subject library for two minutes (the default time limit of Randoop). JCrasher uses a "depth" parameter to limit the space of possible sequences for test generation. We run JCrasher on each subject library with the default depth. When testing Berkeley DB, which interacts with the file system, we provide a seeding sequence that specifies an empty directory and an empty file for Randoop, ARTGen,

and RecGen. JCrasher does not provide the functionality of using seeding sequences, so we do not provide the seeding sequence for it.

Experimental Environments

We conduct the experiments on a 2.80GHz Intel(R) Core (TM)2 PC with 3GB physical memory, running Ubuntu 9.04. We apply a Java code coverage tool EclEmma [9] to collect the statement coverage of executing the generated tests.

3.4.2 Results

Statement Coverage

Tables 3.2, 3.3, and 3.4 show the statement coverage results of RecGen and other approaches on Berkeley DB, JDSL, and JScience, respectively. Only lines that are not comments, blanks, standalone braces, or parenthesis are counted. The tables show the statement coverage results for all packages of the subject applications and the total statement coverage of each application, i.e., the ratio of the number of covered statements to the total number of statements in each application. In total, the tests generated by JCrasher achieve 11.0%, 23.2%, and 37.7% statement coverage on Berkeley DB, JDSL, and JScience, respectively. The tests generated by Randoop achieve 37.4%, 45.5%, and 56.1% statement coverage on Berkeley DB, JDSL, and JScience, respectively. The tests generated by ARTGen achieve 24.2%, 35.2%, and 56.0% statement coverage on Berkeley DB, JDSL, and JScience, respectively. The tests generated by RecGen achieve 48.4%, 58.9%, and 64.9% statement coverage on Berkeley DB, JDSL, and JScience, respectively. Note that it is often impractical to cover 100% of the statements. For example, it is difficult to trigger the exception-handling code.

JCrasher achieves low statement coverage because it cannot search the

Table 3.2: Statement coverage (%) on Berkeley DB (LOC: lines of code)

Package	#LOC	JCrasher	Randoop	ARTGen	RecGen
com.sleepycat.je	4755	9.8	36.6	32.5	**44.3**
com.sleepycat.je.cleaner	2850	1.6	30.6	8.5	**52.8**
com.sleepycat.je.config	764	89.1	**95.9**	95.5	95.2
com.sleepycat.je.dbi	4401	10.4	40.0	27.9	**53.4**
com.sleepycat.je.evictor	456	0.0	**11.2**	0.2	8.6
com.sleepycat.je.incomp	318	0.3	**23.3**	0.3	16.0
com.sleepycat.je.jca.ra	278	0.0	0.0	0.0	0.0
com.sleepycat.je.jmx	441	49.2	58.3	57.8	**64.6**
com.sleepycat.je.latch	215	27.0	74.9	67.4	**76.7**
com.sleepycat.je.log	3789	9.6	36.3	15.1	**49.6**
com.sleepycat.je.log.entry	366	15.0	47.5	29.8	**65.6**
com.sleepycat.je.recovery	1954	7.0	33.9	7.8	**34.4**
com.sleepycat.je.tree	4398	9.3	34.8	22.0	**47.4**
com.sleepycat.je.txn	2608	6.6	37.6	22.1	**52.5**
com.sleepycat.je.util	1564	5.9	22.9	22.5	**34.6**
com.sleepycat.je.utilint	678	19.3	63.7	50.7	**64.5**
Total	29835	11.0	37.4	24.2	**48.4**

Table 3.3: Statement coverage (%) on JDSL (LOC: lines of code)

Package	#LOC	JCrasher	Randoop	ARTGen	RecGen
jdsl.core.algo.sorts	91	24.2	**48.4**	24.2	**48.4**
jdsl.core.algo.traversals	26	0.0	0.0	0.0	0.0
jdsl.core.api	62	69.4	**93.5**	90.3	25.8
jdsl.core.ref	2497	26.1	49.4	39.4	**67.4**
jdsl.core.util	60	**30.0**	6.7	6.7	1.7
jdsl.graph.algo	602	8.7	40.0	20.1	**41.4**
jdsl.graph.api	46	47.8	**89.1**	82.6	37.0
jdsl.graph.ref	541	15.7	29.6	25.9	**51.9**
Total	3925	23.2	45.5	35.2	**58.9**

Table 3.4: Statement coverage (%) on JScience (LOC: lines of code; GEO.COOR: geography.coordinates, MATH: mathematics)

Package	#LOC	JCrasher	Randoop	ARTGen	RecGen
org.jscience.	396	3.0	4.5	**4.8**	**4.8**
org.jscience.economics.money	55	43.6	87.3	85.5	**96.4**
org.jscience.GEO.COOR	667	17.4	**61.9**	60.9	21.9
org.jscience.GEO.COOR.crs	198	52.5	**64.1**	61.6	61.1
org.jscience.MATH.function	692	32.8	32.7	37.3	**39.6**
org.jscience.MATH.number	1683	68.1	83.1	79.3	**86.1**
org.jscience.MATH.vector	1551	22.0	39.8	46.1	**82.8**
org.jscience.physics.amount	614	36.5	67.4	57.8	**70.5**
org.jscience.physics.model	60	58.3	96.7	96.7	**100**
Total	5916	37.7	56.1	56.0	**64.9**

whole space of possible sequences effectively. In particular, JCrasher excludes void-returning methods, e.g., the `DatabaseConfig.setAllowCreate` method, to reduce the search space. But void-returning methods, which may mutate objects, are critical in generating desired inputs. Randoop improves the statement coverage much over JCrasher by using execution feedback to prune illegal and duplicate sequences. The feedback information helps to reduce the search space without excluding any methods. ARTGen performs comparably with Randoop on JScience, but worse than Randoop on Berkeley DB and JDSL. The poor performance of ARTGen may be caused by non-rigorous weights associated with object fields. ARTGen also generates fewer tests within the time limit due to the high cost of calculating the distances between objects that are complex data structures. In addition, ARTGen prefers to use different objects of each argument for different tests. But for some arguments of the MUTs, similar objects may be required and combined with different objects of other arguments. RecGen improves the statement coverage much over Randoop. By focusing on *relevant methods*, RecGen improves the chance of generating desired sequences for each attempt and further improves the chance by covering all *relevant methods*. In some cases, RecGen improves the statement coverage by both focusing on sequences containing *relevant methods* and preferring the receiver sequences, the latter of which is important for testing data structures such as collections. We next provide an example to illustrate such scenarios.

Figure 3.3 shows a MUT, called `finishTime`, in the `DFS` class of the `jdsl.graph.algo` package. The `DFS` class implements the depth-first search (DFS) algorithm for graphs. After invoking a method `execute` to perform the DFS algorithm on a graph, a record called `FINISH_TIME` is set for each vertex that has been visited. The `finishTime` MUT is to get the `FINISH_TIME` record for a given vertex. To test the functionality of getting the `FINISH_TIME` record of a vertex that has

```
// a MUT
public Integer finishTime(Vertex v) {
    return (Integer)v.get(FINISH_TIME);
}

// a desired sequence for the MUT
jdsl.graph.algo.DirectedDFS var0 =
          new jdsl.graph.algo.DirectedDFS();
jdsl.graph.ref.IncidenceListGraph var1 =
          new jdsl.graph.ref.IncidenceListGraph();
java.lang.String var2 = "";
jdsl.graph.api.Vertex var3 =
          var1.insertVertex((java.lang.Object)var2);
var0.execute((jdsl.graph.api.InspectableGraph)var1);
java.lang.Integer var5 = var0.finishTime(var3);
```

Figure 3.3: The `finishTime` MUT and a desired sequence

been visited, the depth-first search should be performed on some graph and then a vertex in the graph should be taken as the argument of the MUT. Figure 3.3 shows one of the desired sequences (`DirectedDFS` is a subclass of `DFS`). RecGen identifies `execute` as a *relevant method* of the `finishTime` MUT, and recommends a sequence that includes `execute` for the receiver of `finishTime`. RecGen also prefers to use an object of `Vertex` that is produced by the sequence that generates the receiver for the `Vertex` argument. Such an object of `Vertex` is more likely to have been visited by the `execute` method. Thus RecGen successfully generates a desired sequence that helps to test the functionality of the `finishTime` MUT.

For some packages, RecGen achieves lower code coverage than Randoop. For example, the package `jdsl.core.api` of JDSL consists of mainly simple classes that are subclasses of `java.lang.Exception`. RecGen filters out the methods of these classes by default, and thus achieves lower code coverage for the package.

Table 3.5: The number of generated tests

Library	JCrasher	Randoop	ARTGen	RecGen
Berkeley DB	100077	10061	12267	13805
JDSL	99964	13957	2409	12806
JScience	99977	4362	3607	13413

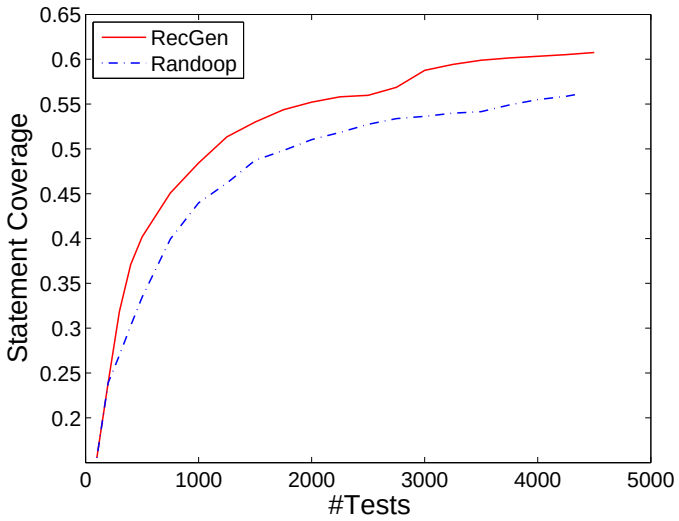


Figure 3.4: Code coverage w.r.t. #tests on JScience

Number of Generated Tests

We list the number of generated tests for the three subjects in Table 3.5. As shown in the table, JCrasher generates a very large number of tests which, however, achieve low code coverage. Such a result is due to that JCrasher does not use execution feedback to guide sequence generation. It could generate a lot of sequences quickly but most of them are invalid. Randoop generates much fewer tests than JCrasher and improves the code coverage by pruning invalid sequences. ARTGen generates fewer tests than Randoop in JDSL, but at the cost of lower code coverage. RecGen generates similar numbers of tests as Randoop in Berkeley DB and JDSL, but generates much more tests than Randoop in JScience. To execute and to check the results of more tests require more efforts. This issue of a large number of tests, however, can be addressed by test selection techniques [61, 132]. We plan to investigate this issue in our future work. In any case, using the same number of tests as those generated by Randoop, RecGen can achieve a statement coverage of 60.7%, which is still higher than that of Randoop in JScience. Figure 3.4 shows the code coverage of executing the first k number of tests generated by RecGen and Randoop on JScience. The results show that the tests generated by RecGen are more effective in increasing the code coverage, since they contain sequences that mutate the object fields accessed by the MUTs.

A Bug Found

We have described the statement coverage results of RecGen. We next present an example to illustrate how our approach helps to find real bugs in the applications.

Figure 3.5 shows a test case generated by RecGen for Berkeley DB. The test case creates an object of the `DatabaseEntry` class, sets the `data` field of the object, and then sets the `size` field of the object. Finally, a test oracle is

```
com.sleepycat.je.DatabaseEntry var0 =
              new com.sleepycat.je.DatabaseEntry();
java.lang.Byte var1=new java.lang.Byte((byte)1);
java.lang.Byte var2=new java.lang.Byte((byte)123);
java.lang.Byte var3=new java.lang.Byte((byte)122);
byte[] var4 = new byte[] var1, var2, var3;
var0.setData(var4);
java.lang.Integer var6 = new java.lang.Integer(100);
var0.setSize(var6);
// Checks var0.equals(var0)
var0.equals(var0);
```

Figure 3.5: A test case generated by RecGen

added to check whether the object is equal to itself. Executing this test case throws an unexpected exception `ArrayIndexOutOfBoundsException`.

Figure 3.6 shows the code of the `DatabaseEntry` class. Checking the code of the `equals` method, we can find that the exception is caused by crossing the bound of the `data` field. While `size` is 100, `data` contains only 3 elements. The `equals` method does not expect such an inconsistency between `data` and `size`, and it contains no checks when accessing the elements of `data`. Therefore, there is a bug either in the `setSize` method or in the `equals` method. On one hand, if `size` is intended to represent the number of elements in `data`, it should be updated with `data` simultaneously. In this case, the class should not provide the `setSize` method, or at least, should not declare it as a public method. On the other hand, if `size` is allowed to have other meanings (which is confusing since in the `setData` method, `size` is always set as the number of the elements in `data`), the `equals` method (and possibly many other methods) should not use `size` to access the elements in `data`. We have reported the bug to the developers of Berkeley DB and the bug has been confirmed.

```
public class DatabaseEntry {
    private byte[] data;
    private int size = 0;
    ...

    public void setData(byte[] data) {
        this.data = data;
        offset = 0;
        size = (data == null) ? 0 : data.length;
    }

    public void setSize(int size) {
        this.size = size;
    }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof DatabaseEntry)) {
            return false;
        }
        DatabaseEntry e = (DatabaseEntry) o;
        if (partial || e.partial) {
            if (partial != e.partial ||
                   dlen != e.dlen ||
                   doff != e.doff) {
                return false;
            }
        }
        if (data == null && e.data == null) {
            return true;
        }
        if (data == null || e.data == null) {
            return false;
        }
        if (size != e.size) {
            return false;
        }
        for (int i = 0; i < size; i += 1) {
            if (data[offset + i] !=
                e.data[e.offset + i]) {
                return false;
            }
        }
        return true;
    }
}
```

Figure 3.6: The DatabaseEntry class in Berkeley DB

3.4.3 Threats to Validity

The threats to external validity primarily include the degree to which the subject programs are representative of true practice. We have evaluated our approach on only three libraries that are of medium to large size. These threats could be reduced by more experiments on wider types of subjects in future work. The threats to internal validity are instrumentation effects that can bias our results. Faults in our implementation and other random testing tools might cause such effects.

3.5 Discussions

3.5.1 Accuracy of Relevant Methods

Our current implementation of identifying *relevant methods* may not be totally accurate. For example, we have not distinguished read and write operations. Therefore, RecGen may identify a method that reads an object field accessed by a MUT as a *relevant method* of the MUT. Recommending sequences that consist of such a method does not help to improve the chance of generating desired inputs. We plan to reduce such noisy methods using finer-grained analysis. In addition, a method that is relevant to a MUT may forbid some object fields from being accessed by the MUT. For example, the method `close` of the `java.io.InputStream` closes an input stream and then the method `read` cannot read any data from the input stream. To avoid generating invalid sequences that invoke methods on an object after the object is "destroyed", we may identify such destroying methods using heuristics of naming conventions or manual annotations and assign negative weights to them.

