ABSTRACT

THUMMALAPENTA, SURESH. Improving Software Productivity and Quality via Mining Source Code. (Under the direction of Tao Xie.)

The major goal of software development is to deliver high-quality software efficiently. To achieve this goal of delivering high-quality software efficiently, programmers often reuse existing frameworks or libraries, hereby referred to as libraries, instead of developing similar code artifacts from the scratch. However, programmers often face challenges in reusing existing libraries due to two major factors. First, many existing libraries are not well-documented. Even when such documentations exist, they are often outdated. Second, many existing libraries expose a large number of application programming interfaces (APIs), which represent interfaces through which libraries expose their functionalities. For example, the .NET base library provides nearly 10,000 API classes. Due to these two preceding factors, there exist three major problems that affect both software productivity and quality. First, programmers often spend more time in reusing existing libraries, thereby reducing software productivity. Second, programmers introduce defects while using APIs due to lack of proper knowledge on how to reuse those APIs. Third, existing white-box test generation techniques face challenges in effectively generating test inputs for the client code that reuses libraries.

To address these three preceding issues, in this dissertation, we propose a general framework, called *WebMiner*, that uses existing open source code available on the web by leveraging a code search engine. In particular, WebMiner infers usage specifications for API methods under analysis by automatically collecting relevant code examples from the open source code available on the web. WebMiner next applies data mining techniques on those collected code examples to identify common patterns, which represent likely usage of APIs, referred to as API usage specifications. The primary reason for identifying common patterns is based on the observation that majority of the programmers correctly adhere to API usage specifications and those common patterns are likely to represent the correct usage of APIs.

We further propose six approaches based on our general framework, where each approach focuses on a specific software engineering (SE) task such as detecting defects in an application under analysis. In particular, the first two approaches assist programmers in effectively reusing APIs provided by existing libraries. The next two approaches use mined API usage specifications as programming rules and detect defects in applications under analysis as deviations from the mined specifications. Finally, the last two approaches mine static and dynamic traces, respectively, for effectively generating test inputs that achieve high structural coverage of the code under test. We also propose another approach that addresses a major issue with mining-based approaches, which are not effective in scenarios where usage information is not

available for the API methods under analysis or usage information is not sufficient to achieve the SE task under analysis.

Our empirical results show that the approaches developed based on our WebMiner framework effectively address the respective SE tasks handled by those approaches. In particular, our empirical results demonstrate the effectiveness of expanding the data scope of mining-based approaches to large open source code available on the web. Our results also show that our approaches address queries posted in developer forums and detect new defects that are not detected by existing related approaches, thereby improving both software productivity and quality.

Improving Software Productivity and Quality via Mining Source Code

by
Suresh Thummalapenta

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

Computer Science

Raleigh, North Carolina

2011

APPROVED BY:

Mladen Vouk	Laurie Williams
Yan Solihin	Tao Xie Chair of Advisory Committee

DEDICATION

To my parents, Venkata Ramana and Adilakshmi.

BIOGRAPHY

Suresh Thummalapenta was born in Anakapalle, Andhra Pradesh, India. He completed his Master of Science degree in 2008 from North Carolina State University and Bachelor of Technology in 2001 from Andhra University, India, all in Computer Science. His primary research interest is software engineering with a primary goal to develop techniques and tools that can help in developing high-quality software more productively. His current research investigated problems in the software engineering domain and developed techniques not only in the areas of program analysis and software testing, but also adapted and refined techniques from other areas such as information retrieval and data mining. Before joining the PhD program, He had five years of professional experience in the design, development, and testing of software applications. During his PhD program, his internship experiences include Research Center on Software Technology - Italy (Summer 2007), NEC Laboratories - America (Summer 2008), and Microsoft Research - Redmond, USA (Summer 2009). He is a member of Phi Kappa Phi honor society for academic excellence and a recipient of University Outstanding Teaching Assistant award in 2007. He is a student member of ACM.

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Tao Xie, for his invaluable guidance and financial support throughout my Ph.D. studies. His professional advice helped me improve my research skills, technical writing, and effective presentation, and will continue to benefit me in my future career. I thank Mladen Vouk, Laurie Williams, and Yan Solihin for serving on my dissertation committee and providing valuable feedback on my dissertation research. Portions of this dissertation were previously published at ASE 07 [119] (Chapter 3), ASE 08 [120] (Chapter 3), ASE 09 [121] (Chapter 4), ICSE 09 [122] (Chapter 4), ESEC/FSE 09 [125] (Chapter 5), and TAP 10 [118] (Chapter 5). The material included in this dissertation has been updated and extended.

I thank my internship mentors in the industry, who have immensely helped me expand my research to a broader scope: Massimiliano Di Penta (RCOST, Italy), Guofei (Geoff) Jiang, Franjo Ivancic, and Sriram Sankaranarayanan (NEC Laboratories, USA), Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte (Microsoft Research, USA). I thank Massimiliano Di Penta, Guofei Jiang, Nikolai Tillmann, Peli de Halleux, and Tao Xie for writing recommendation letters supporting my summer internship and full-time job applications. I am also grateful to my research collaborators: Hao Zhong, David Lo, Luigi Cerulo and Lerina Aversano. I would like to gratefully acknowledge researchers who generously shared their research tools and results used in my dissertation. I thank Westley Weimer (University of Virginia) for sharing his research results, Shuvendu Lahiri (Microsoft Research) for sharing Randoop tool, Nikolai Tillmann and Peli de Halleux (Microsoft Research) for providing support for the Pex tool. I am very much thankful to all my peers at the Automated Software Engineering group, Tao Xie's research group at NC State, for their constant support, insightful discussions and useful feedback on my research. My research was supported by Army Research Office (ARO) grant W911NF-08-1-0443, and NSF grants CNS-0720641 and CCF-0725190, and ARO grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI). Gratitude to the Department of Computer Science for Teaching Assistantship opportunity during my first semester and financial support during my PhD.

Heartfelt thanks go to Margery Page, Carol Allen, and all the helpful staff at the Department of Computer Science. Special thanks go to my dearest friends Padmashree Ravindra, Madhuri R. Marri, DaYoung Lee, Kunal Taneja, Jeehyun Hwang, and Yoonki Song for making Raleigh a home away home for me. I thank Abhik Sarkar, Gaurav Bawa, and Nikhil Deshpande for sharing the apartment and making my stay at Raleigh eventful. I thank my undergraduate friends who believed and convinced me for pursuing PhD. Finally, last but not the least, I am very thankful to my family. My family Adilakshmi Thummalapenta (mom), Venkata Ramana

Thummalapenta (dad), Srinivasu Thummalapenta (brother), Sailaja Thummalapenta (sister-in-law), Rajyalakshmi Golla (sister), and Subhash Golla (brother-in-law) were my constant source of support and encouragement throughout my journey as a doctoral student. Without their support, I would not have completed this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Motivation

The major goal of software development is to deliver high-quality software efficiently. To achieve this goal of delivering software efficiently, programmers often reuse existing frameworks or libraries, hereby referred to as libraries, instead of developing similar code artifacts from the scratch. These libraries could be proprietary libraries such as the .NET library of C# or open source libraries such as Eclipse [35].

During the past decade, there has been an exponential growth of open source libraries [32]. For example, sourceforge.net, the most popular website for open source software development, hosts about 230,000 projects[1] with two million registered users and a large number of anonymous users. Along with the exponential growth of open source libraries, reuse of these open source libraries has also been increasing rapidly [19, 52]. Recently, Mockus [90] conducted an empirical study to identify large-scale reuse of open source libraries. Their study shows that more than 50% of source files among their projects under analysis include code from other open source libraries. Furthermore, recent trends in reusing libraries also led to a new programming paradigm, called Opportunistic Software Systems Development (OSSD) [93]. In OSSD, programmers develop systems from readily available components by melding the software pieces together. The key insight of OSSD is to advance the state of the art by making most of the existing artifacts, rather than recreating similar artifacts from the scratch [20]. Along with improving software productivity, reuse of existing libraries also helps in reducing effort during software maintenance [33, 91]. For example, libraries that are highly reused tend to have better quality and require less effort in maintenance of applications reusing those libraries.

In this dissertation, we focus on object-oriented libraries, where reuse is one of the major objectives. In general, object-oriented libraries expose their functionality through an interface,

[1] `http://sourceforge.net/,2010`

called Application Programming Interface (API), that programmers can reuse for various tasks. In object-oriented languages, API is an abstract term used to represent a group of classes and methods provided by libraries. We use notations API class and API method to refer to an individual class and method, respectively. To effectively reuse existing libraries, programmers require the knowledge of how to use the APIs exposed by those libraries. However, in practice, programmers often face challenges in understanding the usage of the APIs due to two major factors [26, 115], which are described next.

First, many existing libraries are not well-documented [1, 21]. Even when such documentations exist, they are often outdated [73]. To illustrate the challenges faced by programmers in the absence of documentation, consider that a programmer is reusing the Eclipse library [35]. Consider that the programming task at hand is to write code for parsing code in a dirty editor (an editor whose content is not yet saved) of the Eclipse IDE framework. Since a dirty editor is represented as an object of the `IEditorPart` class and the programmer needs an object of `ICompilationUnit` for parsing, the programmer has to identify a method sequence that accepts the `IEditorPart` object as input and results in an object of `ICompilationUnit`. One such possible method sequence is shown below:

```
IEditorPart iep = ...
IEditorInput editorInp = iep.getEditorInput();
IWorkingCopyManager wcm = JavaUI.getWorkingCopyManager();
ICompilationUnit icu = wcm.getWorkingCopy(editorInp);
```

The preceding code example exhibits the challenges faced by programmers in reusing the existing libraries. A programmer unfamiliar to Eclipse may take long time to identify that an `IWorkingCopyManager` object is needed for getting the `ICompilationUnit` object from an object of the `IEditorInput` class. Furthermore, it is not trivial to find an appropriate way of instantiating the `IWorkingCopyManager` object as the instantiation requires a static method call on the `JavaUI` class.

Second, many existing libraries expose a large number of APIs. For example, .NET library provides nearly 10,000 API classes [94]. These classes provide various functionalities such as common data structures (`Stack`, `LinkedList` etc.,) and reading or writing files. Among those large number of APIs, not all APIs are of the same importance and some APIs are more important compared to other APIs. Lack of such knowledge about libraries often poses additional challenges such as how to start using a library or which API classes of that library to use for achieving the programming task at hand.

1.2 Problems

There are three major problems (described next) that affect both software productivity and quality while reusing existing libraries due to the preceding two factors.

- Programmers spend more effort in understanding APIs, thereby reducing software productivity [103]. In a recent empirical study [103], Robillard shows that lack of documentation or insufficient examples is the major obstacle in understanding APIs. Therefore, for example, if a programmer is not aware of which APIs of a library to use to achieve a programming task at hand, the programmer spends effort in searching for relevant code examples (on the web) that show similar usage of APIs, increasing the amount of time required to finish the programming task at hand.

- Programmers introduce defects while using APIs due to lack of proper knowledge on how to reuse those APIs [24, 38, 75]. The primary reason for such defects is that often APIs require programmers to follow implicit programming rules. Failing to follow such programming rules introduces defects in the *client code* that reuses those APIs. For example, consider the `next` method of the `Iterator` class in the Java Util package [60]. The `next` method throws `NoSuchElementException` when invoked on an empty `collection`. This exception can be avoided through a `boolean` condition check on the `hasNext` method before invoking the `next` method. This relation between the `next` and `hasNext` methods forms an implicit programming rule as "`boolean-check` on the return of `Iterator.hasNext` before `Iterator.next`". However, programmers unfamiliar with this rule may not perform the `boolean` check on the `hasNext` method, thereby introducing defects in the client code. It is quite challenging for static or dynamic verification techniques to detect such defects in the client code, since these techniques require those implicit programming rules, which often do exist in practice [1]. Without such programming rules, existing verification techniques can detect only robustness-related defects such as division by zero or dereferencing a `null` pointer.

- Existing automated white-box test-generation techniques [59, 67, 128, 137] face challenges in effectively generating test inputs for the client code that reuses libraries. Although these challenges apply for general object-oriented code, these challenges are more serious for the client code that reuses libraries. The primary reason is that reuse of existing libraries indirectly increases the amount of code under test, since test-generation techniques require the knowledge of reused libraries as well. For example, consider generation of test inputs for testing the client code reusing APIs from an existing library such as .NET library. In particular, test inputs require method sequences that create and mutate objects and exercise the code under test. These sequences help achieve high structural coverage such as

branch coverage by covering the `true` or `false` branches in the code under test. However, sequences involving existing libraries often include multiple classes. It is quite challenging to automatically generate desirable method sequences with multiple classes due to a large search space of possible method sequences, and the valid sequences constitute only a small portion of the entire search space.

1.3 Solution

To address these preceding issues and thereby improve both software productivity and quality, in this dissertation, we propose a novel general framework, called *WebMiner*. Our WebMiner framework includes new techniques developed from four major research areas: information retrieval, program analysis, data mining, and software testing. The key insight of our WebMiner framework is that the existing API usage knowledge available in the open source code can be effectively leveraged to address these three preceding problems described in Section 1.2. In particular, WebMiner automatically infers usage specifications for APIs under analysis by automatically collecting relevant code examples from the open source code available on the web and by applying data mining techniques on those collected code examples, also referred to as a methodology, called *mining software engineering data*. WebMiner is a general framework and can be easily extended based on the software engineering task (*SE task*) at hand. We next present an overview of the general framework that includes four major phases and describe how the general framework can be extended for achieving an SE task under analysis in subsequent chapters.

First, given an API method under analysis, WebMiner automatically collects relevant code examples of that API among open source code available on the web. These code examples help infer usage specifications of the API under analysis. To collect relevant code examples, WebMiner interacts with code search engines (CSE) such as Google Code Search [48]. CSEs accept queries such as the names of API classes or methods and return relevant code examples from the CVS or SVN repositories of existing open source projects available on the web. The primary reason for leveraging CSEs is that, while applying data mining techniques, along with the mining techniques, the data on which the techniques are applied is also important. In previous work [24, 38, 75], mining techniques are applied on limited data scope, i.e., only a few example applications. Therefore, those approaches may not be able to mine patterns that do not have enough supporting samples in those example applications. To address this issue, in this dissertation, we expand the data scope to a large amount of open source code available on the web via leveraging code search engines.

Second, WebMiner uses a light-weight heuristic-based partial-program analysis for analyzing collected code examples. Our analysis does not require the code examples to be compilable,

since often the code examples collected from CSEs cannot be compiled due to lack of other source files that the current code example is dependent upon. Being light-weight, our analysis is highly scalable and can handle a large number of code examples, as shown in our evaluations.

Third, WebMiner transforms analyzed code examples into pattern candidates suitable for applying data mining techniques such as Frequent Itemset Mining [22]. These mining techniques help identify common patterns, which represent likely usage of APIs, among all patterns candidates. The primary reason for identifying common patterns is based on the observation that majority of the programmers correctly adhere to API usage specifications and those common patterns are likely to represent the correct usage of APIs.

Finally, WebMiner uses mined API usage specifications for increasing software productivity and quality. In particular, WebMiner assists programmers in understanding how to use APIs by suggesting related API usage specifications, thereby improving productivity. Similarly, WebMiner uses API usage specifications to detect violations in the client code as deviations from the mined specifications. These mined API usage specifications can also be used to assist state-of-the-art test-generation approaches in generating test inputs that can achieve high structural coverage such as branch coverage of the applications under analysis using those APIs under analysis.

1.4 Contributions

In summary, this dissertation makes the following major contributions.

- **Data-scope expansion.** A technique for expanding the data scope to the large amount of open source code available on the web. Our technique leverages code search engines to collect relevant code examples of APIs under analysis.

- **Partial-program analysis.** A light-weight heuristic-based analysis for analyzing code examples collected from a code search engine. Our analysis does not require building symbol tables or resolving method calls. Furthermore, it is simple and highly scalable, enabling analysis of a large number of code examples.

- **Approaches for improving productivity.** Two approaches, called *PARSEWeb* [119] and *SpotWeb* [120], that assist programmers in effectively reusing APIs provided by existing libraries. *PARSEWeb* accepts queries of the form "*Source* $\rightarrow$ *Destination*" and mines frequent method sequences that accept an object of the *Source* type and produce an object of the *Destination* type. In our evaluations, we show that *PARSEWeb* addresses queries posted in developer forums and performs better than two related approaches [56, 80]. *SpotWeb* assists programmers in understanding how to start reusing a library. In our

evaluations, we show that the suggestions (in the form of API classes) given by SpotWeb are the same as API classes reused by a real application and show that SpotWeb performs better than a related approach [64].

- **Approaches for improving quality via static verification.** Two approaches, called *CAR-Miner* [122] and *Alattin* [121, 123], that mine API patterns and detect defects in client code as deviations from mined patterns. CAR-Miner focuses on reducing false negatives by detecting new defects, whereas Alattin focuses on reducing false positives among detected violations. CAR-Miner and Alattin also include two new data mining algorithms, respectively, that mine patterns in desired formats required for addressing the SE task under analyses. In our evaluations, we show that CAR-Miner mines 294 real programming rules in five real-world applications (including 285 KLOC) and also detects 160 defects, where 87 defects are new defects that are not found by a related approach [133]. Similarly, in Alattin approach, we show how different new pattern formats such as *Or* or *Xor* patterns help reduce false positives among detected violations. We also suggest best pattern-mining and violation-detection techniques.

- **Approaches for improving quality via dynamic test generation.** Two approaches, called *MSeqGen* [125] and *DyGen* [118], that mine static and dynamic traces, respectively, for effectively generating test inputs that achieve high structural coverage of the code under test. MSeqGen statically extracts method sequences from existing code bases and assists random [97] and dynamic-symbolic-execution [126] based approaches. In our evaluations, we show that with the assistance of *MSeqGen*, a random approach [97] achieves 8.7% (with a maximum of 20.0% for one namespace) higher branch coverage and a dynamic-symbolic-execution-based approach achieves 17.4% (with a maximum of 22.5% for one namespace) higher branch coverage than without the assistance of *MSeqGen*. In contrast to *MSeqGen*, DyGen automatically generates regression tests from dynamic traces. In our evaluations, we show that *DyGen* recorded ≈1.5 GB C# source code (including 433,809 traces) of dynamic traces from applications using two core libraries of the .NET library. DyGen eventually generated 501,799 regression tests, where each test exercises a unique path. This dissertation also includes an additional approach, called *Seeker*, that addresses a major issue with mining-based approaches, which are not effective in the scenarios where usage information is not available for the API methods under analysis or usage information is not sufficient to use for achieving high structural coverage. In contrast to mining-based approaches, *Seeker* generates test inputs based on implementation information. In our evaluations, we compare *MSeqGen* and *Seeker* and discuss their benefits and limitations, respectively.

1.5 Scope

The approaches presented in this dissertation focus on two major research areas: mining software engineering data (MSED) and software testing. In the MSED research area, unlike a few existing approaches [7, 55, 134] that focus on API implementation information, our approaches [119–122] primarily focus on API usage information, complementing those approaches. Furthermore, unlike existing approaches [24,38,75,133] that focus on a few example code bases, our approaches leverage the large amount of open source code available on the web. On the other hand, in the software testing research area, unlike existing approaches [49,70,97,126,129] that primarily focus on API implementation information, our approaches [118, 124, 125] focus on both API usage and implementation information.

Although a majority of our approaches focus on open source code available on the web, our approaches are general and can be applied on proprietary code bases as well, as shown in the evaluation results of our DyGen approach [118]. Finally, all our approaches focus on source code. However, there exist various other SE artifacts such as bug reports or programmer forums on the web that can be leveraged for addressing various other problems in software engineering. Chapter 6 discusses our new approaches that target at leveraging those other SE artifacts beyond the source code.

1.6 Outline

The remainder of this dissertation is organized as follows. Chapter 2 presents key concepts of our general WebMiner framework. The subsequent chapters present six approaches developed based on our general framework to address different SE tasks, respectively. In particular, Chapter 3 presents two approaches that assist programmers in reusing APIs provided by existing libraries. Chapter 4 presents two approaches that use mined API usage specifications as programming rules and detect defects in applications under analysis as deviations from the mined specifications. Chapter 5 presents two approaches that mine static and dynamic traces, respectively, for effectively generating test inputs that achieve high structural coverage of the code under test. Chapter 5 also presents another approach that addresses a major issue with mining-based approaches, which are not effective in scenarios where usage information is not available for the API methods under analysis or usage information is not sufficient to achieve the SE task under analysis. Chapter 6 discusses our future work. Finally, Chapter 7 concludes with a summary of our contributions and lessons learned.

Chapter 2

WebMiner Framework

2.1 Introduction

We next describe our WebMiner framework that leverages the knowledge available in open source code available on the web. The key insight of our WebMiner framework is that the large amount of open source code available on the web includes API usage knowledge that can be effectively leveraged for improving both software productivity and quality. However, a major issue with handling open source code is that the amount of existing open source code is quite large and complete analysis is not possible. To address this issue, WebMiner leverages code search engines such as Google code search [48], Krugle [71], Koders [69], and Codase [27] to collect relevant code examples for APIs under analysis and analyzes those relevant code examples. Although we focus on open source code, our techniques are general and can be applicable to large proprietary code bases such as Microsoft code bases. We next present the four major phases of our WebMiner framework.

Figure 2.1 shows the four major phases of our WebMiner framework for achieving an SE task such as detecting defects in API client code via mining API usage specifications: *search*, *analyze*, *mine*, and *apply* phases. Given an SE task, the *search* phase collects SE data in the form of code examples by leveraging a code search engine such as Google code search [48]. These collected code examples include necessary information for mining API usage specifications. The *analyze* phase analyzes collected code examples to extract relevant SE data. A major issue with collected code examples is that they are often partial and not compilable. In our context, a partial code example indicates that the code example is complete; however, the other source files on which the code example is dependent upon are not available. To address this issue, the *analyze* phase includes partial-program analysis for resolving object types based on heuristics. The *analyze* phase next transforms extracted SE data into pattern candidates suitable for applying mining algorithms.

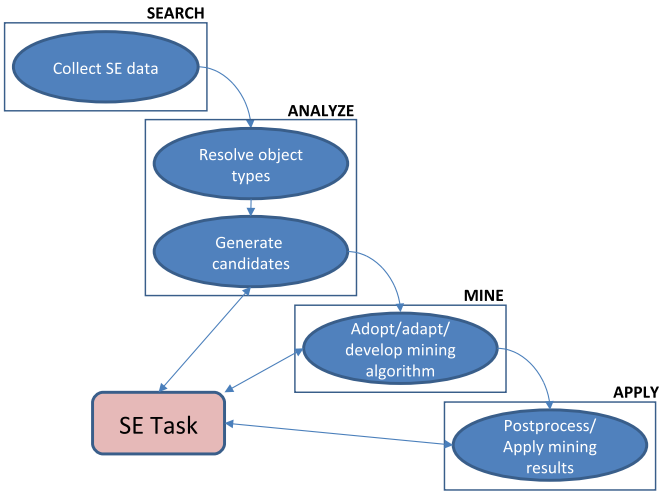


Figure 2.1: Four major phases of our WebMiner framework.

The *mine* phase applies data mining algorithms to identify frequent patterns among all pattern candidates. The key insight behind the *mine* phase is that majority of the programmers correctly adhere to API specifications and those frequent patterns are likely to represent the correct usage of APIs. Based on the SE task under analysis, the mine phase either uses an existing off-the-shelf mining algorithm such as frequent itemset mining [22] or a new mining algorithm developed based on the requirements unique to the SE task under analysis.

The *apply* phase transforms frequent patterns identified by the *mine* phase into a format required to achieve the SE task. The apply phase next uses mining results for achieving the SE task. For example, the apply phase detects defects in API client code by using a static-verification tool using mined API specifications. Among these four phases, the search and analyze phases are general and the remaining two phases need to be customized based on the SE task under analysis. We next explain each phase in detail.

2.2 Search Phase

Given an SE task, the *search* phase collects SE data in the form of code examples by leveraging a code search engine (CSE) such as Google code search (GCSE) [48]. The primary reason for the introduction of CSEs is that the normal search engines such as Yahoo (`www.yahoo.com`) and Google (`www.google.com`) mainly search based on the textual content of a file. However, a file including source code is more than just a textual file. For example, each word in the source code file has a different meaning depending on several factors such as the programming language or the location of that word in the source code file. Therefore, to effectively search in available open source code, CSEs index source code based on the semantics of the corresponding programming language. We next explain how the search phase collects code examples via CSE

and present additional techniques that help address the challenges faced while collecting those code examples.

To search and collect SE data such as code examples relevant to APIs whose specifications need to be mined, the search phase constructs queries to CSEs. We consider that a code example is relevant to an API method, if the code example includes call sites of that API method. These constructed queries include key words derived based on the names of API methods. CSEs also provide additional options such as the language of the API class or method for further filtering out the code examples during the search. For example, the search phase constructs the query "`lang:java org.apache.regexp RE`" to collect relevant code examples of the `RE` class provided by the Apache library [11] via GCSE. In the preceding query, the option "`lang:java`" describes that the language under consideration is Java. GCSE returns around $2,000$ code examples for this query. These code examples include information that helps in mining API usage specifications for the `RE` class.

A major issue with the code examples collected from CSEs is that these code examples often include duplicates. We consider a code example ce_i as a duplicate of another code example ce_j, if both ce_i and ce_j belong to the same project and the same source file in that project. For example, among $2,000$ code examples returned by GCSE for the query "`lang:java org.apache.regexp RE`", the source file `JakartaRegexpRegexp.java` is found 13 times. Among these 13 copies, there are 5 different versions of the source file and the remaining 8 copies are duplicates of these 5 versions. There are both desirable and undesirable consequences with duplicates or multiple versions of source files among code examples. For example, code examples that are duplicate of the same source file, such as those belonging to a particular jar file, can be found to be used in various projects. The existence of duplicate or multiple copies for a code example can indicate that the code example is widely used and therefore the code example can be trusted more than those code examples that do not have duplicate or multiple versions. On the other hand, duplicate or multiple copies can bias the results of mining approaches that try to mine common patterns. To mine unbiased patterns used across a large number of code bases, in our WebMiner framework, we identify and filter out duplicate code examples.

In achieving various SE tasks described in subsequent chapters, we used GCSE [48] for collecting relevant code examples for the APIs under analysis. The primary reason is that GCSE provides convenient open APIs for third-party tools to interact with and has been consistently improved and maintained. However, our WebMiner framework is independent of the underlying CSE and can be extended easily to any other CSE.

2.3 Analyze Phase

We next explain how the analyze phase analyzes SE data collected in the form of code examples. The analyze phase includes two major tasks: resolve object types and generate candidates. Since these code examples are collected from a CSE, traditional program analysis techniques cannot be used for analyzing these code examples. The primary reason is that CSEs often return only individual source files (i.e., code examples) that include the search term, and these code examples are often partial and not compilable. In our context, a partial code example indicates that the code example is complete; however, the other source files on which the code example is dependent upon are not available. Therefore, to analyze these code examples, we propose partial-program analysis for resolving object types based on heuristics. These heuristics are contrary to type checking done by a compiler. In particular, our heuristics are based on simple language semantics like object types of left and right hand expressions of an assignment statement are either the same or related to each other through inheritance.

A primary advantage of our partial-program analysis is that it does not require building symbol tables or resolving method calls to their declaring method bodies. Furthermore, it is simple and highly scalable, enabling analysis of a large number of code examples. Our heuristics are not complete, since our heuristics cannot resolve the entire type information. However, the evaluation results of our approaches [119–123] developed based on our WebMiner framework show that these heuristics are often effective in resolving required object type information. We first present the heuristics used by our partial-program analysis for resolving object types and next describe how we generate pattern candidates from collected code examples.

2.3.1 Resolve object types

We use 16 heuristics for resolving object types such as receiver or argument object types of an API method call in collected code examples. We next explain two of our major heuristics used for identifying the return type of an API method call.

Heuristic 1: *The return type of an API method call contained in an initialization expression is the same as the type or a subtype of the declared variable.*

Consider the code example shown below:

```
QueueConnection connect;
QueueSession session = connect.createQueueSession(false,int)
```

The receiver type of the method `createQueueSession` is the type of `connect` variable. Therefore, the receiver type can be simply inferred by looking at the declaration of the `connect` variable. However, since our framework mainly deals with code that is partial and not compilable, it is difficult to get the return type of the method `createQueueSession`. The reason is the

lack of access to method declarations. However, the return type can be inferred from the type of variable `session` on the left hand side of the assignment statement. Since the type of variable `session` is `QueueSession`, we can infer that the return type of the method `createQueueSession` is `QueueSession` or its subtype.

Heuristic 2: *The return type of an outermost API method call contained in a return statement is the same (or a subtype) as the return type of the enclosing method declaration.*

Consider the code example shown below:

```
public QueueSession test()   { ...
      return connect.createQueueSession (false,int);
}
```

In the preceding code example, the `createQueueSession` method is a part of the return statement of the method declaration. In this scenario, we can infer the return type of this method from the return type of the method `test`. Since the method `test` returns `QueueSession`, we can infer that the return type of the `createQueueSession` method is also `QueueSession` or its subtype. We next present a scenario, where our partial-program analysis cannot identify necessary object types due to lack of additional information.

Consider the code example shown below:

```
QueueConnectionFactory factory = jndiContext.lookup("t");
QueueSession session = factory.createQueueConnection()
            .createQueueSession(false, Session.AUTOACKNOWLEDGE);
```

In the second statement of the preceding code example, our heuristics cannot infer the receiver type of the `createQueueSession` method and the return type of the `createQueueConnection` method. The reason is due to lack of information regarding the intermediate object returned by the `createQueueConnection` method. However, we identify that such scenarios happen rarely based on our empirical evidence shown in our evaluation results [119–123].

2.3.2 Generate candidates

After type resolution based on preceding heuristics, the *analyze* phase generates pattern candidates suitable for applying mining algorithms in the *mine* phase. In general, these pattern candidates are based on the SE task under analysis. For example, consider the SE task as suggesting method sequences to programmers for achieving a particular programming task at hand (recall the code example shown in Section 1). For this SE task, the pattern candidates are in the form of sequences of method calls.