3.5.2 Fault-revealing Capability of Generated Tests

We have shown that RecGen can generate test cases automatically to improve the structural coverage. Test cases that achieve high structural coverage often have a higher chance to reveal more bugs and improve the software reliability [32]. However, due to the difficulty of manually checking the results of a large number of tests and the lack of application-specific test oracles, we cannot fully exploit the fault-revealing capability of the generated tests currently. We can address this issue by using test selection approaches [61, 132] to reduce the number of tests for result inspection and adding application-specific test oracles. We plan to investigate the effectiveness of these two approaches in our future work. In addition, the current implementation of RecGen organizes the generated test cases in the order of time of test generation. We plan to improve the organization by putting test cases generated for the same methods or classes together. It is easier for programmers to check and modify test cases in this way.

3.5.3 Symbolic Execution with Sequence Recommendation

Our current approach integrates MUT-aware sequence recommendation with random test generation. It is also possible to integrate sequence recommendation with symbolic execution approaches [49, 111, 116, 119, 128]. While random test generation approaches sparsely sample a large portion of the state space, symbolic execution approaches often thoroughly sample a tiny, localized portion of the space. In particular, symbolic execution approaches are effective in finding suitable primitive arguments of a sequence skeleton (a method-call sequence with primitive arguments unspecified) to improve structural coverage of a MUT. But there is no approach that provides an effective guidance to the choice of sequence skeletons, which can greatly affect the performance of symbolic execution approaches. Our sequence recommen-

dation technique can help symbolic execution approaches by recommending sequence skeletons that contain the MUT and its *relevant methods*. We plan to investigate this direction in our future work.

3.6 Summary

Given a method under test (MUT), a key component of object-oriented unit-test generation is to find method-call sequences that create and mutate desired receiver or arguments. In this chapter, we propose a MUT-aware sequence recommendation approach, called RecGen, to improve the effectiveness of random object-oriented unit-test generation. Unlike existing random testing approaches that select inputs without considering how the MUT may use the inputs, RecGen recommends a short sequence that mutates object fields accessed by the MUT to generate the inputs. The rationale is that a sequence is more likely to exercise new behaviors of a MUT if it mutates object fields accessed by the MUT. In addition, for MUTs whose test generation keeps failing, RecGen recommends a set of sequences to cover all the methods that mutate object fields accessed by the MUT. This technique further improves the chance of generating desired inputs. We have implemented RecGen as an open source tool in Java. We also evaluate RecGen on three libraries. The results show that RecGen can improve the code coverage over previous random testing tools and it finds a real bug in Berkeley DB.

We plan to pursue several future directions for improving our approach. First, we plan to improve the accuracy of identifying *relevant methods*. Second, we plan to improve the fault-revealing capability of generated test suites. Third, we plan to conduct experiments on wider types of subjects. Finally, we plan to combine our sequence recommendation approach with symbolic execution approaches.

□ **End of chapter.**

Chapter 4

Test Selection via Mining Operational Models

In automated testing, especially test generation in the absence of specifications, a large amount of manual effort is spent on test-result inspection. Test selection for result inspection helps to reduce this effort by selecting a small subset of tests that are likely to reveal faults. A promising test-selection approach is to dynamically mine operational models as potential test oracles and then select tests that violate them. Existing work adopting this approach mines operational models based on dynamic invariant detection. In this chapter, we propose to mine common operational models, which are often but not always true in all observed traces, from a (potentially large) set of unverified tests. Specifically, our approach collects branch coverage and data value bounds at runtime and then mines implication relationships between branches and constraints of data values as potential operational models after running all the tests. Our approach then selects tests that violate the mined common operational models for result inspection. We have evaluated our approach on a set of programs, compared with previous code-coverage-based, clustering-based, dynamic-invariant-based, and random selection approaches.

The experimental results show that our approach can more effectively reduce the number of tests for result inspection while revealing most of the faults.

4.1 Problem and Motivation

It is labor-intensive to manually generate a large set of test inputs and verify their outputs. Recently, there have been various practical approaches on automatic test-input generation [101, 112, 119, 123]. However, test-result inspection still remains a largely manual task. Given *a priori* specification, developers can reduce this manual effort by selecting test inputs (in short as *tests*) using specification coverage criteria [31]. But it is uncommon to have *a priori* specification in practice. Sometimes developers may use general test oracles based on memory monitoring tools such as Valgrind [96]. But these oracles are limited in checking specific kinds of faults. It is highly demanded to develop practical test-selection techniques, which can select a small subset of tests that are likely to reveal faults.

A promising test-selection approach is to dynamically mine operational models as potential test oracles and then select tests that violate them. Existing approaches such as Jov [127] and Eclat [100] mine dynamic invariants using Daikon [45] from a set of manually written passing unit tests whose results are verified with manually written assertions. Due to nontrivial effort for writing the assertions, the number of these existing passing unit tests is often limited. Therefore, the mined dynamic invariants could be noisy and thus many model violations could be false positives. The operational difference approach [54] starts with an empty test suite and repeatedly adds new tests if they violate the invariants mined from the previously selected tests. As the number of previously selected tests is also limited, this approach faces the same problem of producing many false positives. DIDUCE [52] mines operational models from normal execution of long-running applications and

relaxes the models gradually. At the beginning of a program run, many presumed operational models may be violated and a violation that reveals a fault can be overwhelmed by the false-positive noise.

In this chapter, we propose to mine common operational models, which are often but not always true in all observed traces, from a (potentially large) set of unverified tests. A program that is not of poor quality should pass most of the tests. So the common operational models mined from a large set of unverified tests may be similar to the true operational models and their violations are likely to reveal faults. By using the information of all the unverified tests at hand, our approach can avoid the noise caused by a small number of data samples, without requiring a large set of verified tests. As a common operational model is not always true over the whole set of tests, the type of Daikon inference techniques does not work anymore. Alternatively, we may generate and collect all the potential models at runtime (instead of immediately discarding any violated potential models) and evaluate them after running all the tests. However, such an approach can incur high runtime overhead if Daikon-like operational models, which are in a large number, are used.

To mine common operational models efficiently, we propose an approach based on mining control rules and data rules. A control rule is an implication relationship between branches, and a data rule is an implicit constraint of the variable values. Specifically, our approach collects branch coverage and data value bounds at runtime using the Cooperative Bug Isolation (CBI) tools [82]. Our approach then mines control rules and data rules as potential operational models after running all the tests. The likelihood of a common operational model to be a true oracle is evaluated using the concept of confidence, i.e., the ratio of the number of tests that satisfy the model over the number of tests that can evaluate the model. When a model's confidence is not equal

```
1   not_text = (! (always_text | out_quiet)
              && memchr(bufbeg,'\0', buflim-bufbeg));
2   done_on_match += not_text;
3   out_quiet += not_text;
4   for (;;)
5   {
6       ...
7       nlines += grepbuf (beg, lim);
8       ...
9       if (nlines && done_on_match && !out_invert)
10          goto finish_grep;
11      ...
12  }
13  finish_grep:
    /* missing code: done_on_match -= not_text;
                     out_quiet -= not_text; */
14  if ((not_text & ~out_quiet) && nlines != 0)
15      printf(_("Binary file %s matches\n"),filename);
```

Figure 4.1: Faulty code of the *grep* program

to 1, the higher the model's confidence is, the more suspicious its violations are in revealing a fault. Finally, our approach selects a small subset of tests that violate the mined common operational models for result inspection.

To illustrate what a common operational model may look like and how its violation is useful for indicating faults, we present an example in Figure 4.1. The example is a faulty version of *grep* 2.3, which is downloaded from the Software Infrastructure Repository (SIR) [8]. *grep* is a GNU utility that searches the input files for lines containing a match to a given pattern list. When *grep* finds a match in a line of a text file, it copies the line to the standard output. But if the input file is a binary file, it normally[1] outputs either a one-line message saying that a binary file matches, or no message if there is no match. Furthermore, given the option "–quiet", *grep* should not write anything to the standard output.

In Line 1, `always_text` is 0 by default, `out_quiet` is 0 when the option "–quiet" is not specified, `memchr(bufbeg, '0',buflim-bufbeg)` returns a non-null

[1]By default, binary files are not treated as text files

pointer if the input is a binary file. In Line 3, `out_quiet` should be set to be positive if `not_text` is 1, so as to suppress the normal outputs of matchings. In Line 7, `nlines` records the number of found matches. Finally, in Line 14, the program checks whether the input is a binary file, the "–quiet" option is not specified, and there are some matches found. If so, the program should output a one-line message saying that a binary file matches. However, since `out_quiet` may be changed in Line 3, the checking in Line 13 may not work as expected. When the input is a binary file, the "–quiet" option is not specified, and there are some matches found, the checking in Line 14 is expected to return true. In this case, `out_quiet` is originally 0 and then is changed to be 1 in Line 3. Without restoring the original value of `out_quiet`, the checking in Line 14 returns false and no message would be output. Because returning no message is not the desired result, we call the test a failing test.

Without knowing the fault *a priori*, how can we identify such a test to be a suspicious one among a large number of tests? Our insight is that we may get some guidance from the (unverified) executions of the program. We run the *grep* programs on 809 tests that are also downloaded from SIR. Among the 809 tests, 667 tests cover the branch that `nlines` is true (Line 9), among which only 8 tests cover the branch that `memchr(bufbeg,'0',buflim-bufbeg)` is true (Line 1). Therefore, we can uncover a common operational model from the executions: "`nlines` >0 is ever true $\Rightarrow$ `memchr(bufbeg,'0',buflim-bufbeg)` is never true (in a test)". Although this model is not always true, it reflects the fact that we search more often in a text file than in a binary file, and a binary file is less likely to contain a match to a given pattern. The violations of this model are corner cases that may require special handling. Yet such corner cases are often neglected by programmers or their handling is too tedious to be fault free. Therefore, it is valuable to check the result of a test that violates the model, i.e., satisfying "`nlines` >0 is ever true $\wedge$ `memchr(bufbeg,'0',buflim-bufbeg)` is

ever true". We then find the selected test to be a failing test that reveals the fault.

We have implemented the proposed approach and conducted a comprehensive experimental study of the effectiveness and efficiency of the approach. The experimental results, compared with previous code-coverage-based, clustering-based, dynamic-invariant-based, and random selection approaches, show that our approach can more effectively reduce the number of tests for result inspection while revealing most of the faults.

The rest of this chapter is organized as follows. Section 4.2 reviews related work. Section 4.3 presents the proposed approach to mine common operational models. Section 4.4 presents the test selection approach. Section 4.5 describes the empirical studies and results. Section 4.6 concludes the work with future directions.

4.2 Related Work

In this section, we briefly review the existing work in test selection for result inspection, which can be classified into three main categories.

Coverage-based Test Selection. There exist a number of specification-coverage-based approaches for test selection. In partition testing [92], a test input domain is divided into subdomains based on some criteria, and then developers can select one or more representative inputs from each subdomain. When *a priori* specifications are provided for a program, Chang and Richardson [31] used specification coverage criteria to select a candidate set of test cases that exercise new aspects of the specification.

Moreover, various kinds of code coverage criteria have been proposed for test selection, such as control-flow testing criteria [59] and data-flow testing criteria [47]. Hutchins et al. [61] reported an experimental study investigating the effectiveness of control-flow and data-flow testing criteria. Their results

suggested that test sets achieving high code coverage levels usually showed significantly better fault-detection capability than randomly chosen test sets of the same size. However, the results also indicated that 100% code coverage alone is not a reliable indicator of the effectiveness of a test set. Leon et al. [76] evaluated the effectiveness of complex information flow criteria, which model indirect control/data dependencies between instructions or objects, for test selection. Their results suggested that test sets maximizing complex information flow criteria revealed more faults than test sets maximizing block coverage with substantial additional cost, and in some subjects the profiles could not be generated due to memory constraints.

Clustering-based Test Selection. Dickinson et al. [41] used clustering analysis to partition executions based on structural profiles, and employed sampling techniques to select executions from clusters for result inspection. They further proposed a failure-pursuit sampling approach [42] to enhance the efficiency in finding failures. Moreover, Leon et al. [76] evaluated the effectiveness of clustering analysis based on complex information flow criteria. Their results suggested that the effectiveness of the clustering analysis did not depend strongly on the type of used profiling.

Dynamic-invariant-based Test Selection. There are many approaches that mine dynamic invariants for test selection. A dynamic invariant is a property that holds at specific program points, e.g., a precondition `x < y` at entry to a procedure f. Ernst et al. [45] proposed to mine dynamic invariants from passing tests and develop a tool named Daikon. Daikon defines a set of templates for generating candidate dynamic invariants, including preconditions, postconditions, and loop invariants. The candidate invariants are evaluated on dynamic traces and a candidate is immediately discarded once it is violated. Any new test that violates the mined dynamic invariants may reveal faults and deserve manual inspection.

Harder et al. [54] proposed the operational difference approach to select tests based on Daikon. Their approach repeatedly adds new tests if they violate the invariants of the previously selected tests. Xie and Notkin [127] developed an operational violation approach called Jov for unit-test selection and generation. They mined operational models using Daikon from a set of manually written passing unit tests and selected automatically generated test inputs that violated the operational models. Pacheco and Ernst [100] developed a similar tool named Eclat, which further distinguishes illegal and fault-revealing inputs with some strategies. Hangal and Lam [52] developed DIDUCE that extracts operational models dynamically from long-running program executions. DIDUCE reports all detected violations at runtime and gradually relaxes invariants to allow for new behavior. Due to the limited number of existing passing tests or previously selected tests, the mined dynamic invariants of these approaches could be noisy and thus many model violations could be false positives.

4.3 Mining Common Operational Models

In this section, we propose two kinds of common operational models that are potentially fault-revealing, including control rules and data rules. A control rule is an implication relationship between branches. A data rule is an implicit constraint of the variable values. We mine common operational models from all the unverified tests. A program that is not of poor quality should pass most of the tests. Therefore, the common operational models mined from a set of unverified tests may be similar to the real models in passing tests.

4.3.1 Control Rules

Many faults can be revealed only when specific control paths are executed. Such control paths may have special branch combinations that the program-

mers do not expect. Therefore, we can mine common relationships between branches and isolate their violations as suspicious tests.