Initially, the *analyze* phase constructs control-flow graphs (CFG) for each code example collected from the code search engine. The *analyze* phase next generates pattern candidates by

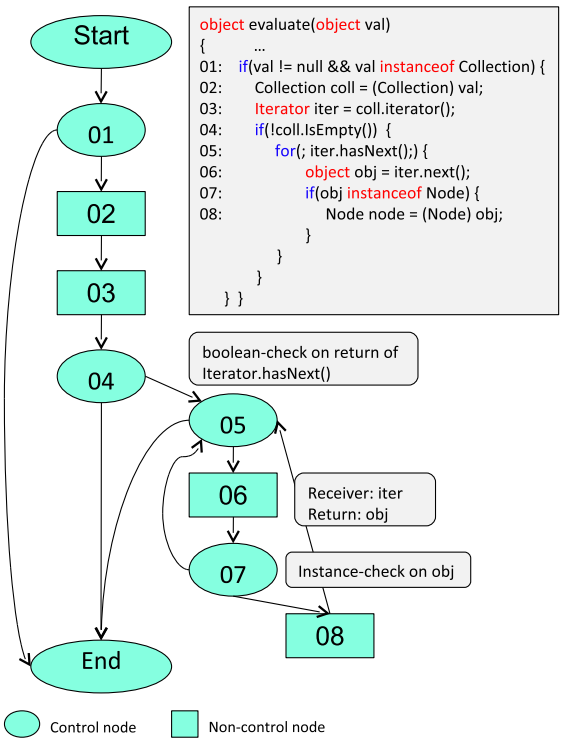


Figure 2.2: A sample code example collected from Google code search and its constructed CFG.

traversing constructed CFGs. The traversal of CFG is primarily based on the SE task under analysis. We next explain how the *analyze* phase constructs CFGs using the illustrative example shown in Figure 2.2. The figure also shows the constructed CFG for the code example. First, the *analyze* phase creates an Abstract Syntax Tree (AST) for each code example. the *analyze* phase uses the created AST to build a CFG. A node in the constructed CFG contains a single statement and an edge represents the control flow between the two statements. Our constructed CFGs include two kinds of nodes: control (CT) and non-control (NT) nodes. Control nodes represent control-flow statements such as *if*, *while*, and *for*, which control the flow of the program execution. Non-control nodes represent other statements such as method calls or type casts. For example, Statement 5 in the code example (Figure 2.2) is a control node and Statement 6 is a non-control node. When encountering a control node, say CT_i (i indicates the statement id), our WebMiner framework also extracts all variables, say $\{V_1, V_2, ..., V_n\}$, that participate in the conditional expression of that node and the condition checks on those variables. For example, Control node CT_1 includes the {(`val`, `null-check`), (`val`, `instance-check`)} pairs. Here, "(`val`, `null-check`)" indicates that there is a `null-check` on the variable `var` in Control node CT_1. If the control node includes comparisons with expressions such as method calls, the *analyze* phase stores those method calls also as additional information within the control node.

In our constructed CFGs, non-control nodes include three kinds of statements. The primary reason is that only these three kinds of statements result in the transformation of one object

type to another and are required for achieving SE tasks such as suggesting method sequences to programmers.

- Method call: Generally, a method call with a `non-void` return type can be considered as a statement that transforms either the receiver type or argument types to the return type. For example, the method call `ReturnObj obj1 = RefObj.method(Arg1, Arg2)` can be considered as a statement that transforms objects of type `RefObj`, `Arg1` or `Arg2` to `ReturnObj`. However, we consider method calls with `void` return types also, since those method calls can modify the state of its receiver or arguments.
- Constructor: Since a constructor generally takes some arguments, it can be considered as a statement that transforms objects of its argument types to the newly created object type.
- Typecast: A typecast can be considered as a transformation statement, because it performs an explicit transformation from one object type to another.

While constructing CFGs, the *analyze* phase performs method inlining by replacing the method calls of the class under analysis with the body of the corresponding method declarations. The *analyze* phase cannot perform method inlining if the corresponding method declaration is *abstract*. This method inlining process helps identify pattern candidates that are split among methods of the class under analysis. The *analyze* phase next transforms extracted SE data into pattern candidates suitable for applying mining algorithms. For example, for the SE task of suggesting method sequences to programmers, the *analyze* phase replaces each distinct method call among pattern candidates (in the form of method sequences) with a unique id. The primary reason is that data mining algorithms used in the *mine* phase cannot understand data such as method sequences and handles pattern candidates such as sequence of integers.

2.4 Mine Phase

We next describe the *mine* phase of our WebMiner framework. The *mine* phase applies data mining algorithms to identify frequent patterns among all pattern candidates. In our WebMiner framework, the mining algorithm used in the *mine* phase is specific to the SE task under analysis. The primary reason is that different SE tasks require different pattern types. For example, if the SE task is to suggest method sequences, the preferred mining algorithm is frequent subsequence mining algorithm [130], since this algorithm identifies subsequences among pattern candidates in the form of sequences.

As shown in our evaluations [121–123] in subsequent chapters, we identify that existing approaches [24, 38, 75, 112, 131, 133] often produce a large number of false negatives and false positives. Here, false negatives represent the defects that exist in applications under analysis

and are not detected by those approaches. On the other hand, false positives indicate those violations that do not represent real defects. The primary reason for such false negatives and false positives is that these approaches attempt to directly reuse existing off-the-shelf mining algorithms such as a frequent itemset miner [22]. Such direct reuse of existing off-the-shelf mining algorithms often generates patterns that do not describe the *nearly complete* behavior among data points used as inputs to mining algorithms. Therefore, using such patterns for SE tasks such as detecting defects in applications under analysis results in both false negatives and false positives among detected violations. To address these issues, we adapt or develop new mining algorithms for the approaches (presented in the subsequent chapters) developed based on our WebMiner framework.

2.5 Apply Phase

We next describe the last phase, the *apply* phase, of our WebMiner framework. The *apply* phase transforms frequent patterns identified by the *mine* phase into a format required to achieve the SE task under analysis. The *apply* phase next uses mining results for achieving the SE task. For example, for the SE task of suggesting method sequences, the *apply* phase transforms the frequent patterns in the form of integers to method sequences suitable for suggesting to programmers.

2.6 Chapter Summary

In this chapter, we presented a framework, called *WebMiner*, that leverages the knowledge available in open source code available on the web for improving software productivity and quality. Our WebMiner framework includes four major phases: *search*, *analyze*, *mine*, and *apply*. Among these four phases, the *search* and *analyze* phases are general, and the *mine* and *apply* phases need to be customized based on the requirements of SE task under analysis.

Chapter 3

Improving Software Productivity

3.1 Introduction

Programmers often spend more efforts in reusing existing libraries due to various factors such as their complexity or lack of documentation. Therefore, to assist programmers in effectively reusing existing libraries, and thereby to increase programmer's productivity, we developed two approaches, called *PARSEWeb* [119] and *SpotWeb* [120], based on our general WebMiner framework. These two approaches address two specific problems, respectively, in reusing libraries. However, our WebMiner framework is general and can be easily extended to address other problems as well.

In particular, *PARSEWeb* accepts queries of the form "*Source* $\rightarrow$ *Destination*" and mines frequent method sequences that accept an object of the *Source* type and produce an object of the *Destination* type. These method sequences can help programmers while they are writing code using existing libraries. In contrast to *PARSEWeb*, *SpotWeb* assists programmers in reusing API classes and methods of an existing library by detecting hotspots and coldspots of the library. Hotspots are API classes and methods that are frequently reused and can serve as starting points for programmers in understanding and reusing the library. On the other hand, coldspots are API classes and methods that are rarely used. Coldspots serve as caveats for programmers, since there can be difficulties in finding relevant code examples and are generally less exercised compared to hotspots. We next explain each approach in detail.

3.2 PARSEWeb: Assisting programmers in writing method sequences

3.2.1 Motivation

In general, reuse of existing libraries involve instantiation of several object types of those libraries. For example, consider the programming task of parsing code in a dirty editor (an editor whose content is not yet saved) of the Eclipse IDE framework. Since a dirty editor is represented as an object of the `IEditorPart` type and the programmer needs an object of `ICompilationUnit` for parsing, the programmer has to identify a method sequence that takes the `IEditorPart` object as input and results in an object of `ICompilationUnit`. One such possible method sequence is shown below:

```
IEditorPart iep = ...
IEditorInput editorInp = iep.getEditorInput();
IWorkingCopyManager wcm = JavaUI.getWorkingCopyManager();
ICompilationUnit icu = wcm.getWorkingCopy(editorInp);
```

The code example shown above exhibits the difficulties faced by programmers in reusing existing libraries. A programmer unfamiliar to Eclipse may take long time to identify that an `IWorkingCopyManager` object is needed for getting the `ICompilationUnit` object from an object of the `IEditorInput` type. Furthermore, it is not trivial to find an appropriate way of instantiating the `IWorkingCopyManager` object as the instantiation requires a static method call on the `JavaUI` class.

In many such situations, programmers know what type of object that they need to instantiate (like `ICompilationUnit`), but do not know how to write code to get that object from a known object type (like `IEditorPart`). For simplicity, we refer the known object type as *Source* and the required object type as *Destination*. Therefore, the proposed problem can be translated to a query of the form "*Source* $\rightarrow$ *Destination*".

There exist approaches [56, 80, 108] that address the preceding problem. However, the common issue faced by these existing approaches is that the scope of these approaches is limited to the knowledge available in a fixed (often small) set of applications reusing the libraries of interest. On the other hand, CSEs cannot be directly used to address the preceding problem. For example, programmers can issue the query "`IEditorPart ICompilationUnit`" to gather relevant code examples with usages of the object types `IEditorPart` and `ICompilationUnit`. However, CSEs often return a large number of code examples and programmers have to manually browse through those examples to find the relevant code example. For example, GCSE returns nearly 100 results for this query and the desired method sequence shown above is present in the 25^{th} source file among those results.

In this chapter, we propose an approach, called PARSEWeb (based on our WebMiner framework), that addresses the preceding problem by accepting queries of the form "*Source* → *Destination*" and suggests frequently used method sequences (MIS) that can transform an object of the *Source* type to an object of the *Destination* type. Our approach also suggests relevant code examples that are extracted from a large number of publicly accessible source code repositories. These suggested MISs along with the code examples can help programmers in addressing the preceding problem and thereby help reduce programmers' effort in reusing existing libraries.

In particular, PARSEWeb interacts with GCSE [48] to search for code examples with the usages of the given *Source* and *Destination* object types, and downloads those examples to form a local source code repository. PARSEWeb analyzes the local source code repository to extract different MISs and clusters similar MISs (Section 3.2.3). These extracted MISs can serve as a solution for the given query. PARSEWeb also sorts the final set of MISs using several ranking heuristics. PARSEWeb uses an additional phase called *query splitter* that helps address the problem where code examples for the given query are split among different source files.

PARSEWeb makes the following major contributions:

- An approach for reducing programmers' effort while reusing existing libraries by providing frequently used MISs and relevant code examples.
- A technique for clustering similar MISs and two ranking heuristics for sorting the final set of MISs.
- A technique, called query splitter, that splits a query into multiple sub-queries for addressing the issue of relevant code examples being split among different source files or different projects.
- An Eclipse plugin tool implemented for the proposed approach and several evaluations to assess the effectiveness of the tool.

3.2.2 Example

We next use an example to illustrate PARSEWeb and how PARSEWeb can help in reducing programmers' effort when reusing existing libraries. We use object types `QueueConnectionFactory` and `QueueSender` from the OpenJMS library[1], which is an open source implementation of Sun's Java Message Service API 1.1 Specification, as illustrative examples. Consider that a programmer has an object of type `QueueConnectionFactory` and does not know how to write code to get a `QueueSender` object, which is required for sending messages.

To address this problem, the programmer can use our PARSEWeb approach as follows. The programmer first translates the problem into a "`QueueConnectionFactory` → `QueueSender`"

[1] `http://java.sun.com/products/jms`

```
01:FileName:0_UserBean.java MethodName:ingest  Rank:1 NumberOfOccurrences:6
02:QueueConnectionFactory,createQueueConnection() ReturnType:QueueConnection
03:QueueConnection,createQueueSession(boolean,Session.AUTO
                                _ACKNOWLEDGE) ReturnType:QueueSession
04:QueueSession,createSender(Queue) ReturnType:QueueSender
```

Figure 3.1: Method sequence suggested by PARSEWeb.

```
01:QueueConnectionFactory qcf;
02:QueueConnection queueConn = qcf.createQueueConnection();
03:QueueSession qs = queueConn.createQueueSession(true,Session.AUTO
                                                        _ACKNOWLEDGE);
04:QueueSender queueSender = qs.createSender(new Queue());
```

Figure 3.2: Equivalent Java code for the method sequence suggested by PARSEWeb.

query. Given this query, PARSEWeb interacts with GCSE for relevant code examples with usages of the given *Source* and *Destination* object types (*Search* phase of our WebMiner framework). PARSEWeb next downloads the code examples to form a local source code repository. The downloaded code examples are often partial and not compilable, since GCSE retrieves (and subsequently PARSEWeb downloads) only source files with usages of the given object types instead of entire projects. PARSEWeb analyzes each partial code example using an Abstract Syntax Tree (AST) and builds a CFG that represents each given code example in order to capture control-flow information in the code example (*Analyze* phase of our WebMiner framework). PARSEWeb traverses this CFG to extract MISs that take `QueueConnectionFactory` as input and result in an object of `QueueSender`. The output of PARSEWeb for the given query is shown in Figure 3.1. The sequence starts with the invocation of the `createQueueConnection` method that results in an instance of the `QueueConnection` type. Similarly, by following other methods, the method sequence finally results in the `QueueSender` object, which is the desired destination object of the given query.

The example output also shows additional details such as `FileName`, `MethodName`, `Rank`, and `NumberOfOccurrences`. The details `FileName` and `MethodName` indicate the source file and method name, respectively, that the programmer can browse to find a relevant code example for this MIS. For example, a code example for the given query can be found in the method `ingest` of the file `0_UserBean.java`. The prefix of the file name gives the index of the source file that contained the suggested solution among the results of GCSE. In this example, the code example for the suggested method sequence can be found in the first source file returned by GCSE as the prefix value is zero. Generally, many queries result in more than one possible solution. The `Rank` attribute gives the rank of the corresponding MIS among the complete set of results. PARSEWeb derives the rank of a MIS based on the `NumberOfOccurrences` attribute and some other heuristics described in Section 3.2.3. The suggested MIS contains all necessary information for the programmer to write code for getting the *Destination* object from the given *Source* object. The suggested MIS can be transformed to equivalent Java code by introducing

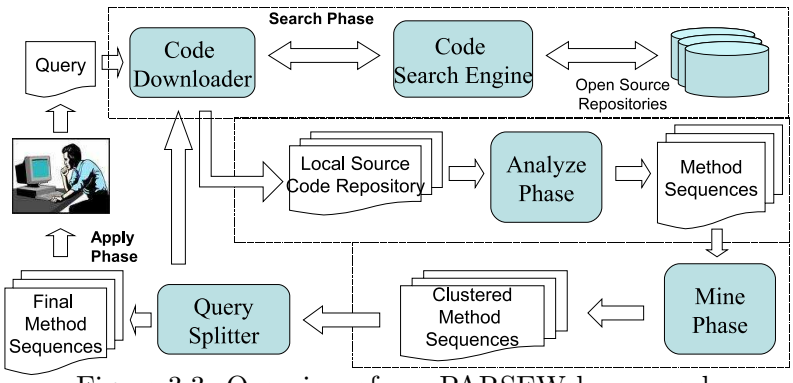


Figure 3.3: Overview of our PARSEWeb approach

required intermediate variables. The code example suggested along with the MIS can assist programmers in gathering this additional information regarding the intermediate variables. Figure 3.2 shows equivalent Java code for the suggested MIS.

3.2.3 Approach

PARSEWeb, developed based on our WebMiner framework, includes an additional phase, called *query splitter*, along with the four major phases of the WebMiner framework: *search*, *analyze*, *mine*, and *apply*. Figure 3.3 shows an overview of all phases of PARSEWeb. Unlike WebMiner, in PARSEWeb, the three phases *search*, *analyze*, and *mine* may be iterated more than once. In particular, in the *search* phase, the code downloader component accepts a query from the programmer and forms a local source code repository with the code examples collected through CSE. The *analyze* phase analyzes the code examples stored in the repository and generates MISs that serve as pattern candidates for the next phase. The *mine* phase clusters similar MISs and ranks the clustered MISs. If the result of the *mine* phase consists of any MISs that can serve as a solution for the given query, *query splitter* simply outputs the result. If there are no solution MISs in the result generated by the *mine* phase, *query splitter* instead splits the given query into different sub-queries and iterates all the preceding three phases for each sub-query. Finally, *query splitter* gathers results of all sub-queries and generates the final output. In PARSEWeb, the *apply* phase simply includes suggesting final output to the programmer. We next present more details about all phases except the *search* phase, since the *search* phase is the same as our WebMiner framework.