Given a branch condition C, let us denote its then-branch and else-branch as C=true and C=false. In a loop, a branch may be executed multiple times. We use two branch predicates C_t and C_f to denote that the branch C=true is ever covered (satisfied) and C=false is ever covered, respectively. Correspondingly, $\neg C_t$ means that C=true is never covered and $\neg C_f$ means that C=false is never covered. Note that $\neg C_t$ is not equivalent to C_f. It is possible that one of them is true and the other is false. The evaluations of a branch predicate x may be implied by other predicates. To model such implication relationships, we consider two kinds of rules $y \Rightarrow x$ and $y \Rightarrow \neg x$, where y is another branch predicate. We call $y \Rightarrow x$ and $y \Rightarrow \neg x$ control rules.

We have shown an example of the rule $y \Rightarrow \neg x$ in Figure 4.1. Here we show an example of the rule $y \Rightarrow x$ in Figure 4.2. The example program is a faulty version of the tcas program, which is an altitude separation controller, in the *Siemens* suite [61]. In Line 4, a $>$ operator is wrongly implemented as $\geq$. Let x and y be two branch predicates for denoting that the branch `upward_preferred=true` in Line 12 is ever covered and the branch `upward_preferred=true` in Line 5 is ever covered, respectively. Running 1608 tests on the program, we can find that y is true in 514 tests, among which x is true 478 times. Therefore, we can uncover a common operational model "$y \Rightarrow x$". This model reflects the real assumption of the programmers. Its violations cause errors that may finally become observable failures.

There may be a large number of control rules. We are interested only in the control rules that are likely to be true oracles and are violated by some tests. To evaluate the likelihood of a control rule to be a true oracle, we use the concept of confidence. The confidence of $y \Rightarrow x$ is defined as the ratio of the number of tests that satisfy $y \wedge x$ over the number of tests that satisfy y.

```
1   bool   Non_Crossing_Biased_Climb()
2   {
3       ...
4       upward_preferred = Inhibit_Biased_Climb()
                           >=Down_Separation;
                           /* >= should be > */
5       if(upward_preferred)
6           ...
7   }
8   bool   Non_Crossing_Biased_Descend()
9   {
10      ...
11      upward_preferred = Inhibit_Biased_Climb()
                           >Down_Separation;
12      if(upward_preferred)
13          ...
14  }
```

Figure 4.2: Faulty code of the tcas program

The confidence of $y \Rightarrow \neg x$ is defined as the ratio of the number of tests that satisfy $y \wedge \neg x$ over the number of tests that satisfy y. If a rule's confidence is 1, it can be omitted since there is no violation of this rule. Our approach then selects a subset of rules with high confidences. More specifically, for each predicate x, our approach selects the most confident rule $y \Rightarrow x$ and the most confident rule $y \Rightarrow \neg x$. Another possible way is to select the rules whose confidences are higher than a preset threshold, whose value may be application-dependent.

4.3.2 Data Rules

The values of a variable may have some implicit constraints. A failure may require or result in suspicious data values, i.e., violating the value constraints. Therefore, we can mine implicit constraints of variable values and isolate their violations as suspicious tests.

Given a variable V, we use V_{max} and V_{min} to denote the maximum value and minimum value ever assigned to V in a test. The values of a variable may be within an expected range. To model such potential constraints of variable values, we consider two kinds of rules $V_{max} \leq c_1$ and $V_{min} \geq c_2$, where c_1 and c_2 are constants (different variables have different c_1 and c_2). We call $V_{max} \leq c_1$ and $V_{min} \geq c_2$ data rules.

Unlike control rules, data rules have some parameters c_1 and c_2 to be determined. Let us first consider the rule $V_{max} \leq c_1$. Assume V_{max} follows the normal distribution. We can get the estimations of the mean value μ_1 and the standard deviation σ_1 based on the values of V_{max} in the observed but unverified tests. We would like to select a c_1 such that there is a high probability, say 0.9, that V_{max} is no more than c_1. Let $Z = (V_{max} - \mu_1)/\sigma_1$, then Z follows the standard normal distribution, i.e., having a mean of 0 and a standard deviation of 1. The problem $Prob(V_{max} \leq c_1) = 0.9$ is equivalent to the problem $Prob(Z \leq (c_1 - \mu_1)/\sigma_1) = 0.9$. Querying the cumulative probabilities of the standard normal table [19], we can get the solution $(c_1 - \mu_1)/\sigma_1 = 1.28$, i.e., $c_1 = 1.28 * \sigma_1 + \mu_1$. For example, if the set of values of V_{max} is {1,2,3,4,5}, we can estimate $\mu_1 = 3$ and $\sigma_1 = 1.58$. We then have $c_1 = 5.02$. There is no violation of the rule $V_{max} \leq c_1$. Alternatively, if the set of values of V_{max} is {1,2,3,4,10}, we can estimate $\mu_1 = 4$ and $\sigma_1 = 3.54$. We then have $c_1 = 8.53$. There is a violation of the rule $V_{max} \leq c_1$. Similarly, we would like to select a c_2 such that there is a high probability, say 0.9, that V_{min} is no less than c_2. We can get $c_2 = -1.28 * \sigma_2 + \mu_2$, where μ_2 and σ_2 are the mean value and the standard deviation of V_{min}.

Figure 4.3 shows an example of the data rule $V_{max} \leq c_1$. The example program is a faulty version of the print_tokens program, which is a lexical analyzer, in the *Siemens* suite [61]. In the loop, `token_ind` is increased by 1 each time. When `next_st` = 30, `token_ind` should be reset to 0, which is

```
1   while(!token_found)
2   {
3       if(token_ind<80)
4       {
5           token_str[token_ind++] = ch;
6           next_st = next_state(cu_state,ch);
7       }
8       ...
9       switch(next_st)
10      {
11          ...
12          case 30:
13              skip(tstream_ptr->ch_stream);
14              next_st = 0;
                /* missing code: token_ind = 0; */
15              break;
16      }
17  }
```

Figure 4.3: Faulty code of the print_tokens program

wrongly omitted. The omission of this assignment may make `token_ind` get unusually large values. Running the program on 4130 tests, the assignment of `token_ind` in Line 5 is executed in 4070 tests. Its maximum value has a mean value of 11 and a standard deviation of 13. So we can get a data rule $V_{max} \leq c_1 = 1.28 * \sigma_1 + \mu_1 = 28$, where V is `token_ind` in Line 5. Violations of this model may be caused by an omission fault and indicate failures.

To evaluate the likelihood of a data rule to be a true oracle, we also use the concept of confidence. The confidence of $V_{max} \leq c_1$ is defined as the ratio of the number of tests that satisfy $V_{max} \leq c_1$ over the number of tests where V is ever assigned. The confidence of $V_{min} \geq c_2$ is defined as the ratio of the number of tests that satisfy $V_{min} \geq c_2$ over the number of tests where V is ever assigned. If a rule's confidence is 1, it can be omitted since there is no violation of this rule.

4.4 Test Selection

Given a set of control rules and data rules, selecting all the tests that violate any of the rules may result in a large subset of the tests. Instead, our approach selects only a small subset of the tests that violate all these rules at least once, in a way that the most confident rules are violated by the selected tests first. Since the control rules and the data rules have different definitions of the confidence, we deal with them separately. The process of test selection based on the control rules is as follows. Initially, the set of selected tests is empty. Our approach sorts the control rules in descending order of confidence. From the top to bottom, if a rule is not violated by any of the previously selected tests, our approach randomly selects a test that violates the rule. Finally, in a greedy way each of the control rules is violated by the selected tests. Our approach also selects a subset of tests that violate all the data rules in the similar way. We merge together the selected tests based on the control rules and those based on the data rules as the final subset of selected tests.

4.5 Empirical Studies

In this section, we present a set of empirical studies to evaluate the effectiveness of our approach in test selection. In particular, we investigate three main research questions:

- RQ1: Can our approach select a small subset of tests that have high fault-detection capability? What is the effectiveness of violating different kinds of rules?
- RQ2: How does our approach compare with the existing approaches, including the code-coverage-based, clustering-based, and dynamic-invariant-based approaches?

Table 4.1: Characteristic of the subjects

Program	LOC	Test Cases	Faulty Versions	Failed Tests (Avg.)	Faulty Versions (Nontrivial)	Failed Tests (Nontrivial)
print_tokens	539	4130	7	69	7	69
print_tokens2	489	4115	10	224	4	109
replace	507	5542	31	106	29	93
schedule	397	2650	9	88	6	21
schedule2	299	2710	9	33	9	33
tcas	174	1608	41	39	36	28
tot_info	398	1052	23	83	11	24
Siemens suite	404	3115	130	92	102	54
Space	9564	13585	34	2111	17	164
grep	13358	809	20	177	9	12

- RQ3: What is the efficiency of our approach?

We next describe the subjects and measurements. We then present the results of our approach in test selection, compared with the existing approaches.

4.5.1 Subject programs

We have implemented the proposed approach and applied it to select tests in three subjects, including the *Siemens* suite [61], the *Space* program [106], and the *grep* program. All the three subjects are downloaded from the Subject Infrastructure Repository [8]. The first two subjects were also used in previous study of dynamic-invariant-based test selection [54].

The first subject is the *Siemens* suite [61], which was created by *Siemens* researchers. The *Siemens* researchers created 132 faulty versions of 7 programs that range in size from 170 to 540 lines. The *Siemens* researchers generated tests automatically from test-specification scripts, then augmented those tests with manually-constructed white-box tests such that each exercisable coverage unit was covered by at least 30 test cases. The numbers of tests range from 1052 to 5542. There are 130 faults that can be detected by the test suite in our environment.

The second subject is the *Space* program [106], which interprets Array Definition Language inputs. The *Space* program was developed at the European Space Agency. It has 9564 lines of C code. The test suite of *Space* contains 13585 test cases, where 10000 were randomly generated and the remainder were added to cover every statement or branch at least 30 times. There are 38 seeded faults, among which 34 faults can be detected by the test suite in our environment.

The third subject is the *grep* program, which is a GNU utility that searches the input files for lines containing a match to a given pattern list. It has 13358 lines of C code. There are 809 test cases, which were generated based on informal specifications and then augmented to increase statement coverage [8]. There are five original versions of the *grep* program, each of which was seeded by a number of faults. In our environment, there are in total 20 faults that can be detected by the test suite.

The characteristic of the subject programs is shown in Table 4.1. In these subjects, there are many trivial faults that fail on more than 5% of the tests. Such faults are less likely to be seen in practical setting. To better evaluate the potential effectiveness of test selection approaches in practice, we conduct additional experiments on only the nontrivial faults.

4.5.2 Measurement

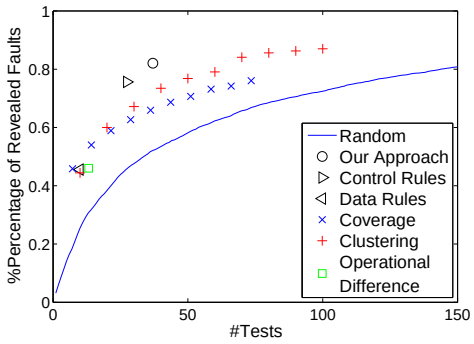
To evaluate the effectiveness of an approach in test selection, we use two metrics. The first metric is the size of the selected test suite, i.e., the number of the selected tests. This metric indicates the amount of human effort required to check the results of the selected tests. The second metric is the percentage of faults that can be revealed by the selected tests. A set of tests are said to reveal a fault if the faulty program fails on one or more of the tests. We would like to reveal as many faults as possible. A good test-selection algorithm should select a small number of tests that can reveal most of the faults.

4.5.3 Results

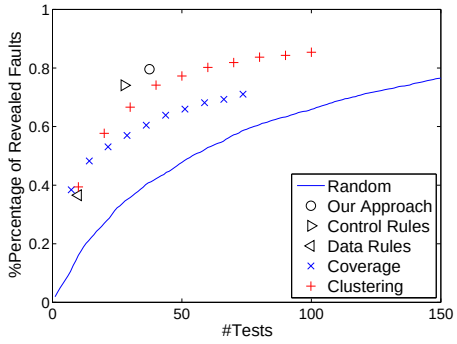
Figure 4.4 shows the results of our approach and the existing test selection approaches, including the code-coverage-based, clustering-based, and dynamic-invariant-based approaches. The x-axis is the number of selected tests and the y-axis is the percentage of faults revealed by the selected tests. We also present the results of random selection as the comparison baseline. Because the three subjects are quite different in the program size and the original test suite size, we present the results in these three subjects separately. Tables 4.2, 4.3, 4.4, and 4.5 show some of the detailed results, including the number of selected tests and the percentage of revealed faults in each subject program. To reduce the potential effect of random noise, we run all of the experiments 50 times and then report the averaged results.

Effectiveness of Our Approach

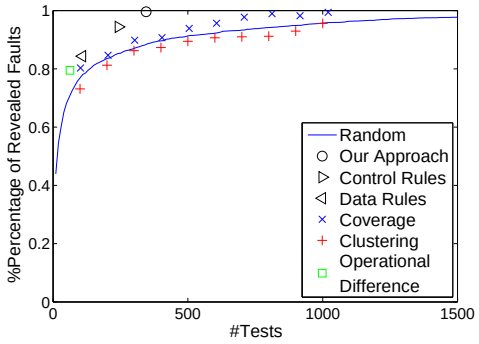
We observe that our approach is effective in reducing the number of tests while revealing most of the faults. In the *Siemens* suite (all faults), our approach selects only 37 tests for the programs on average, which can still reveal 82%



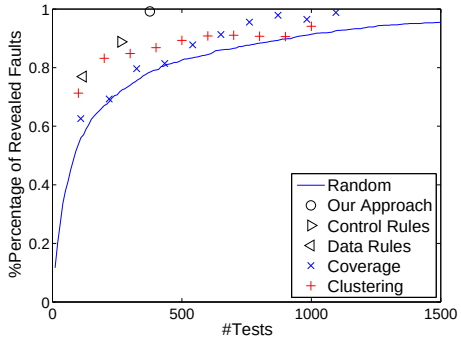
All faults, the *Siemens* suite



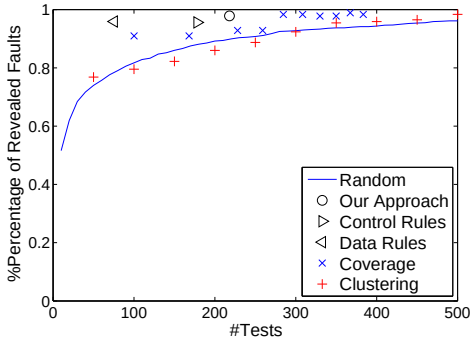
Nontrivial faults, the *Siemens* suite



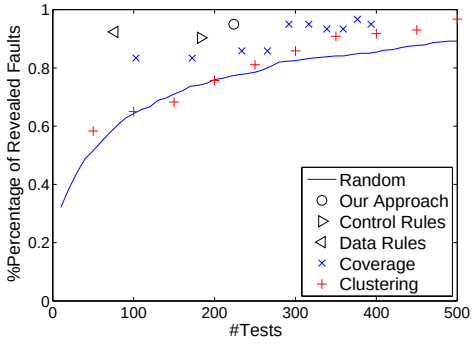
All faults, the *Space* program



Nontrivial faults, the *Space* program



All faults, the *grep* program



Nontrivial faults, the *grep* program

Figure 4.4: Results of test selection

Table 4.2: Results of our approach on all the faults (#T: number of tests, %F: percentage of revealed faults)