Analyze Phase

In the *analyze* phase, PARSEWeb first resolves object types using partial-program analysis and constructs CFGs for each code example (as described in Section 2.3 of our WebMiner framework). PARSEWeb next identifies *Source* and *Destination* nodes in constructed CFG. In particular, when any of receiver, return, or argument types of a non-control node in CFG

matches with the given *Source* object type, PARSEWeb marks the corresponding node as a *Source* node. On the other hand, when the return type of any non-control node in CFG matches with the required *Destination* type, PARSEWeb marks the corresponding node as a *Destination* node.

PARSEWeb extracts a MIS from constructed CFG by calculating the shortest path from a *Source* node to a *Destination* node. The shortest path is sufficient as every path from *Source* to *Destination* nodes contain a desired method sequence. Once a possible sequence is identified from the CFG, the minimization process of PARSEWeb extracts a minimal MIS from the possible sequence by eliminating the extra methods that are not related to the given query. This minimal MIS is identified by traversing the sequence in the reverse direction from the *Destination* node to the *Source* node by continuously matching the receiver type and argument types of each statement with the return type of the preceding statements. For example, consider a possible sequence for the query "`IEditorPart` → `ICompilationUnit`" (where each statement consists of the receiver type, method name, arguments, and return type) as follows:

```
01:IEditorPart,getEditorInput() : IEditorInput
02:CONSTRUCTOR,Shell() : Shell
03:Shell,setText(String) : void
04:JavaUI,getWorkingCopyManager() : IWorkingCopyManager
05:IWorkingCopyManager,connect(IEditorInput) : void
06:IWorkingCopyManager,getWorkingCopy(IEditorInput) : ICompilationUnit
```

The minimization process maintains a special set called a look-up set that initially contains only the required *Destination* object. For the given possible sequence, the process starts from Statement 6 and adds the receiver type `IWorkingCopyManager` and the argument type `IEditorInput` to the look-up set, and removes `ICompilationUnit` from the look-up set. The minimization process retains Statement 5 in the minimal MIS as its receiver type matches with one of the types in the look-up set. In Statement 4, the minimization process adds `JavaUI` to the look-up set and removes `IWorkingCopyManager`. The process ignores Statements 3 and 2 as none of its object types match with the object types in the look-up set. The minimization process ends with Statement 1 and generates the minimal MIS as "1,4,5,6". These minimal MISs serve as pattern candidates for the *mine* phase.

Mine Phase

In the *mine* phase, PARSEWeb clusters similar MISs and ranks the clustered MISs. Clustering of MISs helps to identify distinct possible MISs and also reduces the total number of MISs. This reduction of the number of results can help programmers quickly identify the desired MIS for the given query. To further assist programmers, PARSEWeb sorts the clustered results.

These sorted results can help programmers identify sequences that are more frequently used for addressing the given query.

Sequence Clustering. We next describe the heuristic used by PARSEWeb for identifying and clustering similar MISs. To identify similar MISs, PARSEWeb ignores the order of statements in the extracted MISs. PARSEWeb considers MISs with the same set of statements and with a different order as similar. For example, consider MISs "2,3,4,5" and "2,4,3,5" where each number indicates a single statement associated with the node in the constructed CFG. PARSEWeb considers these sequences as similar because different programmers may write intermediate statements in different orders and these statements may be independent from one another.

To further cluster the identified MISs, PARSEWeb identifies MISs with minor differences and clusters those identified MISs. We introduce an attribute, called *cluster precision*, which defines the number of statements by which two given MISs differ each other. This attribute is configurable and helps PARSEWeb in further clustering the identified MISs. PARSEWeb considers MISs that differ by the given *cluster precision* value as similar, irrespective of the order of statements in those MISs. For example, consider MISs "8,9,6,7" and "8,6,10,7". These two sequences have three common statements (8,6,7) and differ by a single statement. PARSEWeb considers these two MISs as similar under a *cluster precision value* of one, as both the sequences differ by only one method. This heuristic is based on the observation that different MISs in the final set of sequences often contain overloaded forms of the same method.

Sequence Ranking. In general, many queries result in more than one possible solution, and not all solutions are of the same importance to the programmer. To assist the programmer in quickly identifying the desired MISs, PARSEWeb uses two ranking heuristics and sorts the final set of MISs.

Ranking Heuristic 3.2.1 Frequency. Higher the frequency ⇒ Higher the rank

This heuristic is based on the observation that more-used MISs might be more likely to be used compared to less-used MISs. Therefore, MISs with higher frequencies are given higher preference compared to MISs with lower frequencies.

Ranking Heuristic 3.2.2 Length. Shorter the length ⇒ Higher the rank

This ranking heuristic, which was originally proposed in the Prospector [80] approach, is based on the length of the MIS. Shorter sequences are given higher preference to longer sequences. This heuristic is considered based on the observation that programmers would often tend to use shorter sequences instead of longer ones to achieve their task.

Algorithm 1 Pseudocode of the PARSEWeb algorithm with the query splitter phase

Require: Source and Destination object types.
Ensure: Method Sequences

```
1: Extract MISs for the Query "Source → Destination"
2: if MISs are not empty then
3:     return MIS
4: end if
5: //Query Splitting
6: Extract DestOnlyMISs for the Query "Destination"
7: for all MIS in DestOnlyMISs do
8:     lastMI = MIS.lastMethodInvocation()
9:     AltDestSet = ReceiverType and ArgTypes of lastMI
10:    Intialize FinalMISs
11:    for all AltDest in AltDestSet do
12:        Extract AltMIS for the Query "Source → AltDest"
13:        Append lastMI to AltMIS
14:        Add AltMIS to FinalMISs
15:    end for
16: end for
17: if FinalMISs are not empty then
18:    return FinalMISs
19: else
20:    return DestOnlyMISs
21: end if
22: return GlobOSS
```

Query Splitter

Query splitter is an additional phase used by PARSEWeb to address the problem of lack of code examples in the results of CSE. We observed that a code example for some of the queries is split among different files instead of having the entire example in the same file. This phase helps to address this problem by splitting the given query into multiple sub-queries. The algorithm of PARSEWeb including the query splitter phase is described in Algorithm 1.

Initially, PARSEWeb accepts the query of the form "*Source* → *Destination*" and tries to suggest solutions. If no possible MISs are found, PARSEWeb tries to infer the immediate alternate destinations (*AltDest*) by constructing a new query that includes only the *Destination* object type. A query with just the *Destination* object type provides different possible MISs, referred to as *DestOnlyMISs*, that result in the object of the *Destination* type. In these *DestOnlyMISs*, the *Source* can be of any object type. PARSEWeb infers the *AltDestSet* by identifying the receiver type and argument types in the last method call (*lastMI*) of each MIS in the *DestOnlyMISs* set. The primitive types, such as `int`, are ignored while identifying the *AltDest*.

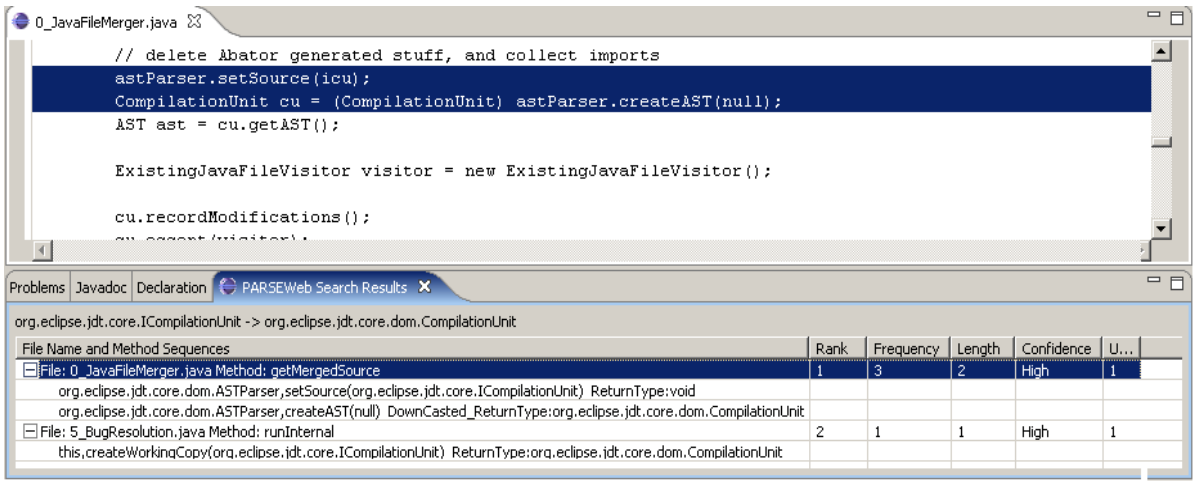


Figure 3.4: A snapshot of PARSEWeb plugin interface.

For each of the *AltDest*, new MISs are generated by constructing queries of the form "*Source* → *AltDest*". The *lastMI* of the earlier sequence is appended to the new set of sequences to generate a complete MIS. In case if PARSEWeb including the query splitter phase is not able to suggest any MISs, PARSEWeb simply returns the *DestOnlyMISs*, hoping that these MISs can provide hints to programmers in constructing new queries.

Implementation

We developed an Eclipse plugin[2] for our PARSEWeb approach. We used Google Code Search Engine (GCSE) [48] as an underlying CSE. To improve performance, PARSEWeb uses the multi-threading feature of the Java programming language, and invokes a post processor written in the Perl language to transform the source files returned by GCSE from HTML to Java. Eclipse JDT Compiler [10] is used for building ASTs from Java files. We used Dijkstra's shortest path algorithm from the Jung [65] library to gather the required path from *Source* to *Destination* object types.

PARSEWeb displays the suggested MISs for the given query in a tree-structured tabular form. A snapshot of our PARSEWeb output is shown in Figure 3.4. Each MIS is associated with additional information like rank, frequency, and length. Programmers can browse the relevant code sample of the suggested MIS by double clicking on the corresponding entry.

The current implementation of PARSEWeb shows only the first ten MISs that can serve as a solution for the given query. Furthermore, the query splitter is configured to iterate all three main phases for only the first five elements in *DestOnlyMISs* (shown in Algorithm 1). However, both these parameters are configurable through a property file.

[2]Available at `http://research.csc.ncsu.edu/ase/projects/parseweb/`

3.2.4 Evaluation

We conducted four different evaluations to show that PARSEWeb is effective in solving programmers' queries. In the first evaluation, we showed that PARSEWeb is able to solve programming problems posted in developer forums of existing libraries. In the second evaluation, we showed that PARSEWeb-suggested solutions are available in a real project. We also compared our results with the results of two related approaches: Prospector[3] [80] and Strathcona[4] [56]. Prospector tries to solve the queries related to a specific set of libraries by using API signatures. Strathcona tries to suggest similar code examples stored in an example repository by matching the context of the code under development with the examples stored in the example repository. In the third evaluation, we compared PARSEWeb with Prospector. We showed that PARSEWeb performs better than Prospector. Moreover, PARSEWeb is not limited to the queries of any specific set of libraries as Prospector is. In the fourth evaluation, we showed the significance of different techniques used in PARSEWeb.

Programming Problems

The purpose of this evaluation is to show that PARSEWeb is not limited to the queries of any specific set of libraries. We collected two programming problems that were posted in developer forums of existing open source libraries and checked whether PARSEWeb is able to suggest solutions for those problems. Existing approaches such as Prospector and Strathcona cannot address these problems, since the queries for these problems fall out of the scope of these two tools: J2SE, Eclipse, and Eclipse GEF (Graphical Editing Framework). The results indicate that PARSEWeb is able to solve these real programming problems.

Jakarta BCEL User Forum. The Byte Code Engineering Library (BCEL) provides the ability to analyze, create, and manipulate Java bytecode files. We collected the programming problem "How to disassemble Java byte code" posted in the BCEL forum [16]. The programming problem describes that the programmer is using the BCEL library and has Java byte code under analysis. In the BCEL library, Java byte code is represented through the `Code` class. The programmer wants to obtain a Java assembler command list, which is represented in the form of instructions in the BCEL library. Therefore, we identified the relevant query for the given problem description as "`Code` → `Instruction`". PARSEWeb suggested a solution for the query as shown below:

[3] `http://snobol.cs.berkeley.edu/prospector/`
[4] `http://strathcona.cpsc.ucalgary.ca/`

```
01:FileName:2_RepMIStubGenerator.java MethodName: isWriteMethod
                                              Rank:1 NumberOfOccurrences:1
02:Code,getCode() ReturnType:#UNKNOWN#
03:CONSTRUCTOR,InstructionList(#UNKNOWN#) ReturnType:InstructionList
04:InstructionList,getInstructions() ReturnType:Instruction
```

The suggested solution is the same as the response posted in the forum. The programmer can refer to a related code example by browsing the `isWriteMethod` method in the file `2_RepMIStubGenerator.java`. The original code example collected from the preceding method is as follows:

```
Code code;
InstructionList il = new InstructionList(code.getCode());
Instruction[] ins = il.getInstructions();
```

In the preceding code example suggested by PARSEWeb, the return type of `getCode` method is described as `UNKNOWN`. The keyword `UNKNOWN` denotes that PARSEWeb is not able to infer the return type through its partial-program analysis, since the return type of `getCode` method is not explicitly specified in the code example. However, PARSEWeb still correctly suggested to pass the return type directly to the constructor of `InstructionList`.

Dev2Dev Newsgroups. We applied PARSEWeb on another problem "how to connect db by sessionBean" posted in the Dev2Dev Newsgroups [17]. We transformed the question into the query "`InitialContext` $\rightarrow$ `Connection`" and used PARSEWeb. PARSEWeb suggested the following solution, which is the same as the one described in the forum.

```
01:FileName:3_AddrBean.java MethodName:getNextUniqueKey
                                              Rank:1 NumberOfOccurrences:34
02:InitialContext,lookup(String) ReturnType:DataSource
03:DataSource,getConnection() ReturnType:Connection
```

Real Project

We next show that PARSEWeb-suggested MISs exist in a real project, and compare the results with those of two related tools: Prospector and Strathcona. As described by Bajracharya et al. [13], there is still a need (but lack) of a benchmark for open source code search that can be used by similar tools for comparing their results. In our evaluation, we used an open source project *Logic* [79] as a subject project. The *Logic* project was developed based on Eclipse Graphical Editing Framework (GEF). The reason for choosing *Logic* for evaluation is that *Logic* is one of the standard example projects delivered with the Eclipse GEF framework.

To avoid bias in our evaluation results, we chose all queries from the largest source file ("LogicEditor.java") of the subject project. By choosing the largest file, we can also find many queries that can be used to evaluate all three tools. Within the source file, we picked the first ten available queries of the form "*Source* → *Destination*" from the beginning of the class, and used all three tools to suggest solutions for each query. The query selection process is based on two criteria: a new object type is instantiated from one of the known object types and the selected query is the maximal possible query, which we shall explain next through the code example extracted from the source file used in the evaluation:

```
01:public void createControl(Composite parent) {
02:    PageBook pageBook = new PageBook(parent, SWT.NONE);
03:    Control outline = getViewer().createControl(pageBook);
04:    Canvas overview = new Canvas(pageBook, SWT.NONE);
05:    pageBook.showPage(outline);
06:    configureOutlineViewer();
07:    hookOutlineViewer();
08:    initializeOutlineViewer();
09:}
```

The possible queries that can be extracted from the preceding code example are "Composite → PageBook", "Composite → Control", and "Composite → Canvas". However, the maximal possible query among these three queries is "Composite → Canvas", since this query subsumes the other two queries. We consider a task as successful only when the suggested code example is the same as the code snippet in the corresponding subject project. Since PARSEWeb tries to suggest solutions from available open source repositories, which may include the subject project under consideration, we excluded the results of PARSEWeb that are suggested from the subject project under consideration.