Program	Original Test Suite		Our Approach		Control Rules		Data Rules	
	#T	%F	#T	%F	#T	%F	#T	%F
print_tokens	4130	100	25	89	17	88	8	50
print_tokens2	4115	100	41	100	30	100	10	61
replace	5542	100	75	80	60	73	16	37
schedule	2650	100	31	86	24	70	7	49
schedule2	2710	100	32	62	24	61	9	25
tcas	1608	100	26	74	15	68	12	23
tot_info	1052	100	29	84	21	71	9	74
Siemens suite	3115	100	**37**	**82**	27	76	10	46
Space	13585	100	**345**	**100**	242	94	109	84
grep	809	100	**218**	**98**	178	96	75	96

Table 4.3: Results of other approaches on all the faults (#T: number of tests, %F: percentage of revealed faults)

Program	Original Test Suite		Random Selection		Coverage (k=1)		Clustering		Operational Difference	
	#T	%F	#T	%F	#T	%F	#T	%F	#T	%F
print_tokens	4130	100	37	39	6	61	40	84	9	37
print_tokens2	4115	100	37	78	4	90	40	100	6	51
replace	5542	100	37	45	12	33	40	57	18	45
schedule	2650	100	37	48	7	26	40	60	10	33
schedule2	2710	100	37	34	5	26	40	47	13	30
tcas	1608	100	37	46	11	31	40	84	26	55
tot_info	1052	100	37	75	5	53	40	82	9	72
Siemens suite	3115	100	**37**	**52**	**7**	**46**	**40**	**73**	**13**	**46**
Space	13585	100	**345**	**89**	**102**	**80**	**400**	**87**	**63**	**80**
grep	809	100	**219**	**90**	**100**	**91**	**250**	**89**	-	-

Table 4.4: Results of our approach on nontrivial faults (#T: number of tests, %F: percentage of revealed faults)

Program	Original Test Suite		Our Approach		Control Rules		Data Rules	
	#T	%F	#T	%F	#T	%F	#T	%F
print_tokens	4130	100	25	89	17	88	8	50
print_tokens2	4115	100	42	100	32	100	10	38
replace	5542	100	75	78	60	71	16	35
schedule	2650	100	32	87	25	84	7	35
schedule2	2710	100	32	62	24	61	9	25
tcas	1608	100	26	72	15	65	12	18
tot_info	1052	100	30	69	21	50	9	55
Siemens suite	3115	100	**37**	**80**	28	74	10	37
Space	13585	100	**376**	**99**	265	89	118	77
grep	809	100	**224**	**95**	183	90	76	92

Table 4.5: Results of other approaches on nontrivial faults (#T: number of tests, %F: percentage of revealed faults)

Program	Original Test Suite		Random Selection		Coverage (k=1)		Clustering	
	#T	%F	#T	%F	#T	%F	#T	%F
print_tokens	4130	100	37	39	6	61	40	84
print_tokens2	4115	100	37	54	5	76	40	100
replace	5542	100	37	42	12	32	40	60
schedule	2650	100	37	26	7	26	40	80
schedule2	2710	100	37	34	5	26	40	47
tcas	1608	100	37	40	11	25	40	82
tot_info	1052	100	37	51	5	23	40	67
Siemens suite	3115	100	**37**	**41**	**7**	**38**	**40**	**74**
Space	13585	100	**376**	**79**	**110**	**63**	**400**	**87**
grep	809	100	**225**	**77**	**103**	**83**	**250**	**81**

of the faults. In the *Space* and the *grep* programs (all faults), our approach selects 345 and 219 tests on average, which can reveal 100% and 97% of the faults, respectively. We note that randomly selecting the same number of tests as our approach can also reveal 52%, 89% and 90% of the faults in these subjects. This result is due to two main factors: (1) many trivial faults can be easily revealed and (2) the probability of finding no failures of a fault decreases exponentially with the number of selected tests. Checking the results in nontrivial faults, we can see that our approach is still effective while the effectiveness of random selection decreases much. For example, in the *Siemens* suite (nontrivial faults), our approach selects only 37 tests for the programs on average, which can still reveal 80% of the faults. Randomly selecting the same number of tests can reveal only 41% of the faults.

We also evaluate the effects of the two kinds of common operational models that we proposed, i.e., control rules and data rules, separately. The results are plotted in Figure 4.4, and shown in Tables 4.2 and 4.4. We observe that both these two kinds of rules are helpful in revealing faults. Among them, violating all the control rules can reveal more faults than violating all the data rules, but also requires much more tests. The data rules are better for some programs such as tot_info, which is an information measure program that deals with data tables. In summary, these two kinds of rules are complementary to each other as they reflect different aspects of program behaviors. When combined together, they are able to help select quite a good subset of tests.

Comparison with the Code-coverage-based Approach

We compare our approach with the code-coverage-based approach. The code-coverage-based approach attempts to cover as many program elements of a given type as the original test suite with as few test cases as possible [76]. Selecting a minimal-size, coverage-maximizing subset of a test suite is an

instance of the set-cover problem, which is often solved using a greedy approximation algorithm. On each of its iterations, the greedy algorithm selects the test that covers the largest number of elements not covered by the previously selected tests. This approach is based on the assumption that many software faults and their caused failures can be revealed simply by exercising such elements, regardless of other factors. To evaluate the potential capability of finding more faults using the code-coverage-based approach, we also extend the basic code-coverage-based approach by increasing the number of times each program element should be covered. We say a program element is covered k times if there are k different tests that cover it. We use the branch coverage and we experiment with k from 1 to 10. The results are plotted in Figure 4.4, and shown in Tables 4.3 and 4.5.

We observe that the basic code-coverage-based approach, in which each branch is covered at least once, is good in selecting a small subset of tests that can reveal many faults. However, it misses many other faults, e.g., it misses more than half of the faults in the *Siemens* suite. Our approach can select a test suite with a much higher fault-revealing capability, and the number of selected tests is only a few times larger than that of the basic code-coverage-based approach. Increasing the number of the times that each branch should be covered helps reveal more faults, but is less cost-effective than our approach. Sometimes increasing k may decrease the percentage of revealed faults. Such an observation is due to that the smallest subset of tests that covers each program element at least k times may not be a subset of the smallest subset of tests that covers each program element at least k+1 times.

Comparison with the Clustering-based Approach

We next compare the results of our approach with the results of the clustering-based approach [41]. The clustering-based approach uses agglomerative hi-

erarchical clustering to cluster the tests, and then selects one test from each cluster. The main assumption of the clustering-based approach is that a significant number of failures are isolated in clusters of small size. Besides one-per-cluster sampling, there are other possible sampling schemes that aim at finding more failing tests for each fault [42]. As our concern is to increase the likelihood of finding at least one failing test for each fault, we do not compare our approach with these other sampling schemes. We use the binary branch profiles for clustering. We do not use the variable-value profiles since a variable may not be observed in all the tests. We experiment with 10 different values of the number of the clusters. The values are 10 to 100, 100 to 1000, and 50 to 500 in the *Siemens* suite, the *Space* program, and the *grep* program, respectively. We select these values based on the program sizes and the numbers of tests selected by the code-coverage-based approach. The results are plotted in Figure 4.4, and shown in Tables 4.3 and 4.5.

We observe that the clustering-based approach performs better than random selection in the *Siemens* suite (all faults), the *Siemens* suite (nontrivial faults), and the *Space* program (nontrivial faults), but not better than random selection in the other experiments. There are two main reasons for such results. First, there is a large number of tests in the *Siemens* and the *Space* program. The tests could be highly redundant, especially for the common execution paths. The clustering-based approach may then help to cover more different execution paths. However, there are not a large number of tests in the *grep* program, and these tests are not redundant. In this case, the clustering-based approach may not help cover more different execution paths or isolate the most suspicious tests. Second, there are many trivial faults in the *Space* program and the *grep* program, which may violate the assumption of the clustering-based approach that a significant number of failures are isolated in clusters of small size. Our approach can isolate tests that are

suspicious in some program elements. It is more stable in different cases and is better than the clustering-based approach in general.

Comparison with the Dynamic-Invariant-based Approach

Finally, we compare our approach with the dynamic-invariant-based approach. Harder et al. [54] proposed the operational difference approach to select tests based on Daikon. It starts with an empty test suite and repeatedly adds new tests if they violate the invariants of the previously selected tests. To control the number of tests, the algorithm terminates when n (n=50 in their experiments) consecutive tests are considered and rejected. They also conducted experiments on the *Siemens* suite and the *Space* program that we used[2]. We adopt the experimental results from their original paper directly, which are shown in Table 4.3 and plotted in Figure 4.4. No result of the operational difference approach in the *grep* program or on the nontrivial faults is available (Daikon cannot be scalable to deal with the *grep* program and therefore poses barriers for us to re-implement and apply the operational difference approach on the *grep* program).

We observe that the operational difference approach performs similarly to the basic code-coverage-based approach. It works well in selecting a small subset of tests that can reveal many faults. However, it misses many other faults at the same time. The operational difference approach may be able to reveal more faults if the termination condition is removed. However, it may then select much more tests due to the false positives in the detected invariants. By using the information of all the unverified tests at hand, our approach can reduce the noise of mined operational models and thus select a reasonably sized subset of tests that can reveal most of the faults.

[2]In their experiments, they only have 30 faulty versions for the replace program (31 in our experiments). But this difference can be negligible.

Table 4.6: Efficiency of Our Approach (*seconds*)

Program	LOC	#Tests	Our Approach	Coverage	Clustering
print_tokens	539	4130	1.4	0.4	366.2
print_tokens2	489	4115	5.4	0.7	82.5
replace	507	5542	7.8	1.0	390
schedule	397	2650	1.0	0.2	65.1
schedule2	299	2710	1.3	0.2	93.8
tcas	174	1608	0.7	0.1	38.8
tot_info	398	1052	0.8	0.1	14.7
Siemens suite	400	3115	2.6	0.4	150.2
Space	9564	13585	598.8	5.4	1364
grep	13358	809	92.9	4.9	6.8

Efficiency

To evaluate the efficiency of our approach, we measure the time cost of our approach on the three subjects, compared with that of the basic code-coverage-based approach and that of the clustering-based approach. The result is shown in Table 4.6.

Among the three approaches, the basic code-coverage-based approach is the fastest. It takes at most a few seconds in each of the subject programs. The clustering-based approach is fast in the *grep* program but is not fast in other programs. Basically, the more tests there are, the longer time the clustering-based approach takes. However, there is no clear polynomial relationships between the time cost and the number of tests. The time cost of the clustering approach can be greatly affected by the distribution of the tests.

Our approach is fast in the *Siemens* suite and takes reasonable time in the *Space* program and the *grep* program. More specifically, our approach takes about 10 minutes in the *Space* program, which contains about 9564 lines of code and 13,585 tests. However, our approach may not be efficient enough for large programs with large test suites. The main cost of our approach is the time of mining the control rules, which is proportional to the number of tests and to the square of the number of branches. For a large program with 100,000 lines of code and a large test suite that has 13,585 tests, our approach may take about 20 hours. In this case, a possible improvement is to mine only the control rules between the branches in the same modules. We can then greatly reduce the number of candidate control rules and thus the time cost of our approach.

Summary and Discussions

Our results suggest the following observations:

- Our approach is effective in reducing the number of tests while revealing most of the faults. Control rules and data rules are complementary to each other as they reflect different aspects of program behaviors.
- Our approach can select a test suite with a much higher fault-revealing capability than those of the code-coverage-based approach and the dynamic-invariant-based approach, and is more robust and cost-effective than the clustering-based approach. But we note that there are some issues to be considered. The code-coverage-based approach may reveal more faults if more complex coverage profiles are used, which however are more difficult to collect. The clustering-based approach is more flexible for selecting different numbers of tests.
- Our approach takes reasonable time for small to medium sized programs

with large test suites. However, it may not be efficient enough for large programs with large test suites. This issue may be alleviated by mining only the control rules between the branches in the same modules. We need to conduct further experiments to investigate this technique.

4.5.4 Threats to Validity

The threats to external validity primarily include the degree to which the subject programs, faults, and test cases are representative of true practice. The *Siemens* programs are small and the *Space* program is of medium size. Most of the faulty versions involve simple, one or two-line manually seeded faults. Moreover, the tests are manually or randomly generated, while in automated testing, a number of test generation approaches generate tests systematically. These threats could be reduced by more experiments on wider types of subjects with systematically generated tests in future work. The threats to internal validity include the fact that different approaches select different numbers of tests. This fact may make the comparison of fault-revealing effectiveness bias to the approaches that select larger number of tests. To reduce this bias, we present the results of random selection as the baseline. We also evaluate the fault-revealing capabilities of other approaches with different numbers of selected tests.

4.6 Summary

We have proposed an approach for test selection without *a priori* specifications. We propose to mine common operational models, which are often but not always true in all observed traces, from a set of unverified tests. Specifically, we collect branch coverage and data value bounds at runtime and then mine implication relationships between branches and constraints of data values as common operational models after running all the tests. We then

select tests that violate all these common operational models greedily. We have evaluated our approach on the *Siemens* suite, the *Space* program, and the *grep* program, compared with code-coverage-based, clustering-based, and dynamic-invariant-based approaches. The experimental results show that our approach can select a test suite with a much higher fault-revealing capability than those of the code-coverage-based approach and the dynamic-invariant-based approach, and is more robust and cost-effective than the clustering-based approach.

We plan to pursue several future directions for improving our approach. First, we plan to combine our approach with automatic test generation tools. Our current experiments are based on existing test suites whose tests are manually or randomly generated. It is valuable to investigate how our approach can work on automatically generated test sets. Second, we plan to conduct experiments on large programs with large test suites. Our current implementation may not be efficient enough for such cases. We would like to evaluate some possible improvements, such as mining only the control rules between the branches in the same modules. Third, we plan to propose common operational models for specific applications based on domain knowledge.

□ **End of chapter.**

Chapter 5

Mining Test Oracles of Web Search Engines

Web search engines have major impact in people's everyday life. It is of great importance to test the retrieval effectiveness of search engines. However, it is labor-intensive to judge the relevance of search results for a large number of queries, and these relevance judgments may not be reusable since the Web data change all the time. In this chapter, we propose to mine test oracles of Web search engines from existing search results. The main idea is to mine implicit relationships between queries and search results, e.g., some queries may have fixed top 1 result while some may not, and some Web domains may appear together in top 10 results. We define a set of properties of queries and search results, and mine frequent association rules between these properties as test oracles. Experiments on major search engines show that our approach mines many high confidence rules that help to understand search engines and detect suspicious search results.

5.1 Problem and Motivation

Web search engines are becoming more and more important for people to search for information in the World Wide Web. Given a query, a good search engine should return desired search results that possess various properties such as relevance, authority, and freshness. Providing inadequate search results could mislead or dissatisfy users. As an example, Figure 5.1 shows the clarification message put in the official PuTTY (a free telnet/ssh client) Web page due to the unexpected change of Google's search results.

However, it is difficult to test search engines due to the lack of test oracles. In particular, since the Web data and the information need of users keep changing, the desired search results may change along the time, even when the search engines do not change. Existing approaches on search engine testing/evaluation rely on relevance judgments of search results, collected either explicitly [69] or implicitly [67, 120]. It is labor-intensive to manually label a large number of relevance judgments of search results, i.e., test oracles for the queries, and these relevance judgments may not be reusable due to the dynamic nature of the Web. On the other hand, clickthrough data can be used as implicit judgments of search results [67, 120]. But clickthrough data suffer from various biases such as the position bias and summary bias [130]. In particular, if a desired result is not found by a search engine, there is no clickthrough data for it.

In this work, we propose to mine test oracles of Web search engines from existing search results. Previous work on specification mining [23, 45, 56] suggests that one can mine likely invariants or frequent patterns as test oracles from the execution of existing tests. Violations of these mined test oracles are suspicious and may reveal potential bugs of the systems under test. Using such approaches, we may mine test oracles of search engines to help testers understand and examine search engines' behaviors, or detect suspicious search

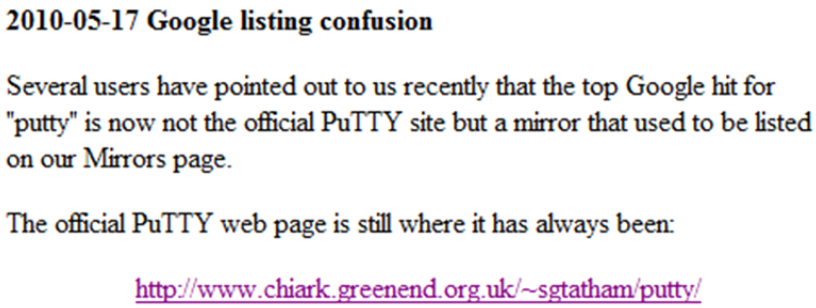


Figure 5.1: Declaration from the official PuTTY Website for Google's search result change

results automatically. We can then increase the cost-effectiveness of finding defects of search engines significantly.

However, there are two challenges of applying existing specification mining approaches on Web search engines. First, many interesting patterns of search engines may need to be mined from search results of multiple days or multiple search engines. We need to integrate all these search results for mining, regardless of the changes in a search engine's implementation or the differences in different search engines' implementations. Existing specification mining approaches often mine patterns with regard to the implementation and are thus not suitable for this task. To address this problem, we define a set of properties of queries, search results, matches between queries and search results, and search engine identities. These properties reveal many aspects of the search results and are not affected by differences of the implementations. We can then map search results of different search engines in different time to itemsets of these properties for mining.

Second, it could be difficult to mine all interesting specifications from a large set of search results effectively. The invariant detection approach [45] can mine invariants from passing tests quickly by discarding any violated can-

didates. But it is difficult to get a large set of passing tests for search engines. On the other hand, the frequent pattern mining approaches [23] often assume the existence of a suitable threshold of the frequency of patterns. However, different kinds of specifications of search engines may have great differences in the frequency. For example, a query's best top 1 search result (recently) may have only a few supporting itemsets, while implications between two Web domains may have a large number of supporting itemsets. Given a large set of search results, there may be not a good frequency threshold that can finish in a reasonable time without missing many important specifications. To ease the mining of different kinds of specifications, we apply the association rule mining technique, and design a set of controlling schemes for rule generation. The schemes include stop words, left-hand-side (LHS) patterns, and right-hand-side (RHS) patterns. These schemes can help the testers offer more guidance to generate the desired kinds of rules effectively.

To evaluate our apporach, we have collected the search results of Google and Bing for 4232 queries in 4 months. We choose Google and Bing to test as they are the most popular search engines nowadays. These two search engines, together with many other search engines powered by them (e.g., Yahoo! Search is now powered by Bing and AOL Search is powered by Google), possess more than 90 percent search market share in U.S. [1]. The queries consist of 800 common queries used in the KDD-Cup 2005 competition task [78] and 3432 hot queries of Google and Yahoo[1]. We collected the search results of the queries from December 25, 2010 to April 21, 2011. Our approach mines many high confidence rules that help to understand search engines. Our approach also detects suspicious search results, which are much less than the queries that change search results, for manual investigation.

The rest of this chapter is organized as follows. Section 5.2 reviews related

[1]Bing used to have a service of hot queries named Bing xRank, which however has been shut down.

work. Section 5.3 provides the background of the association rule mining technique that our approach is based on. Section 5.4 presents the proposed approach for mining test oracles of search engines. Section 5.5 describes the data collection and the evaluation of the proposed approach. Section 5.6 concludes the chapter.

5.2 Related Work

5.2.1 Search Engine Evaluation

There are a number of approaches evaluating the retrieval effectiveness of information retrieval systems including Web search engines. The basic procedure includes constructing a set queries (and a set of documents), running information retrieval systems on the queries, collecting relevance judgments for the search results, and calculating well-defined measurements such as Precision, MAP, and NDCG [64, 29] to estimate the retrieval effectiveness of different information retrieval systems.

A main issue in evaluating the retrieval effectiveness is how to reduce the manual efforts of collecting relevance judgments. An approach is to use the *pooling* process [69], which is widely employed in TREC evaluation [16]. In TREC, different information retrieval systems submit the top K results per topic to NIST. NIST then forms pools of unique documents from all submissions, in which the assessors judge for relevance. Systems are then evaluated using the relevance judgments. For Web search engines, clickthrough data can be used as implicit feedback to evaluate the retrieval effectiveness of search engines [66, 67, 120]. But clickthrough data suffer from various biases such as the position bias and summary bias [130]. Therefore, manual investigation are still needed for a large amount of search results.

5.2.2 Mining Specifications

Our approach is also related to the work of mining specifications dynamically, which mines specifications from the execution of existing tests. These work mainly mines three kinds of specifications: temporal models [23], algebraic models [56], and operational models [45].

A temporal model is a requirement on the ordering of specific actions or events that are often function calls, e.g., `fopen` should be followed by `fclose`. Ammons et al. [23] proposed an approach to summarize the frequent interaction patterns in program execution as probabilistic finite state automata (PFSA). Lo and Khoo [84] improved the quality of mining results by filtering erroneous execution traces and clustering related traces.

An algebraic model is an axiom that describes the observable behavior of a class without revealing implementation details, e.g., for a stack s and an object o, `pop(push(s,o).state).state=s`. Henkel and Diwan [56] presented an automatic tool for extracting algebraic specifications from Java classes. Their tool maps a Java class to an algebraic signature and then uses the signature to generate a large number of terms. The tool evaluates these terms, proposes equations, and generalizes equations to axioms. Xie and Notkin [128] proposed an approach to mine statistical algebraic abstractions, each of which is associated with the counts of its satisfying and violating instances during test executions.

An operational model is a property that holds at specific program points, e.g., a precondition `x < y` at entry to a procedure f. Ernst et al. [45] proposed to mine dynamic invariants from passing tests and develop a tool named Daikon. Daikon defines a set of templates for generating candidate dynamic invariants, including preconditions, postconditions, and loop invariants. The candidate invariants are evaluated on dynamic traces and a candidate is immediately discarded once it is violated.

All of these approaches focus on mining properties of a specific implementation in some specific time. Instead, our approach designs a set of properties for the inputs and outputs of search engines, so as to integrate the search results of different search engines in different time for mining. Moreover, our approach provides a set of controlling schemes for rule generation, including stop words, left-hand-side (LHS) patterns, and right-hand-side (RHS) patterns. These schemes can help the testers offer more guidance to generate the desired kinds of rules effectively.

5.2.3 Test Selection for Result Inspection

Using the mined test oracles, our approach can reduce the efforts of manually investigating search results, which is essentially a test selection for result inspection task. Many approaches [54, 100, 127] select tests that violate mined specifications, as described above. Moreover, various kinds of code coverage criteria have been proposed for test selection, such as control-flow testing criteria [59] and data-flow testing criteria [47]. Hutchins et al. [61] reported an experimental study investigating the effectiveness of control-flow and data-flow testing criteria. Their results suggested that 100% code coverage alone is not a reliable indicator of the effectiveness of a test set. Dickinson et al. [41] used clustering analysis to partition executions based on structural profiles, and employed sampling techniques to select executions from clusters for result inspection. They further proposed a failure-pursuit sampling approach [42] to enhance the efficiency in finding failures. These approaches select tests based on the information related to the implementation, while our approach mines frequent patterns in the system level and thus can easily integrate the tests of different systems in different time.

5.3 Background of Association Rule Mining

Our approach is based on a data mining technique called *association rule mining* [51], which originates from data mining problems of market basket transactions [22]. For the ease of discussion, we briefly review some basic terminologies of association rules.

Let $I = \{i_1, i_2, \ldots, i_m\}$ denote a set of *items*. Let D be a database of transactions, where each transaction T is a set of items such that $T \subseteq I$. A collection of zero or more items is called an *itemset*. A transaction T contains an *itemset* X if $X \subseteq T$. The *support* of an *itemset* X, denoted as $\sup(X)$, is the number of transactions in D that contain X. We call an itemset X a *frequent itemset* if its support is large, i.e., $\sup(X) >$ *minsup*, where *minsup* is a threshold of support. For example, consider a datebase $\{\{a, b, c\}, \{a, d, e\}, \{a, b\}\}$. The support of the itemset {a} is 3 and the support of the itemset {a,b,c} is 1. If $minsup = 2$, the frequent itemsets are {a}, {b}, and {a,b}.

An *association rule* is an implication expression between two *itemset*. Formally, an *association rule* is defined as follows: An *association rule* is an implication expression of the form $X \Rightarrow Y$, where $X \subseteq I$ and $Y \subseteq I$ are two disjoint *itemsets*, i.e., $X \cap Y = \emptyset$.

We can measure the importance of an association rule $X \Rightarrow Y$ using the *support* and the *confidence*. The *support* of $X \Rightarrow Y$ is equal to the support of the union set $X \cup Y$. The *confidence* of the rule $X \Rightarrow Y$, denoted as $\text{conf}(X \Rightarrow Y)$, is the percentage of transactions containing X that also contain Y. For example, in the example database described above, the support and confidence of the rule $a \Rightarrow b$ is 2 and 2/3, respectively.

For a database of itemsets, the problem of mining association rules is to find all association rules having $support \geq minsup$ and $confidence \geq minconf$, where *minsup* and *minconf* are the corresponding support and con-

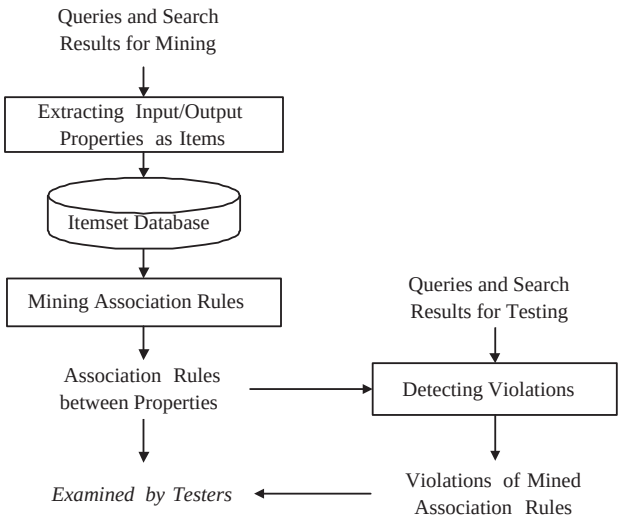


Figure 5.2: Overview

fidence thresholds. Generally, the association rule mining problem consists of two main subtasks: mining frequent itemsets and generating rules from the frequent itemsets.

5.4 Our Approach

5.4.1 Overview

Figure 5.2 presents an overview of our approach. The main idea of our approach is to extract rules between properties of queries and search results automatically. We use items to represent certain properties. An item can be a word in the query, the domain of a top 10 search result, whether the URLs contain the words of the query, and so on. In general, we consider items of four broad categories: (i) items based on the query, (ii) items based on the results, (iii) items based on the matching between the query and the results, and (iv) search engine identities. Section 5.4.2 describes the design of these items in detail. Given a set of search results (and their queries), our approach

will extract the items and build a database of itemsets for the search results. The set of search results for mining could be the current or previous search results of a search engine or multiple search engines.

There may be implicit rules between different items. For example, some domain may always be ranked high for some query in a series of time. In order to mine these rules efficiently, our approach employs the association rule mining technique on the database of itemsets. Our approach first mines frequent itemsets from the database of itemsets. It then mines association rules from the mined frequent itemsets. In particular, our approach designs a set of schemes for controlling the generation of rules. Testers may examine the mined rules to learn and examine search engines' behaviors. If there is any rule that seems unreasonable, there may be design pitfalls in some modules of the search engine. Section 5.4.3 describes the mining approach in detail.

After mining association rules, our approach automatically detects violations of these rules in any given search results. It groups and ranks violations based on the violated rules. Testers can examine the top violations together with the violated rules. Section 5.4.4 describes the procedure of detecting violations in detail.

5.4.2 Extracting Items from Queries and Search Results

Our approach extracts items based on the query, the results, the matching between the query and the results, and search engine identities. We next describe the items of different categories in detail. Table 5.1 provides an summary of the items.