Since both PARSEWeb and Prospector accept the query of the preceding form, we gave constructed queries directly as input. Strathcona compares the context of code given as input and suggests relevant code examples. Therefore, for each evaluation, we built separate driver code that can convey the context of the query. In the driver code, we declared two local variables with the *Source* and *Destination* object types, respectively.

We used PARSEWeb, Prospector, and Strathcona to suggest solutions for each query. The results of our evaluation are shown in Table 3.1. In Columns "PARSEWeb", "Prospector", and "Strathcona", Sub-columns "#" and "Rank" show the number of results returned by each tool, and rank, respectively, of the suggested solution that matches with the original source code of the subject project from which the query is constructed. The maximum number of results returned by PARSEWeb, Prospector, and Strathcona are 10, 12, and 10, respectively. The last column "GCSE" shows the index of the source file that contained the solution among

Table 3.1: Evaluation results of programming tasks from the Logic Project.

Query		PARSEWeb		Prospector		Strathcona		GCSE
Source	Destination	#	Rank	#	Rank	#	Rank	
IPageSite	IActionBars	1	1	3	1	10	7	1
ActionRegistry	IAction	3	1	4	1	10	3	2
ActionRegistry	ContextMenuProvider	Nil	Nil	2	2	10	3	NA
IPageSite	ISelectionProvider	1	1	12	1	10	Nil	5
IPageSite	IToolBarManager	2	1	12	1	10	6	9
String	ImageDescriptor	10	6	12	Nil	10	Nil	28
Composite	Control	10	2	12	Nil	10	Nil	72
Composite	Canvas	10	5	12	Nil	10	Nil	28
GraphicalViewer	ScrollableThumbnail	2	1	12	8	10	7	2
GraphicalViewer	IFigure	1	Nil	12	Nil	10	Nil	NA

the results by GCSE. This index information is extracted by identifying the first source file in which the resultant MIS is found. We found that both PARSEWeb and Prospector performed better than Strathcona. Between PARSEWeb and Prospector, PARSEWeb performed better than Prospector. We next discuss the results of each tool individually.

From the results, we observed that PARSEWeb suggested solutions for all queries except for two. The reason behind the better performance of PARSEWeb is that PARSEWeb suggests solutions from reusable code examples. We inspected queries for which PARSEWeb could not suggest any solution and found the reason is a limitation of our approach in analyzing partial code examples (discussed in Section 2.3). Prospector tries to solve the given query using API signatures. Therefore, it can often find some feasible solution for a given query, as it can find a path from the given *Source* to *Destination*. One reason for not getting complete results with Prospector in our evaluation could be that Prospector shows only first twelve results of the given query. Due to this limitation, the required solution might not have shown in the suggested set of solutions. Prospector solves the queries through API signatures and has no knowledge of which MISs are often used compared to other MISs that can also serve as a solution for the given query. PARSEWeb performs better in this scenario, since PARSEWeb tries to suggest solutions from reusable code examples and is able to identify MISs that are often used for solving a given query. For example, for query "`Composite` → `Canvas`", the solution is through an additional class called `PageBook`. Although this solution is often used, Prospector is not able to identify the solution, since this solution can be a less favorable solution from the API signature point-of-view.

We suspect that the reason for not getting good results with Strathcona is that Strathcona cannot effectively address the queries of the form "*Source* → *Destination*". We observed that Strathcona generates relevant solutions when the exact API is included in the search context. However, our described problem is to identify that API, since the programmer has no knowledge of which API has to be used for solving the query. Moreover, we found that many code examples

returned by Strathcona contain both *Source* and *Destination* object types in either `import` statements or in different method declarations. Therefore, those code examples cannot address our query, since no MIS can be derived to lead from *Source* to *Destination* object types.

The results shown in Column "GCSE" indicate the problems that may be faced by programmers in using CSEs directly. For example, to find the solution for the seventh task, the programmer has to check 72 source files in the results of GCSE.

Comparison of PARSEWeb and Prospector

We next present the evaluation results of PARSEWeb and Prospector[5] for 12 specific programming tasks. These tasks are based on Eclipse plugin examples from the *Java Developer's Guide to Eclipse* [10] and are the same as the first 12 tasks used by Sahavechaphan and Claypool [108] in evaluating their XSnippet tool. We have not chosen the remaining five tasks used in evaluating the XSnippet tool as they are the same as some previous tasks, but differ in the code context where the tasks are executed. Since neither PARSEWeb nor Prospector considers the code context, these five tasks are redundant to use in our evaluation.

The primary reason for selecting these tasks is their characteristics that include different Java programming aspects like object instantiation via a constructor, a static method, and a non-static method from a parent class. These tasks also require downcasts and have reasonable difficulty levels. For each task, all necessary Java source files and Jar files are provided and code for getting the *Destination* object from the *Source* object is left incomplete. We used open source projects such as `org.eclipse.jdt.ui`, and examples from Eclipse corner articles[6] for creating the necessary environment. We used PARSEWeb and Prospector to suggest solutions for each query. A task is considered as successful if the final code can be compiled and executed, and the required functionality is enabled with at least one suggested solution. The task is also considered as successful if the suggested solution acts as a starting point and the final code could be compiled with some additional code. Prospector can generate compilable code for its suggested solutions, but the current implementation of PARSEWeb suggests only the frequent MISs and code examples, but cannot directly generate compilable code. Therefore, we manually transformed the suggested sequences into appropriate code snippets.

Table 3.2 shows the results of 12 programming tasks. PARSEWeb is not able to suggest solution for only one query, whereas Prospector failed to suggest solutions for five queries. This result demonstrates the strength of PARSEWeb, since PARSEWeb suggests solutions from reusable code examples gathered from publicly available source code repositories. Figure 3.5 shows a summary of percentage of tasks successfully completed by each tool along with the

[5]We chose only Prospector for detailed comparison, since another related tool XSnippet [108] was not available and Strathcona did not perform well in addressing the described problem based on the previous evaluation.

[6]`http://www.eclipse.org/articles/`

Table 3.2: Evaluation results of programming tasks previously used in evaluating XSnippet.

Query		PARSEWeb	Prospector
Source	Destination		
ISelection	ICompilationUnit	Yes	No
IStructuredSelection	ICompilationUnit	Yes	Yes
ElementChangedEvent	ICompilationUnit	Yes	Yes
IEditorPart	ICompilationUnit	Yes	Yes
IEditorPart	IEditorInput	Yes	Yes
ViewPart	ISelectionService	Yes	Yes
TextEditorAction	ITextEditor	Yes	No
TextEditorAction	ITextSelection	Yes	No
ITextEditor	ITextSelection	Yes	Yes
AbstractDecoratedTextEditor	ProjectViewer	No	No
TextEditor	IDocument	Yes	No
TextEditor	ITextSelection	Yes	Yes

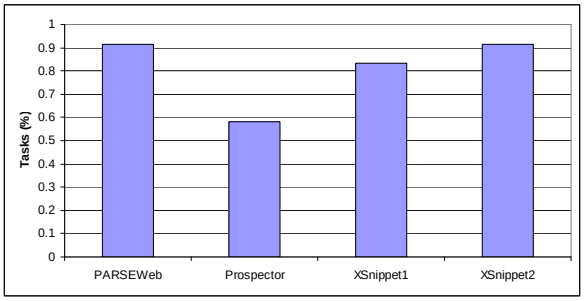


Figure 3.5: Percentage of tasks completed by PARSEWeb, Prospector, and XSnippet.

results collected from the XSnippet [108] approach. The x-axis shows different tools and the y-axis shows the percentage of tasks successfully completed by each tool. The "XSnippet1" and "XSnippet2" entries show two XSnippet query-type techniques: *Type-Based Instantiation Query* (IQ_T) and *Generalized Instantiation Query* (IQ_G), respectively. PARSEWeb performed better than Prospector and XSnippet's IQ_T query type. The results of PARSEWeb are at par with XSnippet's IQ_G query type. However, the IQ_G query type of XSnippet cannot effectively address the issue targeted by PARSEWeb as this query type simply returns the set of all code examples contained in the example repository that instantiate the given *Destination* object type, irrespective of the *Source* object type. Moreover, XSnippet is also limited to the queries of a specific set of libraries.

Significance of PARSEWeb Techniques

We next show the significance and impact of different techniques used in PARSEWeb. We picked some of the queries in preceding evaluations and analyzed different techniques of our approach. The results of our analysis are shown in Table 3.3. The table shows the number of identified MISs after applying respective techniques. As shown in the results, the method

Table 3.3: Evaluation results of PARSEWeb internal techniques.

Query		Simple	Method Inline	Mine Phase	Query Splitter
Source	Destination				
TableViewer	TableColumn	21	23	2	2
IWorkbench	IEditorPart	13	17	8	8
IWorkBenchPage	IStructuredSelection	5	6	1	1
Composite	Control	26	29	24	24
IEditorSite	ISelectionService	Nil	Nil	Nil	2

inlining technique increases the possible number of sequences, whereas the *mine* phase helps in reducing the number of sequences by clustering similar sequences. The query splitter phase helps address the lack of code examples by splitting the query into different sub-queries.

Evaluation Summary

The primary advantage of PARSEWeb compared to other related tools is that PARSEWeb is not limited to the queries of any specific set of libraries, since PARSEWeb is developed based on our WebMiner framework. We showed this advantage in the first part of our evaluation. Although Prospector solves the queries of a specific set of libraries from the API signatures, we showed that the results of PARSEWeb are better than the results of Prospector. The reason is that Prospector has no knowledge of which MISs are often used compared to other possible sets of sequences. This lack of information often results in irrelevant solutions. Although both PARSEWeb and Strathcona suggest solutions from code examples, the results of PARSEWeb are better than Strathcona, since the number of available code examples are limited for Strathcona. Moreover, PARSEWeb has many specialized heuristics compared to Strathcona for helping identify the required MIS. We also showed that GCSE alone cannot handle the queries of the form "*Source* $\rightarrow$ *Destination*", and showed the significance and impact of different techniques in PARSEWeb.

3.3 SpotWeb: Detecting framework hotspots and coldspots

3.3.1 Motivation

In general, libraries expose certain areas (API classes and methods) of flexibility that are intended for reuse by their users. Programmers who reuse classes and methods of these libraries must be aware of these flexible areas for effective reuse of libraries. These areas of flexibility are often referred as *hotspots*. As described by Pree [99], hotspots depict a framework's (or library's) flexibility and proneness to reuse. The foundations of hotspots are built upon the Open-Closed principle by Martin [84]. The Open-Closed principle encompasses two main definitions: the

"open" and the "closed" parts. The "open" parts (referred as *hooks*) represent areas that are flexible and variant, whereas the "closed" parts (referred as *templates*) represent areas that are immutable in the given framework or library. A *hotspot* is defined as a combination of templates and hooks.

Hotspots are useful to both users and programmers of the library in several ways. First, new users can browse and inspect hotspots to understand commonly reused classes and find out the classes that the users want to reuse. Second, users may have more confidence or tendencies in reusing hotspots, since generally defects in these hotspots may be fewer (or more easily exposed previously) than the ones in non-hotspots; we can view the application code that reuses library hotspots to be a special type of test code that can help expose bugs in hotspots. Third, developers or maintainers of these libraries can choose to invest their improvement efforts (e.g., performance improvement or bug fixing) on these hotspots, since the resulting returns on investment may be substantial.

In contrast to hotspots, we call a library's areas that are rarely used by users as *coldspots*. The concept of coldspots is introduced by our approach and these coldspots can serve as caveats to users of the given library. Since coldspots represent rarely used classes and methods, there can be difficulties in identifying relevant code examples that can help users in reusing those classes and methods. Moreover, coldspots are generally less tested compared to hotspots with regards to the "testing" conducted by API client code as test code.

Detecting hotspots and coldspots of a library under analysis requires domain knowledge of how the API classes and methods of the input library are reused by applications, referred as client applications. Various open source projects that reuse classes of the library under analysis are available on the web and these open source projects can serve as a basis for gathering the information of how classes of the input library are reused, and hence can help in detecting hotspots and coldspots. Therefore, our approach, called *SpotWeb*, developed based on our WebMiner framework, leverages a CSE to gather relevant code examples of classes of the library under analysis from these open source projects. SpotWeb analyzes gathered code examples statically, and detects hotspots and coldspots of the library under analysis.

SpotWeb makes the following major contributions:

- A technique for detecting hotspots of a library under analysis by analyzing relevant code examples gathered from a CSE.
- A technique for detecting coldspots of a library under analysis.
- An Eclipse plugin implemented for the proposed approach and several evaluations to assess the effectiveness of the tool. In our evaluation, SpotWeb detects hotspots and coldspots of eight widely used open source libraries by analyzing a total of 7.8 million lines of code. We show the utility of detected hotspots by comparing detected hotspots

of a library with a real application reusing that library. We also compare our results with the results of a previous related approach by Viljamaa [64].

3.3.2 Example

We next use an example to explain our SpotWeb approach and show how the detected hotspots and coldspots can be used by the library users. We use JUnit [66], the *de facto* standard unit testing library for Java, as an illustrative example for explaining our approach.

SpotWeb accepts a library under analysis, say JUnit, and extracts *LibraryInfo* from the framework. The *LibraryInfo* includes all classes, all interfaces, public or protected methods of each class and interface, and inheritance hierarchy among classes or interfaces of the library. SpotWeb also captures the constants defined by the input library. SpotWeb constructs different queries for each class or interface and interacts with a CSE such as Google code search [48] to collect relevant code examples from existing open source projects that reuse the classes of the input library. For example, SpotWeb constructs a query such as "`lang:java junit.framework.TestSuite`" for collecting relevant code examples of the `TestSuite` class. These collected code examples are referred as a *LocalRepository* for the input library. SpotWeb analyzes collected code examples statically and computes *UsageMetrics* for classes, interfaces, and public or protected methods of all classes and interfaces. For example, the *UsageMetrics* computed for the `TestSuite` class show that the class is instantiated for 165 times and is extended for 32 times. Similarly, the *UsageMetrics* computed for the method `addTest` of the `TestSuite` class show that the method is invoked for 95 times. SpotWeb also collects code examples for each class or method and stores these code examples in a repository, referred as *ExampleDB*. Then SpotWeb uses the algorithm shown in Algorithm 2 for detecting hotspots from the computed *UsageMetrics*.

Initially, SpotWeb ranks methods in a non-ascending order based on their *UsageMetrics* and uses a threshold percentage HT to detect hotspot methods: the methods in the top HT percentage with a non-zero *UsageMetrics* are detected as hotspot methods. The detected hotspot methods are then grouped into their declaring classes, detected as hotspot classes. These hotspot classes are ranked based on the minimum rank of the hotspot methods declared by these classes. SpotWeb classifies the hotspot classes into two categories (templates and hooks) based on heuristics described in Steps 9-16 of Algorithm 2. The hotspot classes of each category are further grouped into hierarchies based on their inheritance relationships. For example, SpotWeb detected classes `Assert` and `TestCase` as hook hotspots in the JUnit library. Since `TestCase` class extends `Assert` class, SpotWeb groups both the classes into the same hierarchy. SpotWeb assigns a rank to each hierarchy based on the minimum rank of the hotspot classes contained in the hierarchy. For example, consider that the `Assert` class has Rank

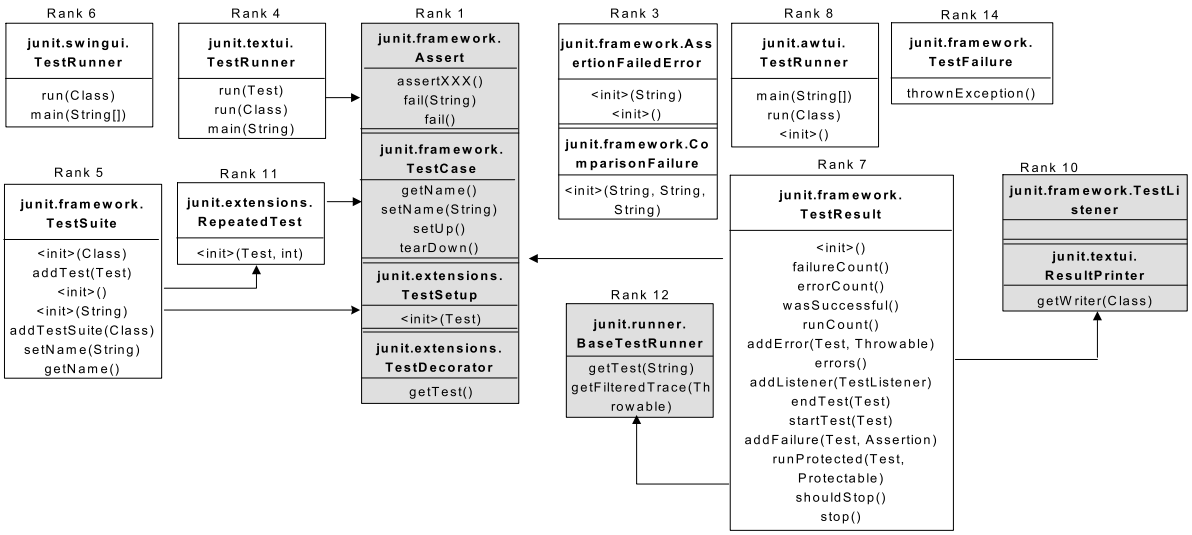


Figure 3.6: Hotspot hierarchies identified for the JUnit library.

1 and the `TestCase` class has Rank 2, then the grouped hierarchy of the `Assert` and `TestCase` classes is assigned with Rank 1. The rank attribute uniquely identifies a hierarchy among all other hierarchies. Hierarchies with smaller ranks have higher preference or importance to the hierarchies with higher ranks.

Figure 3.6 shows the hotspot hierarchies detected for the JUnit library. The figure also shows ranks assigned to each hierarchy. Since the rank attribute uniquely identifies a hierarchy, we use the rank as an identity for describing a hierarchy. Each hierarchy includes one or more hotspot classes and is shown as pairs of class and its methods. For example, Hierarchy 1 (hierarchy with Rank 1) has classes `Assert`, `TestCase`, `TestSetup`, and `TestDecorator`. We show template hierarchies in white and hook hierarchies in gray. For example, Hierarchy 1 is a hook hierarchy and Hierarchy 3 is a template hierarchy.

Methods inside each class of a hierarchy are sorted based on their computed *UsageMetrics*. Sorting methods of a class can assist the library users in quickly identifying the methods that are often used inside a given hotspot class. For example, consider the `TestSuite` class shown in Hierarchy 5. The `TestSuite` class has three constructors `<init>(Class)`, `<init>()`, and `<init>(String)`. However, the `<init>(Class)` constructor is often used compared to the other two constructors. Due to space limit, we show all assertion methods such as `assertEquals` and `assertTrue` of the class `Assert` of Hierarchy 1 as `assertXXX`.

The figure also displays dependencies among hotspot hierarchies (shown as arrows between hierarchies). SpotWeb captures the usage relationships among hotspot classes through dependencies. For example, Hierarchy 5 has a `TEMPLATE_HOOK` dependency with Hierarchy 1. This dependency indicates that to reuse methods such as `addTest` of the class `TestSuite` in Hierarchy 5, the user has to define a new behavior for the classes in Hierarchy 1.

```
01:public class SRDAOTestCase extends TestCase {
02:   private SRDAO dao = null;...
03:   public SRDAOTestCase() {
04:      super(); ...
05:   }
06:   protected void setUp() throws Exception {
07:      ...
08:      dao = (SRDAO)context.getBean("SRDAO");
09:      ...
10:   }
11:   public void tearDown() throws Exception {
12:      dao = null;
13:   }
14:   public void testF() { ... }
15:   public void testB() { ... }
16:   ...
17:}
```

Figure 3.7: Suggested code example for the hook class `TestCase`.

```
01:public class MyTestSuite {
02:   ...
03:   public static Test suite() {
04:      TestSuite suite = new TestSuite("axis");
05:      suite.addTest(new SRDAOTestCase());
06:      return suite;
07:   } ...
08:}
```

Figure 3.8: Suggested code example for the template class `TestSuite`.

We next describe how the hotspots detected by SpotWeb can be used by the library users to reuse classes of the JUnit library. After reviewing the hotspots shown in Figure 3.6, consider that a library user wants to start with the method `addTest` of the template class `TestSuite` in Hierarchy 5. Figure 3.6 shows that Hierarchy 5 of the `TestSuite` class has a `TEMPLATE_-HOOK` dependency with the Hierarchy 1. This dependency indicates that the user may need to define a new behavior for the associated hook hierarchy. SpotWeb recommends the code example shown in Figure 3.7 for the hook class `TestCase`, which is part of Hierarchy 1. The code example exhibits several aspects that need to be handled by the user while extending the `TestCase` class. For example, in the `setUp` method, the user can write code for setting up the environment such as instantiating necessary variables, and in the `tearDown` method, the user can destroy the created variables. In addition, the code example shows that names of the test methods in the extended class of the `TestCase` class should start with the prefix `test`. SpotWeb also recommends a code example for the `addTest` method and the recommended code example is shown in Figure 3.8. The code example shows that the user has to create an instance of the `TestSuite` class and then add test cases through the `addTest` method.

An API class or method is identified as a coldspot if that class or method is neither used directly nor used indirectly by collected code examples. The complete algorithm used for de-

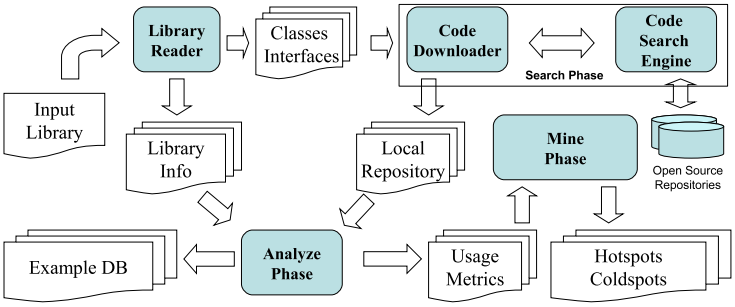


Figure 3.9: Overview of the SpotWeb approach

tecting coldspots is shown in Algorithm 3. SpotWeb identified 20 classes such as `Swapper`, `TestRunListener`, and `ExceptionTestCase` as coldspots in the JUnit library. However, coldspots are only suggestions for users unfamiliar to that library and SpotWeb does not intend to recommend users not to reuse those coldspot classes. Sometimes, coldspots can also be helpful to the library developers in distributing their maintenance effort, since the library developers can give a low preference to the coldspot classes.

3.3.3 Approach

SpotWeb includes an additional phase, called *library reader*, along with the following three major phases of our WebMiner framework: *search*, *analyze*, and *mine*. In SpotWeb, the *apply* phase simply includes suggesting final output of the *mine* phase to the programmer. Figure 3.9 shows an overview of all phases and flows among different phases. We use JUnit as an illustrative example. We next explain all phases except the *search* phase, which is the same as our WebMiner framework.

Library Reader

The library reader component takes a library, say JUnit, as input and extracts the *LibraryInfo* information. The *LibraryInfo* includes all classes, all interfaces, public or protected methods of each class and interface.

Analyze Phase

In the *analyze* phase, SpotWeb analyzes the code examples collected in the *search* phase. These collected code examples are stored in a repository, referred to as *LocalRepository*. SpotWeb applies partial-program analysis on code examples stored in *LocalRepository* (as described in Section 2.3 of our WebMiner framework). SpotWeb next computes *UsageMetrics* for all classes, methods, exceptions, and constants of the library under analysis. The *UsageMetrics* capture several ways of how often a class or an interface or a method of the library under analysis is used by gathered code examples. The *UsageMetrics* for a class include the number of created

instances (more precisely, the number of constructor-call sites) and the number of times that the class is extended. For an interface, the *UsageMetrics* include the number of times that the interface is implemented. We use notations IN_j, EX_j, and IM_j for the number of instances, the number of extensions, and the number of implementations, respectively. The consolidated usage metric UM_j for a class or an interface is the sum of all the three preceding metrics. The *UsageMetrics* for an exception class include the number of times that the exception class is used in the `catch` blocks or the `throw` statements.

SpotWeb computes three types of *UsageMetrics* for methods: Invocations, Overrides, and Implements. The *Invocations* metric gives the number of times that the method is invoked by the code examples. The *Overrides* metric gives the number of times that the method is overridden by the code examples to define a new behavior. The *Implements* metric, specific for interfaces, gives the number of times that the method is implemented. SpotWeb considers `static` methods as regular methods. For constructors, SpotWeb computes only the *Invocations* metric. We use notations IN_i, OV_i, and IM_i for invocations, overrides, and implementations, respectively. The overall usage metric (UM_i) for a method is the sum of all the three preceding metrics.

SpotWeb identifies constants defined by the input library through the Java keywords such as `final` and `static`. The *UsageMetrics* for such a constant include the number of times that the constant is referred by code examples gathered from CSE. SpotWeb also gathers code examples for each class or method and stores these code examples in a repository, referred as `ExampleDB`. The `ExampleDB` is used for recommending relevant code examples for a class or a method requested by the user. The relevant code examples can further assist users in making an effective reuse of API classes and methods of the input library.

Mine Phase

In the *mine* phase, SpotWeb detects hotspots and coldspots in the library under analysis.

Detecting hotspots. We next explain how SpotWeb detects hotspots using computed *UsageMetrics*. Algorithm 2 shows the core algorithm used by SpotWeb for detecting hotspots. We next describe the algorithm through an illustrative example shown in Figure 3.10. The figure shows four classes C_1, C_2, C_3, and `ExceptClass`, and their declarations. The class C_3 is an `abstract` class. The `ExceptClass` is an `Exception` class that can appear in exception-handling constructs such as `catch` blocks. The figure also shows computed usage metrics for each class, and its methods and constant variables. For example, the class C_1 is instantiated for 10 times (shown as `IN`=10) and the abstract class C_3 is extended for 12 times (shown as `EX`=12). The method m_{2_1} is invoked for 6 times and is overridden for 2 times. Similarly, the constant `constC1`

Algorithm 2 Algorithm for detecting hotspots through computed *UsageMetrics*

Require: UsageMetrics of classes and methods, HT percentage
Ensure: Hotspot hierarchies and their dependencies

1: $SortedMET$ = Sort methods based on their usage metric values
2: **for all** MET_i in $SortedMET$ **do**
3: **if** $UM_i \neq 0$ && Position of $MET_i \leq (HT$ * Size of $SortedMET)$ **then**
4: Set MET_i type as $HOTSPOT$
5: **end if**
6: **end for**
7: $\{C_1, ... C_n\}$ = Group $HOTSPOT$ MET_i based on their declaring classes
8: //Assign ranks to each C_i and classify into templates and hooks
9: **for all** C_i in $\{C_1, ... C_n\}$ **do**
10: Rank of C_i = Minimum rank among all MET_i of the C_i
11: **if** C_i is an *Interface* or *Abstract class* or $(EX_i > IN_i)$ **then**
12: Set type of C_i to $HOOK$
13: **else**
14: Set type of C_i to $TEMPLATE$
15: **end if**
16: **end for**
17: Group C_i of the same type into hierarchies based on inheritance
18: Associate hook hierarchies to template hierarchies
19: Define dependencies between template hierarchies
20: **return** hook and template hierarchies as hotspot hierarchies

is accessed 6 times and the exception class `ExceptClass` is detected in `catch` blocks for 9 times among gathered code examples.

First, SpotWeb sorts UM values of all methods, constants, and exception classes. SpotWeb uses a threshold percentage (referred as HT) and selects the top HT methods, whose usage metric is non-zero, as hotspot methods. For example, for a HT of 45%, SpotWeb identifies the methods such as m_{3_1}, m_{3_2}, m_{3_3}, and c_1 as hotspot methods. SpotWeb groups the hotspot methods based on their declaring classes. The resulting classes are sorted based on the minimum rank among included hotspot methods in each class. In the current example, the grouping process results in classes C_3 (methods: m_{3_1}, m_{3_2}, and m_{3_3}), C_1 (methods: c_1 and m_{1_1}), and C_2 (methods: c_2 and m_{2_1}). After grouping, SpotWeb uses computed metrics of classes to classify these classes further into templates and hooks. The criteria used for classifying hotspot classes into templates and hooks are shown in Step 4 of the algorithm. For the current example, SpotWeb identifies class C_3 as a `HOOK` class, and classes C_1 and C_2 as `TEMPLATE` classes. SpotWeb further groups the classes of the same category based on their inheritance relationship. For example, if C_1 has a parent class P_1 and both classes are classified as `TEMPLATE` classes, SpotWeb groups C_1 and P_1 into the same hierarchy.

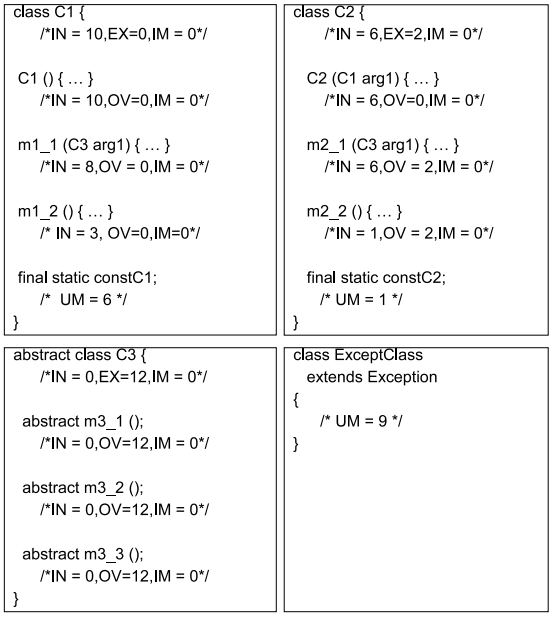


Figure 3.10: Example classes of a sample library

SpotWeb identifies dependencies among detected hotspot hierarchies based on arguments passed to methods of those classes. For example, if a template class, say X, has a constructor that requires an instance of another template class, say Y, then SpotWeb captures dependency of the form "X $\rightarrow$ Y", which describes that X requires Y. SpotWeb identifies two kinds of dependencies: `TEMPLATE_HOOK` and `TEMPLATE_TEMPLATE`. A `TEMPLATE_HOOK` dependency defines a relationship between a template hierarchy and a hook hierarchy. SpotWeb identifies that a template hierarchy is dependent on a hook hierarchy if methods in the template hierarchy types include some classes in the hook hierarchy as arguments. Such a dependency describes that the users have to first define a new behavior for those related hook classes, say extend the classes, and use the instances of those classes as arguments. For example, SpotWeb identifies that the class C_1 has a `TEMPLATE_HOOK` dependency with the class C_3 as the method m_{1_1} requires an instance of C_3 as an argument. Similarly, SpotWeb identifies `TEMPLATE_TEMPLATE` hierarchies when one template hierarchy is dependent on another template hierarchy. For example, the class C_2 has a `TEMPLATE_TEMPLATE` dependency with the class C_1.