Query Items

A natural kind of items in the query phrase are the query words. For example, a query "ase 2011" contains two words "ase" and "2011". The classes of

Table 5.1: Summary of the Items

Category	Item Description	(Example) Items
Query	the query.	Q:ase 2011
Query	A word in the query.	QW:ase
Query	A common query.	CommonQ
Query	A hot query.	HotQ
Query	The number of words in the query.	OneWord
Query	The number of words in the query.	TwoWords
Query	The number of words in the query.	ManyWords
Search Result	The domain of the top 1 search result.	top1:continuinged.ku.edu
Search Result	The domain of a top 10 search result.	top10:continuinged.ku.edu
Search Result	The Alexa Ranks of the top 10 results' top private domains are all greater than 1,000.	ALLGE1K
Search Result	The Alexa Rank of some top 10 result' domain is greater than 100,000.	SOMEGE100K
Search Result	None of the Alexa Rank of the top 10 results' domains is greater than 100,000.	NOGE100K
Match	The whole query phrase does not appear in the title of any top 10 search result.	NoTitleHasQ
Match	The whole query phrase does not appear in the summary of any top 10 search result.	NoSummaryHasQ
Match	The whole query phrase appears in the title of all top 10 search result.	ALLTitleHasQ
Match	The whole query phrase appears in the summary of all top 10 search result.	ALLSummaryHasQ
Search Engine	The search engine that returns the search results.	SE:google

queries are also important items. For example, search results of hot queries may have different characteristics from those of common queries. Other items can also be derived from query properties, such as the length of the query, etc. Note that some items are annotated with their types, e.g., "Q:ase 2011" means a query "ase 2011". This choice is to make the meaning of items more clear and make it easier to specify constraints on a set of items.

Search Result Items

The output of a query is actually a ranked list of search results, where each result contains a title, a URL, and a summary, etc. Our approach focuses on the URL of the top 10 search results. A URL is often too specific to have implications with other items. Therefore, our approach uses the domains of the URLs as items. For example, the URL "http://www.continuinged.ku.edu/programs/ase/" comes from the domain "continuinged.ku.edu". Our approach also examines the Alexa Rank of the top private domains of search results. A top private domain is a public suffix, under which Internet users can directly register names, plus its first child. For example, the top private domain of the URL "http://www.continuinged.ku.edu/programs/ase/" is "ku.edu". Our approach uses Guava (Google Core Libraries for Java 1.5+) [4] to extract the top private domain of a given URL, based on the Mozilla Foundation's Public Suffix List [17]. Our approach then extract items such as ALLGE1K (the Alexa Ranks of the top 10 results' domains are all greater than 1,000) and SOMEGE1M (the Alexa Rank of some top 10 result' domain is greater than 1,000,000).

Match Items

Our approach also extracts items about how well the search results match the query. An example item is that whether the whole query phrase appears

in the title of any search result. Similar items include whether the query phrase appears in the title of all search results, and whether the query phrase appears in the summary of all search results. One can expect that the titles and the summaries of top search results often contain matches of the query, while few or no matches likely correspond to a less relevant result.

Search Engine Identities

When there are search results of multiple search engines, there may be rules specific for certain search engines. Therefore, our approach also extract items to describe which search engine returns the search results. For example, "SE:google" means that the search results are returned by Google.

5.4.3 Mining Association Rules

Based on the definition of items, our approach maps each output (a ranked list of search results) of a query into a transaction, i.e., a set of items. A set of queries and their search results are then mapped to a database of itemsets (actually, our approach further maps the items to integer values to mine rules effectively). The task is then to mine frequent association rules from the database. Our approach first mines frequent itemsets from the database, and then generates rules with high confidences from the frequent itemsets. Our approach also designs a set of controlling schemes to guide the rule generation. We next describe these three techniques separately.

Frequent Itemset Mining with Length Constraint

In practice, we may not be interested in rules containing too many items, because they are too complex to understand and the implications are more likely to be coincidences. Therefore, our approach uses another threshold for the length of rules (i.e., the number of items in rules), denoted as $maxL$. Given

a length threshold *maxL*, our approach adopts the Apriori algorithm [22] to generate frequent itemsets up to length *maxL* iteratively. Apriori employs an iterative approach known as a level-wise search, where k-itemsets are used to explore $(k+1)$-itemsets. First, the set of frequent 1-itemsets, denoted $L1$, is found by scanning the database to accumulate the count for each item, and collecting those items that satisfy minimum support. $L1$ is then used to find $L2$, the set of frequent 2-itemsets, which is then used to find $L3$, and so on. The Apriori property "all nonempty subsets of a frequent itemsets must also be frequent" is used to generate and prune candidates of $L(k+1)$ based on Lk, which greatly improves the efficiency.

Generating Rules

Given a frequent itemset $x_1, x_2, ..., x_n$, we can generate $2^n - 2$ association rules from it. However, these rules could be highly redundant. In particular, many rules whose righthand side has more than two items may not provide more information than rules whose righthand side has just one item, especially when used for detecting violations. For example, consider the rule $x_1, x_2, ..., x_{n-2} \Rightarrow x_{n-1}, x_n$. Any violation of this rule must also violate the rules $x_1, x_2, ..., x_{n-2} \Rightarrow x_{n-1}$ and $x_1, x_2, ..., x_{n-2} \Rightarrow x_n$, which have higher or equal support and confidence. Therefore, our approach generates only rules whose righthand side has just one item from the frequent itemsets. The algorithm is described as follows.

```
for each mined frequent itemset X,
    for each item x_i in X
        generate a rule X − x_i ⇒  x_i
        if the rule's confidence is greater than minconf,
            store the rule to the rule list,
rank the rules in descending order of confidence and support.
```

Controlling Rule Generation

While our approach can mine association rules between various kinds of items, one may want to offer more guidance to the procedure of rule generation. Our approach provides several schemes for this purpose, including stop words, left-hand-side (LHS) patterns, and right-hand-side (RHS) patterns.

Stop words are items that are filtered out for mining association rules. One may specify some kinds of items as stop words for certain scenarios or for experiments. For example, suppose we have only the search results of one search engine X, the "SE:X" item can then be filtered out as stop words. Otherwise, we may get many uninteresting rules that say some items imply "SE:X".

Our approach also allows to specify the left-hand-side patterns, i.e., what kind of items can be employed in the left-hand-side of rules. A pattern can represent many possible items, e.g., a "top10:" pattern represents all possible items for top 10 search results. In this way, we do not need to enumerate all possible items in advance. In general, any item could be interesting to be part of the left-hand-side itemset, while sometimes one may want to check the rules with specific left-hand-side items.

Similarly, our approach allows to specify the right-hand-side patterns. One may be more interested in what properties of the search results hold, other than what properties of the queries hold, under certain conditions. For this purpose, one can specify all items except the items of the query category as the right-hand-side patterns.

Our approach also consists of an implication checker is to filter out rules whose the left-hand-side itemsets imply the right-hand-side itemsets by nature. For example, consider a rule whose left-hand-side contains "top1:X" and the right-hand-side is "top10:X", where X is some URL. The confidence of the rule is definitely 1.0. Such a rule is uninteresting and can be pruned,

because it does not provide any information of the search results. However, if one is only interested in checking the violations, it is fine to keep such rules as they do not have any violations to check.

5.4.4 Detecting Violations of Mined Rules

The mined rules can be applied to detect violations in any given set of search results automatically, including the set of search results where the rules are mined.

Our approach ranks the mined rules in descending order of confidence and support. Given a set of search results, our approach maps them to a database of itemsets, and then check the rules as follows. From the top to bottom, our approach picks a rule and check it against all the itemsets. If the rule is violated by any itemset, which represents a query and its search results, our approach outputs the rule as well as the violations. The testers can then examine the violated rule and the violations together.

5.4.5 Learning to Classify Search Results

Because the expected search results may change, testers need to check the search results repeatedly. In the result inspection process, we can get a lot of labels about the search results, i.e., whether the search fail or pass. In this case, the mining approach cannot utilize these labels effectively. Instead, we can use classification techniques such as Decision Tree to learn models to classify the search results. The approach consists of two steps: training and testing.

In the training step, we suppose the testers have collected a set of failing and passing labels of the search results. The exact meaning of failing and passing should be determined by the testers. Given a set of queries, search results, and the labels, we can extract items (i.e., features) from queries and

search results, and apply machine learning techniques such as Decision Tree and Logistic Regression to learn classification models.

In the testing step, we use the learnt classification models to classify new search queries and search results. A search query and its search results also need to be transformed into vectors of items (features), and the classification model will classify the search query and its search results as failing or passing directly.

Using the classification techniques, we can learn classification models of failing tests and find suspicious search results automatically. The classification models of failing tests can help to identify the important factors that lead to poor performance of the search engine under test. Moreover, testers can manually examine the suspicious search results to get failing tests for debugging.

5.5 Evaluation

In this section, we describe the process of collecting a set of queries and search results of different search engines in several months. We then describe the results of applying our approach on the collected data to mine test oracles and find suspicious search results automatically.

5.5.1 Data Collection

Queries

We collect two sets of queries for the evaluation. The first set of queries come from the dataset of KDD-Cup 2005 Competition [78]. The KDD-Cup 2005 Competition is in the area of search query categorization. In the competition, 800,000 search queries were randomly selected from MSN search logs with some preliminary filtering. Among them, 800 non-junk queries were further

randomly selected by the organizers for evaluation. We use the 800 queries as a representative set of common queries, which may cover a more wide range of topics.

The second set of queries are collected from Google Trends and Yahoo! Buzz. These two indexes provide the hottest queries submitted to the corresponding search engine everyday. A hot query is a query in which people searches most often in a period of time. Different from common queries, hot queries often refer to recent important events. We have collected the hot queries from November 25, 2010 to April 21, 2011. There are in total 3432 unique hot queries.

Search Results

We collect the search results of the prepared queries from December 25, 2010 to April 21, 2011. We applied the Web services of Google and Bing to collect the top 10 search results of each query every day. We employed these Web services because using them to monitor a large number of queries is more friendly to search engines. But the number of queries that can be submitted every day is still limited by the policies of search engines. Because a hot query may not be hot after some days, in each day we collect search results of only the recent hot queries, i.e. hot queries that appeared not later than 30 days before. In this way, we can capture the trends of search results of hot queries in the very beginning of the associated hot events, without violating search engines' policies. In total, we collect 390797 ranked lists of search results (each list contains the top 10 search results of a query).

5.5.2 Mining Rules

Mining from One Search Engine' Results in One Day

We first apply our approach to the simplest scenario, where only one search engine's search results in one day is available. We mine rules from Google's search results on December, 25, 2010. We set $minsup = 20$, $minconf = 0.95$, and $maxL = 3$ (the maximum length of rules). We also specify that all items that are not of query types are interesting right-hand side patterns, and the items of "SE:" are stop words (since there is only a search engine).

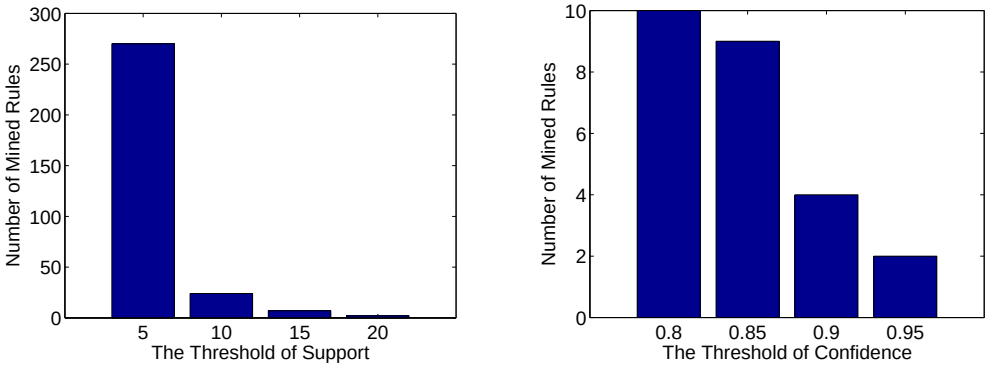
The mining results contain 2 rules, as shown below.

```
1.top10:starpulse.com,HotQ, => top10:imdb.com, : 22/22=1.0
2.top10:starpulse.com,TwoWords, => top10:imdb.com, : 22/23=0.96
```

Rule 1 says that there are 22 itemsets (queries and results) where the top 10 search results contain starpulse.com (an entertainment Website) and the query is a hot query. In all these 22 itemsets, the top 10 search results always contains imdb.com (an online movie database). The first condition of the rule may imply that the query is related to entertainment, and the second condition means that the query is about something popular recently. Under these two conditions, imdb.com has a high chance to have relevant content for the query. Besides, imdb.com is an authoritative Website. Therefore, it is reasonable for imdb.com to be ranked in top 10 for the query. Rule 2 is also about the implication between starpulse.com and imdb.com, but the causality is less clear and there is a violation for this rule.

If we relax the constraints by setting $minsup = 10$, the results will contain 24 rules. All these rules describe the implications between different Web domains. Some other examples rules are shown below. Because there are only search results of one search engine in one day, other potential rules such as rules about the top 1 search results are not frequent enough to be mined.

```
3.TwoWords,top10:last.fm, => top10:youtube.com, : 18/18=1.0
4.top10:rotoworld.com,TwoWords, => top10:sports.espn.go.com, : 14/14=1.0
5.TwoWords,top10:nfl.com, => top10:sports.yahoo.com, : 10/10=1.0
```



(a) Different threshold of support

(b) Different threshold of confidence

Figure 5.3: Number of rules with different thresholds

Figure 5.3(a) shows the effects of adjusting the thresholds of support while fixing minconf=0.95. Figure 5.3(b) shows the effects of adjusting the thresholds of confidence while fixing minsup=20. We observe that reducing the minsup could significantly increase the number of mined rules, while reducing the minconf has a relatively small effect.

Mining from Multiple Search Engines' Results in One Day

We next apply our approach to the search results of multiple search engine, i.e., Google and Bing, in one day. Again, we mine rules from search results on December, 25, 2010. We use the same settings as previous experiments except that we do not specify "SE:" items as stop words.

The mining results contain 24 rules, some of which are shown below. Rules 6-8 show three kinds of general changes of the mined rules. First, Rule 6 is the same pattern as Rule 1, but the number of its supporting itemsets increases from 22 to 24. Second, the previous Rule 2 "top10:starpulse.com,TwoWords,

⇒ top10:imdb.com," is removed because of the decrease in its confidence. Third, Rules 7 and 8 are added due to the increase of their supporting itemsets.

```
6.top10:starpulse.com,HotQ, => top10:imdb.com, : 24/24=1.0
7.HotQ,top10:movies.yahoo.com, => top10:imdb.com, : 20/20=1.0
8.TwoWords,top10:tvguide.com, => top10:imdb.com, : 23/24=0.96
9.top10:absoluteastronomy.com, => SE:bing, : 63/63=1.0
10.top10:thirdage.com, => SE:bing, : 40/40=1.0
11.TwoWords,top10:youtube.com, => SE:google, : 137/143=0.95
12.OneWord,top10:twitter.com, => SE:google, : 28/29=0.97
```

Perhaps more importantly, mining from the results of the two search engines help to discover a new kind of rules, i.e., rules whose right-hand side is an "SE:" item. Rules 9-12 belong to this kind of rules. They show the different opinions of search engines to certain Websites. Rules 9 and 10 say that if the top 10 results contain "absoluteastronomy.com" or "thirdage.com", the search engine is likely to be Bing (the confidence is 1.0). In other words, Google seldom ranks the Website as one top 10 result for queries. Oppositely, Rules 11 and 12 show that while Google often ranks "youtube.com" and "twitter.com" as a top 10 result for queries, Bing seldom does it. When a search engine SE1 often gives a high rank for a Web domain W while another search engine SE2 does not, there are two possible reasons. One is that W is a good Web site that SE2 does not notice or has difficulties in crawling. The other is that W is not that good but SE1 shows bias towards W, either intentionally or being cheated. In either cases, such rules are helpful for understanding the differences of search engines, and may help testers find drawbacks of search engines' spiders or ranking functions.