Detecting coldspots. SpotWeb next identifies classes and methods (of the library under analysis) that are rarely or never used by gathered code examples as coldspots. However, detecting coldspots based on only the *UsageMetrics* can give many false positives. For example, the *UsageMetrics* for an abstract method defined in a class can be zero, since gathered code examples refer to the concrete implementation provided by some of the abstract classes's subclasses. In this case, this abstract method is not a coldspot as the method is indirectly referenced through the subclasses. Therefore, to reduce the number of false positives while

Algorithm 3 Algorithm for detecting whether a method is a coldspot or not

Require: A method M_i of a class C_j, and UsageMetrics of M_i
Ensure: Is the method a coldspot or not?

```
1: if M_i is reused atleast once then
2:     return false
3: end if
4: if C_j is an interface then
5:     if all implemented methods of M_i are coldspots then
6:         return true
7:     else
8:         return false
9:     end if
10: end if
11: if M_i is abstract then
12:     if all overridden methods of M_i are coldspots then
13:         return true
14:     else
15:         return false
16:     end if
17: end if
18: if all callers of M_i are coldspots then
19:     return true
20: else
21:     return false
22: end if
```

identifying coldspots, SpotWeb uses a recursive algorithm shown in Algorithm 3. The *UsageMetrics* referred to by the algorithm is the sum of metrics *Invocations*, *Overrides*, and *Implements* computed for a method. If computed *UsageMetrics* are not zero, then the method is identified as not a coldspot (Steps 1-3). If the current method belongs to an interface, our algorithm recursively checks whether all the corresponding methods in the implemented classes of the input library are also coldspots. If all of them are coldspots, SpotWeb identifies the current method as a coldspot (Steps 4 - 10). SpotWeb uses a similar approach when the method is abstract (Steps 11 - 17). If the current method under analysis does not belong to any of the preceding categories, SpotWeb checks whether all other methods of the input library that invoke the current method are also coldspots. Step 18 of the algorithm (related to callers) is performed to identify indirect usages of a method of the input library. SpotWeb groups detected coldspot methods into their declaring classes.

Table 3.4: Subjects used for evaluating SpotWeb.

Subject	# Classes	# Methods	# Samples	# KLOC	URL
Log4j	207	1543	9768	2064	`www.logging.apache.org/log4j`
JUnit	56	531	8891	1558	`www.junit.org`
JGraphT	177	931	289	30	`www.jgrapht.sourceforge.net`
Grappa	44	561	2071	1978	`www.graphviz.org`
OpenJ	210	1365	1076	113	`www.openjgraph.sourceforge.net`
JUNG	461	3241	2390	353	`www.jung.sourceforge.net`
BCEL	357	3048	5225	1219	`www.jakarta.apache.org/bcel`
Javassit	249	2149	3226	631	`www.csg.is.titech.ac.jp/chiba/javassist`

Implementation

We developed SpotWeb as an Eclipse plugin. SpotWeb can be invoked by selecting a menu item available on projects in Eclipse. In the SpotWeb implementation, we used the HT percentage of 15%, which is derived based on our initial empirical experience.

3.3.4 Evaluation

We evaluated SpotWeb with eight widely used open source libraries, which differ in size and purpose. In our evaluation, we investigate the following research questions.

- RQ1: What is the percentage of hotspot and coldspot classes and methods among the total number of classes and methods in each library, respectively? This research question helps to characterize the usages of a library and reuse effort of the library.
- RQ2: Is the subset of classes and methods detected as hotspots indeed useful in helping effective library reuse? We address the preceding question by showing that detected hotspots include classes and methods of a library reused by a real application. This evaluation helps to show that SpotWeb can help reduce the effort of users by suggesting a subset of classes and methods as hotspots.
- RQ3: What is the effectiveness of our hotspot detection in terms of precision and recall? We address the preceding question through two evaluations. First, we compare the detected hotspot classes with the classes described in the documentation associated with the libraries. Second, we compare the detected hotspot classes with the hotspots detected by a previous related approach by Viljamaa [64].

Subjects

The subjects used in our evaluation and their characteristics such as the number of classes and methods are shown in Columns "Classes" and "Methods" of Table 3.4. Log4j is a library used

Table 3.5: Evaluation results showing the detected hotspots and coldspots.

Subject	Classes	Hotspot classes					Coldspot classes		Hotspot methods		
	(#)	#Classes	%	#Templ	#Hooks	#Dep	#Classes	%	#Total	#Methods	%
Log4j	207	56	27.05	35	11	22	131	63.28	1543	299	19.38
JUnit	56	14	25	8	3	7	22	39.28	531	77	14.50
JGraphT	177	41	23.16	20	8	0	135	76.27	931	102	10.96
Grappa	44	16	36.36	11	2	3	23	52.27	561	50	8.91
OpenJGraph	210	35	16.67	21	6	7	172	81.90	1365	76	5.57
JUNG	461	195	42.29	128	17	109	222	48.15	3241	569	17.55
BCEL	357	121	33.89	74	9	59	192	53.78	3048	580	19.03
Javassit	249	70	28.11	57	4	14	175	70.28	2149	371	17.26

for inserting logs in source code. JUnit is the *de facto* standard unit testing library for the Java programming language. JGraphT, Grappa, OpenJGraph (OpenJ), and JUNG are four graph libraries that provide several graph manipulation utilities. BCEL and Javassit are libraries that provide functionality for creating, analyzing, and manipulating Java class files.

Column "Samples" of Table 3.4 shows the number of code examples gathered from Google code search [48] and Column "KLOC" shows the total number of kilo lines of Java code analyzed by SpotWeb for identifying hotspots and coldspots. One of the major advantages of SpotWeb compared to other approaches is the large number of analyzed code examples that can help detect hotspots and coldspots effectively.

RQ1: Statistics of Hotspots and Coldspots

We next address the first question on the percentage of classes and methods classified as hotspots or coldspots. These statistics help identify the possibilities of reuse in a library and the amount of effort required by a new user in getting familiar with the library under analysis. Table 3.5 shows the statistics of hotspots in all libraries. Column "Subject" shows the name of the input library. Columns "Hotspot classes" and "Coldspot classes" present the number of classes classified as hotspots and coldspots, respectively. Column "Hotspot methods" shows the number of methods identified as hotspot methods. Sub-columns "Classes" and "%" of Column "Hotspot classes" show the number of hotspot classes and their percentages among the total number of classes. Columns "Templ", "Hooks", and "Dep" give the number of template hierarchies, hook hierarchies, and their dependencies, respectively. Sub-columns "Classes" and "%" of Column "Coldspots" show the number of coldspot classes and their percentages.

Our results show that the percentage of hotspots for all subjects ranges from 16% to 42%, whereas the percentage of coldspots ranges from 39% to 82%. These statistics help identify the effort in reusing a given library. For example, the required effort for reusing the JUnit library can be low compared to the effort required for reusing the JUNG library, since the number of hotspots of the JUNG library is greater than the number of hotspots of the JUnit library.

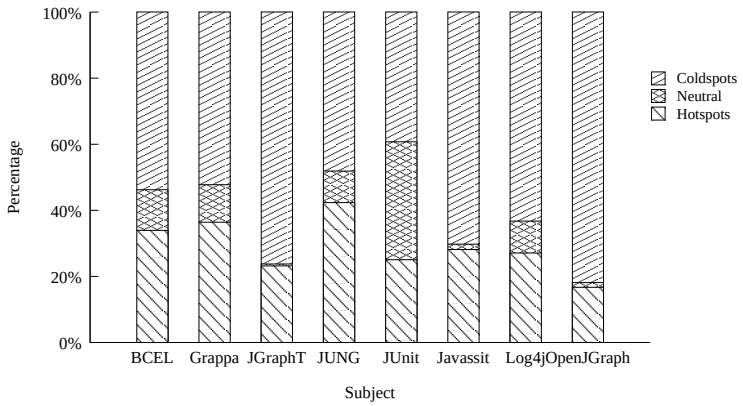


Figure 3.11: Distribution of hotspot and coldspot percentages in all subject libraries.

Figure 3.11 presents the distribution of hotspot and coldspot percentages of all subjects. The distribution chart shows that OpenJGraph and JGraphT libraries have the lowest percentage of hotspots and the highest percentage of coldspots. This scenario can provide a hint that only a few classes of these libraries are often reused. In the figure, we also show a new classification called "Neutral", which represents classes that do not belong to either the hotspot or coldspot category. The graph shows that the percentage of classes in the Neutral category is relatively low for all subjects except JUnit. This characteristic indicates that a class is either reused heavily or is never reused, and only in a few cases a class is occasionally reused.

We next describe a few example hotspot classes detected for the four graph libraries JGraphT, Grappa, OpenJGraph, and Grappa. Each graph library provides several different types of graphs. For example, the JUNG library provides graphs such as `DirectedGraph`, `ArchetypeGraph`, and `HyperGraph`. SpotWeb identified the graph type that is commonly used among all different types provided by each library. For example, SpotWeb identified that graphs `DefaultListenable Graph`, `DirectedSparseGraph`, `DirectedGraphImpl`, and `Graph` are the commonly used graph types in JGraphT, JUNG, OpenJGraph, and Grappa graph libraries, respectively.

RQ2: Utilities of Hotspots

We next address the second question on whether the subset of classes and methods detected as hotspots is indeed useful in helping effective library reuse. We use DNSJava[7], a popular library that provides implementation of DNS in Java, as a library under analysis. We choose DNSJava for two primary reasons: DNSJava is used as a subject in several previous approaches and the DNSJava webpage provides example applications that can be used to validate the detected hotspots. We classify all DNSJava reusing applications available on the web through Google code search (except the James[8] application) as training applications for SpotWeb to detect

[7] `http://www.dnsjava.org/`
[8] `http://james.apache.org/`

Table 3.6: Hotspots of DNSJava reused by James.

	James	SpotWeb	%
Classes	17	16	94.11
Methods	18	16	88.88
Exceptions	1	1	100
Constants	7	7	100

hotspots of DNSJava. We validate the detected hotspots using James as a test application. We selected James as the test application, since James is one of the example applications described in the webpage of DNSJava.

To show the utility of detected hotspots, we identify the DNSJava classes and methods that are reused by James and compute the percentage of those classes detected by SpotWeb. Table 3.6 shows the results of our evaluation. DNSJava includes 151 classes and 1224 methods. The James application reused 17 classes and 18 methods of DNSJava. James also used 1 exception class and 7 constants declared by DNSJava. Table 3.6 shows the evaluation results. Columns "James" and "SpotWeb" show the classes, methods, exceptions, and constants used by James and are among the detected hotspots by SpotWeb. The hotspots detected by SpotWeb include 16 classes and 16 methods reused by James. Moreover, SpotWeb also correctly detected all seven constants referred by James. SpotWeb could not detect one hotspot class, called `Resolver`. The primary reason is that `Resolver` is an interface and gathered code examples do not include any usages of the `Resolver` interface. This evaluation shows that the detected hotspots are indeed useful in helping effective library reuse and can help reducing the effort of a programmer unfamiliar to the library by suggesting a subset of classes and methods as hotspots.

RQ3: Effectiveness of Hotspot Detection

We next address the third question regarding the effectiveness of our hotspot detection with respect to precision and recall through two evaluations. First, we compare the detected hotspots of JUnit and Log4j libraries with available documentations. Second, we compare SpotWeb results with the results of a previous approach by Vijamaa [64].

Comparison with documentation. We next analyze the effectiveness of hotspot detection through the evaluation results with Log4j and JUnit libraries. The primary reason for selecting Log4j[9] and JUnit[10] for analysis is the availability of their documentation that can help validate the detected hotspots.

[9] `http://logging.apache.org/log4j/docs/manual.html`
[10] `http://junit.sourceforge.net/doc/cookstour/cookstour.htm`

Table 3.7: Hotspots described in the Log4j documentation.

Feature	Description	Class	Rank	Type
Loggers	Log the messages of several levels	Category	1	TEMPLATE
		Logger	9	HOOK
		Level	4	HOOK
Appenders	Allows logging to multiple destinations	ConsoleAppender	11	TEMPLATE
		FileAppender	16	TEMPLATE
Layouts	Helps to format the logging request	PatternLayout	5	TEMPLATE
		SimpleLayout	13	TEMPLATE
Configurators	Helps to configure Log4j	BasicConfigurator	2	TEMPLATE
		PropertyConfigurator	3	TEMPLATE
		DOMConfigurator	7	TEMPLATE
Loaders	Helps to load resources	Loader	27	TEMPLATE
NDC	Nested diagnostic constant	NDC	12	TEMPLATE

Log4j provides several features such as Appenders and Layouts, and for each such feature Log4j provides several classes. For example, Log4j provides classes `ConsoleAppender` and `JDBCAppender` for the appender feature. Among those several classes provided for each feature, a few classes are much more often used than other classes. The features described in the documentation of Log4j are shown in Columns "Feature" and "Description" of Table 3.7. Column "Class" shows the commonly used classes for each feature. Each of these classes serves as starting points for using those features.

SpotWeb identified 56 classes as hotspots in Log4j for the HT percentage of 15%, and these classes captured all 12 starting points described in the documentation resulting in a recall of 100%. In constrast, the precision is 21.42%. Column "Rank" of Table 3.7 presents the rank of each documented class among the total number of hotspots detected by SpotWeb. Column "Type" shows whether the detected hotspot is a `TEMPLATE` or a `HOOK`. The table also shows that except the `Loader` class, all other 11 hotspot classes are ranked among the top 16 classes of the total 56 classes. Therefore, a user who plans to reuse classes of Log4j can refer to the first 16 classes suggested by SpotWeb to identify where to start reusing the library.

We used the cookbook provided with the JUnit library to verify the detected hotspots. Hotspots detected in the JUnit library are shown in Figure 3.6. SpotWeb identified 5 out of 6 hotspot classes described in the cookbook resulting in a recall of 83.33% and precision 35.71%.

We next describe our empirical analysis with several HT values and describe why we use HT of 15% to identify hotspots. Figure 3.12 shows the results with three subjects and several threshold values ranging from 5% to 25%. With the increase in the threshold value, the precision decreases and the recall increases. Although decrease in the precision is common with increase in the threshold value, the figure shows that the precision decreases rapidly with increase in the threshold value. This phenomenon shows that the real hotspots of these libraries are beyond those hotspots calculated based on the documentation, since the documentation of existing

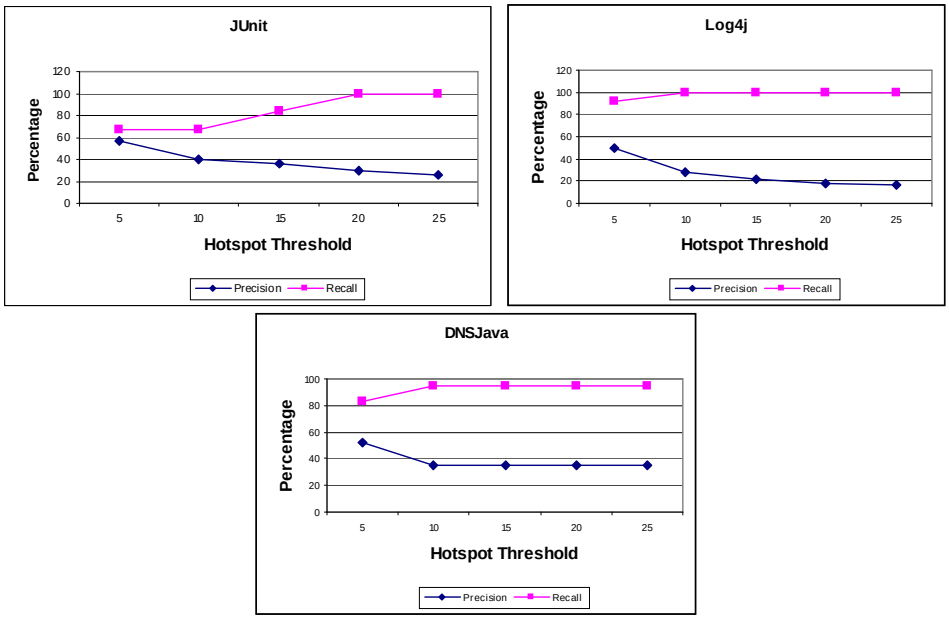


Figure 3.12: Precision and Recall for five HT values with JUnit, Log4j, and DNSJava.

libraries is often outdated. This evaluation shows the necessity of an approach such as SpotWeb that can automatically infer hotspots from the applications that are already reusing the API classes and methods of the input library.