Mining from One Search Engine' Results in Multiple Days

We next apply our approach to the search results of one search engine in multiple days. We mine rules from Google's search results from December, 25, 2010 to March 31, 2011. There are much more search results and thus much more itemsets for mining rules. We thus use stricter constraints by setting $minsup = 200$, $minconf = 0.95$, and $maxL = 2$ for general rules. Note for rules that indicate the best top 1 search results of queries, there may still be few supporting documents. Therefore, we mine this kind of rules separately. In particular, we specify that the left-hand side should be some query ("Q:" items) and the right-hand side should be some top 1 search result ("top1:" items), and set $minsup = 20$, $minconf = 0.95$, and $maxL = 2$.

The mining results contain 58 general rules, mainly describing the implications between Web domains with higher number of supporting documents. We have shown some examples of this kind of rules previously and further examples are omitted here.

```
13.Q:hulu, => top1:hulu.com, : 91/91=1.0
14.Q:facebook, => top1:facebook.com, : 91/91=1.0
15.Q:youtube, => top1:youtube.com, : 91/91=1.0
16.Q:rosenbluth, => top1:rvacations.com, : 91/91=1.0
17.Q:espn picks, => top1:espn.go.com, : 91/91=1.0
18.Q:stock futures, => top1:bloomberg.com, : 91/91=1.0
```

The mining results contain 1597 top 1 rules. We next describe some examples as shown above. Rule 13 shows that for the query "hulu", "hulu.com" (a famous video Website) is always ranked top 1 by the search engines in the collected search results. Similarly, the queries in Rules 14 and 15 have well-known meanings and the most suitable top 1 search results. Rules 16-18 show queries whose meaning are more ambiguous. However, from search results in a series of time, one can have a high confidence that the corresponding search results are suitable to be ranked top 1. Violations of such rules, which might

be caused by spamming or phishing websites, can confuse users and cause user dissatisfaction. On the other hand, many queries may not have stable top 1 results since their meanings are ambiguous and the Web data keep changing. Therefore, it is important to identify whether a query has the most suitable top 1 search result automatically.

Mining from Multiple Search Engines' Results in Multiple Days

Finally, we apply our approach to search results of two search engines in multiple days. We mine rules from search results of Google and Bing during December, 25, 2010 to March 31, 2011. The settings are the same as those of mining from Google's search results in many days, i.e., $minsup = 20$ for top 1 rules and $minsup = 200$ for other rules, except that "SE:" items are not specified as stop words.

The mining results contain 1341 top 1 rules and 275 other general rules. The number of top 1 rules is smaller than that of top 1 rules mined from only search results of Google. This is because when mining from search results of a search engine, a rule has a high confidence once the search engine agrees with it, no matter whether other search engines agree or not. But when mining from search results of both search engines, a rule has a high confidence only if both search engines agree with it. Therefore rules mined from search results of multiple search engines are more reliable. The number of other general rules is larger than that of general rules mined from only search results of Google. This is because in this case there are many meaningful rules whose right-hand side is an "SE:" item.

```
19.top10:absoluteastronomy.com, => SE:bing, : 7657/7657=1.0
20.top10:quotes.nasdaq.com, => top10:finance.yahoo.com, : 314/314=1.0
21.top10:finapps.forbes.com, => top1:finance.yahoo.com, : 262/262=1.0
22.Q:facebook, => top1:facebook.com, : 182/182=1.0
```

Some examples of the mined rules are shown as above. Most of the rules belong to the types that we have described, which describe implications between Web domains, different opinions of search engines to certain Web domain, and the best top 1 search results of queries. But mining from multiple search engines in multiple days can further reduce false positives of rules and generate more reliable rules.

5.5.3 Detecting Violations

In this section, we examine the effectiveness of the mined rules for finding suspicious search results. We use the rules mined from search results of multiple search engines in multiple days, as described in Section 5.5.2. Our approach checks the mined rules against the search results of Google and Bing between April 1, 2011 to April 22, 2011. We next describe an example violation of the mined rules. We then check the number of queries that violate rules, compared with those of queries that change results.

Example Violations

When checking search results of Bing on April 1st, 2011, our approach finds that search results of Bing violate a rule as follows:

```
Q:where to login to john carroll university email, =>
top1:mirapoint.jcu.edu, : 172/180=0.96
```

The high confidence of this rule suggests that for the query "where to login to john carroll university email", the URL "mirapoint.jcu.edu" is often the best answer. Both Google and Bing agree on this result for most of the time. We check this url manually, and find that it is the entrance of the webmail system of the John Carroll University. The investigation confirms

that "mirapoint.jcu.edu" is the best top 1 result for the query in these days.

But on April 1st, 2011, Bing violates this rule. We check the search results of the query of Bing in that day. The top 1 search result of Bing is the URL "http://www.jcu.edu/index.php". Although this URL points to the homepage of the John Carroll University, it is not easy to get the answer of the query, i.e., the entrance of the mail system, from this URL. Therefore, the change of the top 1 search result for this query is inadequate.

There are various factors that affect the ranks of the two URLs above. Since we are not aware of the exact ranking algorithm of Bing, we cannot figure out what the exact problem is. However, we believe that collecting such suspicious cases automatically can help search engine developers in examining the ranking algorithm, either in the experimental stage or in the deployment stage.

Number of Queries Violating Rules

Figure 5.4 shows the number of queries, which violate some rules, for Google and Bing in different days. As shown in Figure 5.4, Bing has more queries that violate some rules than Google. Figure 5.4 also shows the number of queries that change results. We say a query changes results if at least one top 10 search result has a different rank from its previous rank. It is the basic approach for regression testing to check changed outputs. However, we can see that there are too many queries that change results to be manually investigated. On the other hand, the number of queries that violate mined rules is much smaller for both Google and Bing.

Table 5.2 shows the average numbers of common and hot queries that violate rules, compared with numbers of queries that change results. We can see that the number of queries that violate rules is often less than 11% of the number of queries that change results. There are more hot queries that change

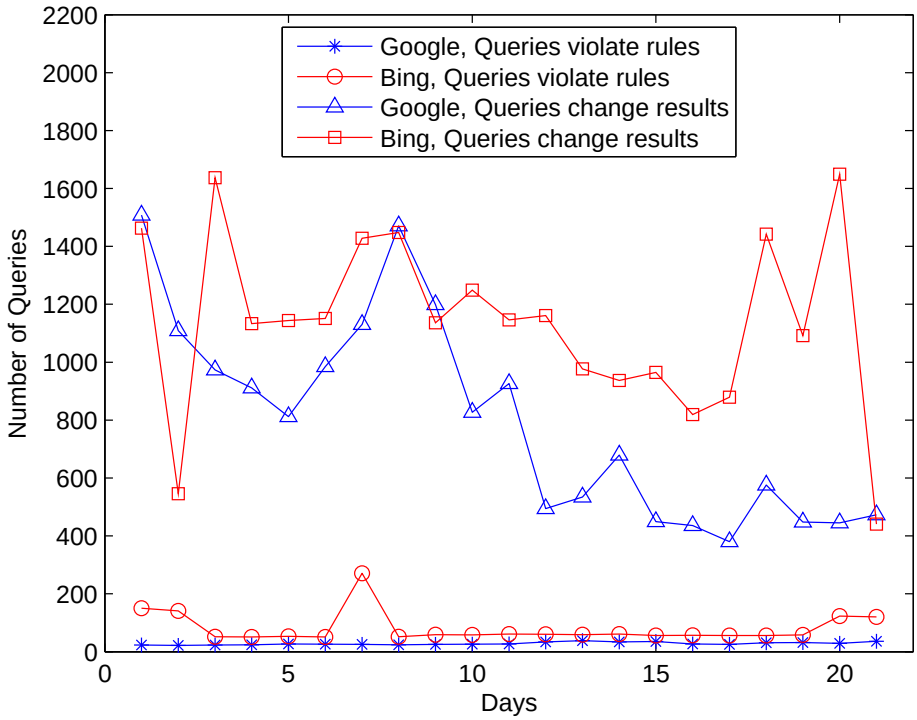


Figure 5.4: Numbers of queries that violate rules or change results

results than common queries, but the number of hot queries that violate rules is smaller than that of common queries. The reason may be that the Web data of hot queries keep changing, and there are few established rules for these queries to be violated. We also note that Bing have more queries that change results, i.e., Bing's search results are more unstable.

5.5.4 Learning Classification Models

We also conduct experiments to evaluate the approach of learning classification models. We do not have the real labeling of the search results. To check the feasibility of the classification approach, we experiment with the following classes: for a query, if the top 1 search result of the search engine under test

Table 5.2: Average numbers of queries that violate rules or change results

	#Queries that have violations	#Queries that change results	The ratio of violations to changes
Google, Common Queries	14	319	4.4%
Google, Hot Queries	14	480	2.9%
Bing, Common Queries	50	453	11.0%
Bing, Hot Queries	31	682	4.5%

Table 5.3: Results of predicting abnormal search result changes

Models	Data	Abnormal Data	Accuray	Precision	Recall
Decision Tree	3429	921	0.72	0.47	0.42
Naive Bayes	3429	921	0.66	0.36	0.38

changes, but the other search engines do not agree with the change (they returned the same top 1 result and do not change), we say the search result's change is unexpected, otherwise, it is normal. The classification task is: given a query, the previous top 1 search result, and the current top 1 result, predict whether the search result's change is unexpected.

We use the items described previously as the features. Each query with changed top 1 search result yields an instance x, which consists of the features of the query, the previous top 1 search result, and the new top 1 search result. We use the data between December 26, 2010 to March 31, 2011 as the training data, and use the data between April 1, 2011 to April 22, 2011 as the testing data.

We use two classification models: Decision Tree and Naive Bayes. The

classification results are shown in Table 5.3. As shown in the table, both the two models can improve the chance of finding abnormal search results (the precision of random classification should be 921/3429=27%). The Decision Tree model is more effectively than the Naive Bayes model. This may be because the different features of the data are not independent, which violates the assumption of the Naive Bayes model. However, both the precision and recall of the models are not high enough. A possible solution is to provide more training data and to utilize more features for learning. We also note that the Naive Bayes model runs much faster than the Decision Tree model, and thus can be applied to large scale data easily. The experiments show the feasibility of applying the classification models to find abnormal search results automatically.

5.5.5 Discussions

The evaluation results suggest the following observations:

- Our approach is effective in mining high confidence rules, which describe different implications between queries, search results, and search engines.
- Our approach can be applied to different scenarios. It can mine meaningful rules even when there are only search results of one search engine in one day. It can mine more reliable rules and more kinds of rules when more information is available.
- The mined rules help to detect suspicious results that may help analyze potential defects of search engines.

However, the current design of properties about queries and search results can be further improved. Search engines tend to produce more outputs for a query other than the traditional ranked list of search results. For example,

the outputs of a query often contain advertisements and query suggestions. There are also embedded search results for News, images, videos, messages in the twitter Web site, and airticket information. That is, for News queries, some News link may be returned as well as the top 10 Web page results [72]. These kinds of search results also affect the retrieval effectiveness perceived by the users. We plan to extend our design of properties about search results for these more general outputs. More over, search engines may provide personalized rankings. We need to incorporate properties about the personalized, anonymous information into our approach.

5.6 Summary

It is of great importance to test the retrieval effectiveness of search engines. However, it is challenging to test search engines due to the lack of test oracles. In this work, we propose to mine test oracles of Web search engines from existing search results. We define a set of properties of queries and search results, and mine frequent association rules between these properties as test oracles. We also design a set of controlling schemes for rule generation to ease the mining of different kinds of specifications. We collect a data set that contains the search results of two major search engines, namely Google and Bing, for 4232 queries in a period of 4 months. Evaluation on this data set shows that our approach mines many high confidence rules that describe different implications between queries and search results. The mined rules also help to detect suspicious results that may help analyze potential defects of search engines.

□ **End of chapter.**

Chapter 6

Conclusions

In this chapter, we summarize the key research results presented in this thesis, and discuss some possible future research work.

6.1 Summary

The contribution of this thesis is to improve the effectiveness of automatic software testing by mining specifications from software data. Although there has been some existing work on mining specifications [23, 45, 56], these approaches do not target at using the mined specifications to help automatic software testing. In this thesis, we consider the data available in different scenarios of software testing, and propose new approaches to mine specifications from these data. The mined specifications are then used to generate test inputs and to verify test outputs. Figure 1.1 presents an overview of the work in this thesis. We have mined relevant APIs, common operational models, and input/output rules from the source code, execution traces, and existing inputs/outputs, respectively. We have used the mined specifications to help the tasks of random unit-test generation and test selection for result inspection. All the proposed approaches have been implemented and evaluated on a set of software programs.

This thesis aims to improve the effectiveness of automatic software testing by mining specifications from software data. In particular, we focus on two tasks: automatic test input generation and test output inspection. We consider the data available in different scenarios of software testing, and propose new approaches to mine specifications from these data. The mined specifications are then used to generate test inputs and to select tests for result inspection. The major achievements and contributions are concluded in the following.

First, we mine relevant APIs from source code to guide random unit-test generation. Given a method under test (MUT), a key component of object-oriented unit-test generation is to find method-call sequences that create and mutate desired inputs. We present an approach, called RecGen, to mine relevant APIs to guide method-call sequence generation. We develop a static analysis module based on Eclipse JDT Compiler [2] to mine relevant APIs that may access the same object fields from the source code under test. Based on these relevant APIs, RecGen recommends short sequences that mutate object fields accessed by a method under test (MUT) to generate the inputs. Evaluation results show that RecGen can improve the code coverage and fault-revealing capability over previous random testing tools.

Second, we mine common operational models from execution traces of unverified tests to select tests for result inspection. Previous work on mining operational models for test selection is based on dynamic invariant detection, which relies on a set of existing passing tests. Differently, our approach mines common operational models, which are often but not always true in all observed traces, from a (potentially large) set of unverified tests. In particular, our approach collects branch coverage and data value bounds at runtime and then mines implication relationships between branches and constraints of data values as potential operational models after running all the tests. Our

approach then selects tests that violate the mined common operational models for result inspection. The experimental results, compared with previous test selection approaches, show that our approach can more effectively reduce the number of tests for result inspection while revealing most of the faults.

Third, we mine pseudo test oracles of Web search engines from existing inputs (Web queries) and outputs (search results). Unlike previous work of mining specifications that focuses on properties of the implementation, our approach focuses on the functionalities of the search engines. We define a set of properties of queries, search results, matches between queries and search results, and search engine identities. We then mine frequent association rules between these properties as test oracles. To facilitate the mining process, we also propose a set of controlling schemes for rule generation. Experiments on major search engines show that our approach mines many high confidence rules that help to understand search engines and detect suspicious search results.

6.2 Future Work

In this thesis, we have proposed several approaches to mine specifications from software data to guide automatic software testing in the absence of specifications. Despite of the initial achievements, there are still numerous open issues that need to be further explored in future work.

First, there could be many ways to mine software data for improving the effectiveness of specification-based testing. For example, we may mine the "interesting" values of input parameters from the source code and existing executions (existing tests or users' feedback report). These parameter values can then be used for combinatorial testing. We may also mine the transition probability between different states of the system under test from existing executions. We can then build a probabilistic finite state machines that re-

flect the usage profiles to generate new tests. Similarly, we may mine the probability of using different production rules and the typical sentences of each non-terminal. These information can help to generate new tests that represent the potential usage patterns automatically. Moreover, different applications may share some kinds of basic inputs, such as regular expressions, databases, web pages of some format. We may build a common repository of such common inputs and select representative ones from them for multiple applications.

Second, we may improve the applicability of symbolic execution by predicting the benefit (e.g. improvement of code coverage) of exploring a node of the control flow graph in a given context. One of the major challenges of symbolic execution is the large search space of control paths of nontrivial programs. We may build a prediction model of the benefit of exploring a subgraph, so as to guide the exploration of symbolic executions. The candidate factors could include the previous exploration results of the node in other contexts, the distance of the node to a uncovered code entity, etc. The training data could be the exploration history of the program, or the exploration results of other programs.

Third, we may build classifiers that predict failures of software programs based on the program's properties and the manual labels. As shown in our work of search engine testing, there are some kinds of applications that need to be tested continuously and the expected results are unstable. Patterns mined from program's properties can help to identify the most suspicious tests for manual inspection. However, due to the consideration of mining efficiency, the patterns are often in a simple form such that they may not be flexible enough and could result in many false positives. The labels collected during the manual inspection can help to build more accurate classification models to identify suspicious tests. As the testing proceeds, this approach

could leverage the more and more collected labels for effective test output inspection.

□ **End of chapter.**

Bibliography

[1] comscore. http://comscore.com/Press_Events/Press_Releases/2010/10/comScore_Releases_September_2010_U.S._Search_Engine_Rankings.

[2] Eclipse Java development tools (JDT). `http://www.eclipse.org/jdt/index.php`.

[3] FindBugs: Find Bugs in Java Programs. `http://findbugs.sourceforge.net/`.

[4] Google's core libraries. `http://code.google.com/p/guava-libraries/`.

[5] googletest: Google C++ Testing Framework. `http://code.google.com/p/googletest/`.

[6] HP QuickTest Professional. `http://en.wikipedia.org/wiki/HP\_QuickTest\_Professional`.

[7] http://jscience.org/.

[8] http://sir.unl.edu/php/index.php.

[9] http://www.eclemma.org/.

[10] http://www.google.com/codesearch.

[11] http://www.jdsl.org/.

[12] http://www.oracle.com/technology/products/berkeley-db/index.html.

[13] Junit testing framework. http://www.junit.org/.

[14] Minisat sat solver. http://minisat.se/.

[15] Selenium web application testing system. http://seleniumhq.org/.

[16] Text REtrieval Conference.
`http://trec.nist.gov/`.

[17] The Public Suffix List.
`http://publicsuffix.org/`.

[18] Watir: Web application testing in ruby. http://watir.com/.

[19] www.math.unb.ca/ knight/utility/NormTble.htm.

[20] J.-R. Abrial, S. A. Schuman, and B. Meyer. Specification language. In *On the Construction of Programs*, pages 343–410. 1980.

[21] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE*, pages 143–151, 1995.

[22] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 487–499. Morgan Kaufmann, 1994.

[23] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.

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

[25] G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. In *ISSTA*, pages 189–200, 2008.

[26] V. R. Basili, L. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22:751–761, 1995.

[27] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *ISSTA*, pages 195–205, 2004.

[28] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Testing concurrent object-oriented systems with spec explorer. In *FM*, pages 542–547, 2005.

[29] B. Carterette, E. Kanoulas, and E. Yilmaz. Low cost evaluation in information retrieval. In *SIGIR*, page 903, 2010.

[30] W. Chan, S. Cheung, and K. R. Leung. A metamorphic testing approach for online testing of service-oriented software applications. *Special Issue on Services Engineering of International Journal of Web Services Research*, pages 60–80, 2007.

[31] J. Chang and D. J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *ESEC/SIGSOFT FSE*, pages 285–302, 1999.

[32] M. Chen, M. R. Lyu, and E. Wong. Effect of code coverage on software reliability measurement. *IEEE Trans. on Reliability*, 50(2):165–170, 2001.

[33] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.

[34] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Artoo: adaptive random testing for object-oriented software. In *30th International Conference on Software Engineering*, pages 71–80, 2008.

[35] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Artoo: adaptive random testing for object-oriented software. In *ICSE*, pages 71–80, 2008.

[36] D. Cohen, I. C. Society, S. R. Dalal, M. L. Fredman, and G. C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23:437–444, 1997.

[37] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Softw., Pract. Exper.*, 34(11):1025–1050, 2004.

[38] J. Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios. http://msdn.microsoft.com/en-us/library/cc150619.aspx, February 2008.

[39] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *ECOOP*, pages 528–550, 2005.

[40] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[41] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE*, pages 339–348, 2001.

[42] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *ESEC/SIGSOFT FSE*, pages 246–255, 2001.

[43] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Software Eng.*, 28(2):159–182, 2002.

[44] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

[45] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):99–123, 2001.

[46] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, 1997.

[47] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Software Eng.*, 14(10):1483–1498, 1988.

[48] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *SIGSOFT FSE*, pages 339–349, 2008.

[49] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[50] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Softw. Test., Verif. Reliab.*, 15(3):167–199, 2005.

[51] J. Han and M. Kamber. *Data Mining: Concepts and Techniques, 2nd ed.* Morgan Kaufmann Publishers, 2006.

[52] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, pages 291–301, 2002.

[53] M. Haran, A. F. Karr, A. Orso, A. A. Porter, and A. P. Sanil. Applying classification techniques to remotely-collected program execution data. In *ESEC/SIGSOFT FSE*, pages 146–155, 2005.

[54] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *ICSE*, pages 60–73, 2003.

[55] A. E. Hassan and T. Xie. Mining software engineering data. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010), Companion Volume, Tutorial*, pages 503–504, May 2010.

[56] J. Henkel and A. Diwan. Discovering algebraic specifications from java classes. In *ECOOP*, pages 431–456, 2003.

[57] M. Hennessy. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In *In Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 104–113, 2005.

[58] H.-Y. Hsu and A. Orso. Mints: A general framework and tool for supporting test-suite minimization. In *ICSE*, pages 419–429, 2009.

[59] J. C. Huang. An approach to program testing. *ACM Comput. Surv.*, 7(3):113–128, 1975.

[60] W. Humphrey. The future of software engineering: I. March,2001.

[61] M. Hutchins, H. Foster, T. Goradia, and T. J. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, pages 191–200, 1994.

[62] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: Relative significance rank for software component search. In *ICSE*, pages 14–24, 2003.

[63] J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2008.

[64] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, 2002.

[65] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC/SIGSOFT FSE*, pages 55–64, 2007.

[66] T. Joachims. Optimizing search engines using clickthrough data. In *KDD*, pages 133–142, 2002.

[67] T. Joachims. Evaluating retrieval performance using clickthrough data. In *Text Mining*, pages 79–96. 2003.

[68] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, 2002.

[69] K. S. Jones and C. van Rijsbergen. Report on the need for and provision of an "ideal" information retrieval test collection.

[70] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *ISSTA*, pages 105–116, 2009.

[71] S. Kim, E. J. W. Jr., and Y. Z. 0001. Classifying software changes: Clean or buggy? *IEEE Trans. Software Eng.*, 34(2):181–196, 2008.

[72] A. C. König, M. Gamon, and Q. Wu. Click-through prediction for news queries. In *SIGIR*, pages 347–354, 2009.

[73] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter. Combinatorial software testing. *IEEE Computer*, 42(8):94–96, 2009.

[74] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *TestCom*, pages 19–38, 2006.

[75] F. Lanubile, A. Lonigro, and G. Visaggio. Comparing models for identifying fault-prone software components. In *IN PROCEEDINGS OF THE 7TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING*, pages 312–319, 1995.

[76] D. Leon, W. Masri, and A. Podgurski. An empirical evaluation of test case filtering techniques based on exercising complex information flows. In *ICSE*, pages 412–421, 2005.

[77] H. Li, M. Jin, C. L. 0002, and Z. Gao. Test criteria for context-free grammars. In *COMPSAC*, pages 300–305, 2004.

[78] Y. Li, Z. Zheng, and H. K. Dai. Kdd cup-2005 report: facing a great challenge. *SIGKDD Explorations*, 7(2):91–99, 2005.

[79] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, pages 289–302, 2004.

[80] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–315, 2005.

[81] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, 2003.

[82] B. R. Liblit. *Cooperative Bug Isolation.* PhD thesis, University of California, Berkeley, Dec. 2004.

[83] Y. Lin, X. Tang, Y. Chen, and J. Zhao. A divergence-oriented approach to adaptive random testing of java programs. In *ASE*, pages 221–232, 2009.

[84] D. Lo and S.-C. Khoo. Smartic: towards building an accurate, robust and scalable specification miner. In *SIGSOFT FSE*, pages 265–275, 2006.

[85] F. Long, X. Wang, and Y. Cai. Api hyperlinking via structural overlap. In *ESEC/SIGSOFT FSE*, pages 203–212, 2009.

[86] M. R. Lyu. *Handbook of Software Reliability Engineering*. McGraw-Hill, 1996.

[87] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 48–61, 2005.

[88] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Software Eng.*, 33(1):2–13, 2007.

[89] A. Michail and T. Xie. Helping users avoid bugs in gui applications. In *ICSE*, pages 107–116, 2005.

[90] C. Murphy, K. Shen, and G. E. Kaiser. Automatic system testing of programs without test oracles. In *ISSTA*, pages 189–200, 2009.

[91] E. R. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *ICSE*, pages 287–297, 2009.

[92] G. J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.

[93] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *ICSE*, pages 452–461, 2006.

[94] P. A. Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound boolean predicates. In *ISSTA*, pages 5–15, 2007.

[95] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.

[96] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.

[97] N. I. of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. 2002.

[98] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *SIGSOFT FSE*, pages 241–251, 2004.

[99] T. J. Ostrand and E. J. Weyuker. The distirubtion of faults in a large industrial software system. In *ISSTA*, pages 55–64, 2002.

[100] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP*, pages 504–527, 2005.

[101] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84, 2007.

[102] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification And Reliability*, 9:263–282, 1999.

[103] P. Purdom. A sentance generator for testing parsers. *BIT*, 12(3):366–375, 1972.

[104] T. W. Reps, T. Ball, M. Das, and J. R. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ESEC / SIGSOFT FSE*, pages 432–449, 1997.

[105] M. P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 11–20, 2005.

[106] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Trans. Software Eng.*, 27(10):929–948, 2001.

[107] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. G. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *ICSE*, pages 341–350, 2008.

[108] Z. M. Saul, V. Filkov, P. T. Devanbu, and C. Bird. Recommending random walks. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 15–24, 2007.

[109] A. Schr?ter, T. Zimmermann, R. Premraj, and A. Zeller. If your bug database could talk... (short paper). In *Proceedings of the 5th International Symposium on Empirical Software Engineering. Volume II: Short Papers and Posters*, pages 18–20, September 2006.

[110] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *ISSTA*, pages 69–80, 2009.

[111] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006.

[112] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[113] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE*, pages 204–213, 2007.

[114] S. Thummalapenta and T. Xie. Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. In *ASE*, pages 327–336, 2008.

[115] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *ASE*, pages 283–294, 2009.

[116] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *TAP*, pages 134–153, 2008.

[117] P. Tonella. Evolutionary testing of classes. In *ISSTA*, pages 119–128, 2004.

[118] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.

[119] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA*, pages 97–107, 2004.

[120] K. Wang, T. Walker, and Z. Zheng. Pskip: estimating relevance ranking quality from web search clickthrough data. In *KDD*, pages 1355–1364, 2009.

[121] X. Wang, S. Cheung, W. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proc. 31th International Conference on Software Engineering (ICSE 2009)*, May 2009.

[122] X. Wang, Z. Guo, X. Liu, Z. Xu, H. Lin, X. Wang, and Z. Zhang. Hang analysis: fighting responsiveness bugs. In *EuroSys*, pages 177–190, 2008.

[123] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *ISSTA*, pages 249–260, 2008.

[124] A. W. Williams and R. L. Probert. A measure for component interaction test coverage. In *AICCSA*, pages 304–312, 2001.

[125] W. E. Wong, Y. Shi, Y. Qi, and R. Golden. Using an rbf neural network to locate program bugs. In *ISSRE*, pages 27–36, 2008.

[126] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 196–205, 2004.

[127] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *ASE*, pages 40–48, 2003.

[128] T. Xie and D. Notkin. Automatically identifying special and common unit tests for object-oriented programs. In *ISSRE*, pages 277–287, 2005.

[129] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. In *ICSE*, pages 282–291, 2006.

[130] Y. Yue, R. Patel, and H. Roehrig. Beyond position bias: examining result attractiveness as a source of presentation bias in clickthrough data. In *WWW*, pages 1011–1018, 2010.

[131] Q. Zhang, W. Zheng, and M. R. Lyu. Flow-augmented call graph: A new foundation for taming api complexity. In *FASE 2011, Fundamental Approaches to Software Engineering*, pages 386–400, 2011.

[132] W. Zheng, M. R. Lyu, and T. Xie. Test selection for result inspection via mining predicate rules. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 219–222, 2009.

[133] W. Zheng, H. Ma, M. R. Lyu, T. Xie, and I. King. Mining test oracles of web search engines.

[134] W. Zheng, Q. Zhang, M. R. Lyu, and T. Xie. Random unit-test generation with mut-aware sequence recommendation. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 293–296, 2010.

[135] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference*, pages 318–343, 2009.

[136] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.