In our implementation, we used HT as 15% as this threshold value has high recall with reasonable precision. The primary reason for inclining toward a high recall with low precision is that the library users do not miss any hotspot classes. Furthermore, our ranking mechanism helps to give higher priority to those hotspot classes that are described in the documentation compared to the other classes as shown in Table 3.7.

Comparison with Viljamaa's approach. We next compare the results of SpotWeb with the results of a previous related approach by Viljamaa [64] for the JUnit library. Their approach also recovers the hotspots of a library by using the source code of the library and a set of available example applications. Their approach uses concept analysis [47] for recovering hotspots of the library. Since both approaches target a similar problem, we compared the results of SpotWeb for the JUnit library with the results of their approach. Viljamaa's approach detected a total of 7 classes. SpotWeb detected 5 classes among these 7 classes. The 2 missing classes are `ExceptionTestCase` and `ActiveTestSuite`. However, both these classes are not described in the available documentation of JUnit. SpotWeb is not able to detect these classes because of low support among gathered code examples. SpotWeb detected 4 new classes and several new methods that are not detected by Viljamaa's approach and are described in the documentation. For example, classes of JUnit such as `Assert`, `TestResult`, and `TestFailure` (shown in Figure 3.6) are often used and only detected by SpotWeb. Similarly, for the `Testcase` class, the `tearDown`

method is also often used along with the `setUp` method. Viljamaa's approach detected only the `setUp` method, whereas SpotWeb detected both `setUp` and `tearDown` methods as hotspot methods. These results show that SpotWeb can perform better than Viljamaa's approach. Furthermore, Viljamaa's approach requires the users to have some initial knowledge of the structure and hotspots of the input library that is being analyzed. In contrast, SpotWeb does not require the users to have any knowledge regarding the input library.

3.4 Related Work

The problems faced by programmers in reusing existing libraries are addressed by various existing approaches. Earlier research in this area mainly focused on identifying related samples by matching keywords [85] or comments [141]. However, solutions suggested based on these approaches often cannot effectively help programmers in reusing the existing code examples. We next discuss the existing work based on code examples and closely related to our PARSEWeb and SpotWeb approaches.

Suggesting Method Sequences. Mandelin et al. developed Prospector [80], an approach that accepts queries in the form of a pair (T_{in}, T_{out}), where T_{in} and T_{out} are class types, and suggests solutions by traversing all paths among types of API signatures between T_{in} and T_{out}. A solution to the query is a synthesized code sample that takes an input object of type T_{in} and returns an output object of type T_{out}. Their approach uses API signatures for suggesting solutions to the given query. Since API signatures are used for addressing the query, Prospector returns many irrelevant examples, as shown in our evaluation. PARSEWeb is different from Prospector, since Prospector uses API signatures, whereas PARSEWeb uses code samples for solving the given query. This feature helps PARSEWeb in identifying more relevant code samples by giving higher preference to code samples that are often used.

Strathcona developed by Holmes and Murphy [56] maintains an example repository and compares the context of the code under development with the code samples in the example repository, and recommends relevant examples. Both PARSEWeb and Strathcona suggest relevant code samples, however, Strathcona is based on heuristics that are generic and are not tuned for addressing the described problem. This limitation often results in irrelevant examples as shown in our evaluation. XSnippet developed by Sahavechaphan and Claypool [108] also tries to address the described problem by suggesting relevant code snippets for the object instantiation task at hand. These suggested code snippets are selected from a sample repository. The major issues with both Strathcona and XSnippet is the availability of limited code samples stored in the repository. Instead, PARSEWeb developed based on our WebMiner framework, does not suffer from these issues.

Coogle developed by Sager et al. [107] extends the concept of similarity measures (often used to find similar documents for a given query) to source code repositories. Their approach detects similar Java classes in software projects using tree similarity algorithms. Both PARSEWeb and Coogle use ASTs for parsing Java code, but a structural similarity at the Java class level may not effectively address the described problem. Similar to Strathcona, Coogle may also result in many irrelevant examples.

Another related tool MAPO, developed by Xie and Pei [139], generates frequent usage patterns of an API by extracting and mining code samples from open source repositories through CSEs. Both MAPO and PARSEWeb exploit CSEs for gathering relevant code samples, but MAPO cannot solve queries of the form "*Source* $\rightarrow$ *Destination*". Programmers need to know the API to be used for using MAPO to identify usage patterns of that API. Moreover, PARSEWeb is more effective than MAPO as we consider control-flow information while generating MISs for the given query, whereas MAPO does not consider the control-flow information.

Hotspots and Coldspots. An approach by Viljamaa [64] recovers the hotspots of a library by using the source code of the input library and a set of available example applications. Their approach uses concept analysis [47] for uncovering hotspots of the library. One major problem with their approach is that applying concept analysis to the entire input source code can result in a huge pattern that is not useful in practice. To address the preceding problem, their approach suggests to select only those program elements that are relevant to the hotspot h at hand. Therefore, their approach requires the users to have some initial knowledge of the structure and hotspots of the library under analysis. In contrast, SpotWeb uses simple statistical analysis and can handle an entire input library. Furthermore, SpotWeb does not require the users to have any knowledge of the input library. Moreover, SpotWeb performs better than their approach as shown in our evaluation.

Mendonca et al. [87] proposed an approach to assist library instantiation and to understand the intricate details surrounding the library design. However, their approach requires library developers to manually specify the library design in a specific process language, called Reuse Definition Language, proposed by their approach.

Holmes and Walker [57] proposed an approach that quantitatively determines how existing APIs are used. Their approach gathers a few applications that already reuse those existing APIs and computes metrics to detect how existing APIs are used by those applications. SpotWeb differs from their approach in three main aspects. First, their approach expects the library users to have knowledge of the APIs of the library. Therefore, their approach is mainly useful to users who are already familiar with those library APIs. Second, their approach presents only the number of times that the APIs are reused. Third, their approach computes metrics from a limited data scope. In contrast, SpotWeb does not require the users to have the knowledge

of APIs of the input library and presents information in a more comprehensive form through templates and hooks.

Baxter et al. [14] proposed an approach to discover the structure of Java programs and the way that the classes relate to each other through inheritance and composition. Their study is useful for the library developers who can evaluate the structural features of their own programming practice and optimize their performance. In contrast, SpotWeb is useful for the library users in effectively reusing the APIs of the library.

3.5 Chapter Summary

In this chapter, we presented two approaches, called PARSEWeb and SpotWeb, that are developed based on our WebMiner framework. PARSEWeb accepts queries of the form "*Source* $\rightarrow$ *Destination*" as input and suggests method sequences that accept the *Source* object type as input and result in the *Destination* object type. On the other hand, SpotWeb assists programmers in reusing API classes and methods of a library under analysis by detecting hotspots and coldspots of the library. Since both these approaches are based on our WebMiner framework that collects code examples on demand, these two approaches are independent of any specific set of libraries. In our evaluations, we showed that both PARSEWeb and SpotWeb perform better than their related approaches, respectively.

Chapter 4

Improving Software Quality via Static Verification

4.1 Introduction

Programming rules or API specifications serve as a basis for applying static or dynamic verification tools to detect rule violations as software defects and improve software quality. However, in practice, these programming rules are often not well documented for APIs due to various factors such as hard project delivery deadlines and limited resources in the software development process [73]. To tackle the issue of lacking documented programming rules, various approaches have been developed in the past decade to mine programming rules from program executions [8,39,140], individual versions [3,4,24,38,75,101,112,131], or version histories [78,135] of program source code. A common methodology adopted by these approaches is to mine common patterns (e.g., frequent occurrences of pairs or sequences of API calls) across a sufficiently large number of data points (e.g., code examples), also referred to as *mining software engineering data*. These common patterns often reflect programming rules that should be obeyed when programmers write code using API calls involved in these rules. Then, these approaches use static defect-detection techniques that accept mined patterns as input and detect pattern violations as potential defects in applications under analysis.

Although these existing approaches are shown to detect a few real defects in applications under analysis, these approaches often result in a large number of false negatives and false positives. Here, false negatives represent the defects that exist in applications under analysis and are not detected by those approaches. On the other hand, false positives indicate those violations that do not represent real defects. In our empirical studies, we identify that the primary reason for such false negatives and false positives is that these approaches attempt to directly reuse existing off-the-shelf mining algorithms such as frequent itemset miner [22]. Such

direct reuse often constrains the patterns mined by these existing approaches, leading to both false negatives and false positives.

In this chapter, we address these issues by proposing two approaches, called *CAR-Miner* [122] and Alattin [121,123]. Both these approaches are developed based on our WebMiner framework, and focus on detecting two specific types of defects. In particular, CAR-Miner mines exception-handling rules, which describe expected behavior when exceptions occur during execution of a program. CAR-Miner mines exception-handling rules of the example form (referred to as sequence association rules) "$(FC_c^1...FC_c^n) \wedge FC_a \Rightarrow (FC_e^1...FC_e^m)$". This example rule describes that the method call FC_a should be followed by a sequence of method calls $(FC_e^1...FC_e^m)$ when FC_a is preceded by a sequence of method calls $(FC_c^1...FC_c^n)$. Such form of rules is required to characterize common exception-handling rules and defect exception-handling related defects.

On the other hand, *Alattin* mines patterns in four pattern formats: conjunctive (*And* or $\wedge$), disjunctive (*Or* or $\vee$), exclusive-disjunctive (*Xor* or $\oplus$), and combinations of these patterns (referred to as *Combo* patterns). We use *Alternative Patterns* to collectively refer to patterns of all four formats and refer to individual patterns such as P_1 and P_2 in $P_1 \wedge P_2$ as alternatives. Alattin uses these mined patterns to detect neglected conditions such as missing condition checks before invoking an API or after invoking an API.

Unlike existing approaches, our two approaches include two new mining algorithms, respectively, that mine patterns that satisfy the unique requirements of defect types under analysis. In our evaluations, we show that both CAR-Miner and Alattin perform better than related approaches (that use off-the-shelf mining algorithms) in reducing both false negatives and false positives.

4.2 CAR-Miner: Detecting Exception-Handling Defects

4.2.1 Motivation

Programming languages such as Java and C++ provide exception-handling constructs such as `try-catch` to handle exception conditions that arise during program execution. Under these exception conditions, program executions follow paths different from normal execution paths; these additional paths are referred to as *exception* paths. Applications developed based on these programming languages are expected to handle these exception conditions and take necessary recovery actions. For example, when an application reuses resources such as files or database connections, the application should release the resources after the usage in all paths including *exception* paths. Failing to release the resources can not only cause performance degradation, but can also lead to critical issues. For example, if a database lock acquired by a process is not released, any other process trying to acquire the same lock hangs till the database releases

Scenario 1

```
1.1: ...
1.2: OracleDataSource ods = null; Session session = null;
     Connection conn = null; Statement statement = null;
1.3: logger.debug("Starting update");
1.4: try {
1.5:     ods = new OracleDataSource();
1.6:     ods.setURL("jdbc:oracle:thin:scott/tiger@192.168.1.2:1521:catfish");
1.7:     conn = ods.getConnection();
1.8:     statement = conn.createStatement();
1.9:     statement.executeUpdate("DELETE FROM table1");
1.10:    connection.commit(); }
1.11:    catch (SQLException se) {
1.12:            if (conn != null) { conn.rollback(); }
1.13:            logger.error("Exception occurred"); }
1.14: finally {
1.15:    if(statement != null) statement.close();
1.16:    if(conn != null) conn.close();
1.17:    if(ods != null) ods.close();
1.18: }
```

Scenario 2

```
2.1: Connection conn = null;
2.2: Statement stmt = null;
2.3: BufferedWriter bw = null; FileWriter fw = null;
2.3: try {
2.4:     fw = new FileWriter("output.txt");
2.5:     bw = BufferedWriter(fw);
2.6:     conn = DriverManager.getConnection("jdbc:pl:db", "ps", "ps");
2.7:     Statement stmt = conn.createStatement();
2.8:     ResultSet res = stmt.executeQuery("SELECT Path FROM Files");
2.9:     while (res.next()) {
2.10:            bw.write(res.getString(1));
2.11:    }
2.12:    res.close();
2.13: } catch(IOException ex) { logger.error("IOException occurred");
2.14: } finally {
2.15:    if(stmt != null) stmt.close();
2.16:    if(conn != null) conn.close();
2.17:    if (bw != null) bw.close();
2.18: }
```

Figure 4.1: Two example scenarios from real applications.

the lock after timeout. A case study [132] conducted on a real application demonstrates the necessity of releasing resources in exception paths for improving reliability and performance. The case study found that there was a surprising improvement of 17% in performance of the application after correctly releasing resources in the presence of exceptions.

Software verification can be challenging for exception cases as verification techniques require specifications that describe expected behaviors when exceptions occur. These specifications are often not available in practice [73]. To address this issue, association rules of the form "$FC_a \Rightarrow FC_e$" are mined as specifications [133], where both FC_a and FC_e are function calls that share the same receiver object. These specifications are used to verify whether the function call FC_a is followed by the function call FC_e in all exception paths. However, simple association rules of this form are often not sufficient to characterize common exception-handling rules. The rationale is that there are various scenarios where FC_a is not necessarily followed by FC_e when exceptions are raised by FC_a, although both function calls share the same receiver object.

We next present an example using Scenarios 1 and 2 (extracted from real applications) shown in Figure 4.1. Scenario 1 attempts to modify contents of a database through the function call `Statement.executeUpdate` (Line 1.9), whereas Scenario 2 attempts to read contents of a database through the function call `Statement.executeQuery` (Line 2.8). Consider a simple specification in the form of an association rule "`Connection creation` $\Rightarrow$ `Connection rollback`". This rule describes that a `rollback` function call should appear in exception paths whenever an object of `Connection` is created. Although a `Connection` object is created in both scenarios, this rule applies only to Scenario 1 and does not apply to Scenario 2. The primary reason is that the `rollback` function call should be invoked *only* when there are any changes made to the database. This example shows that simple association rules of the form "$FC_a \Rightarrow FC_e$" are often insufficient to characterize exception-handling rules.

The insufficiency of simple association rules calls for more general association rules, hereby referred to as *sequence association rules*, of the form "$(FC_c^1 ... FC_c^n) \wedge FC_a \Rightarrow (FC_e^1 ... FC_e^m)$".

This sequence association rule describes that function call FC_a should be followed by function-call sequence $FC_e^1...FC_e^m$ in exception paths only when preceded by function-call sequence $FC_c^1...FC_c^n$. Using this sequence association rule, the preceding example can be expressed as "$(FC_c^1 FC_c^2) \wedge FC_a \Rightarrow (FC_e^1)$", where

FC_c^1 : `OracleDataSource.getConnection`

FC_c^2 : `Connection.createStatement`

FC_a : `Statement.executeUpdate`

FC_e^1 : `Connection.rollback`

This sequence association rule applies to Scenario 1 and does not apply to Scenario 2 due to the presence of FC_a: `Statement.executeUpdate`. The key aspects to be noted in this rule are: (1) `Statement.executeUpdate` is the primary reason to have `Connection.rollback` in an exception path and (2) the receiver object of `Statement.executeUpdate` is dependent on the receiver object of `Connection.rollback` through the function-call sequence defined by $FC_c^1 FC_c^2$.

Our sequence association rules are a super set of simple association rules. For example, sequence association rules are the same as simple association rules when the sequence $FC_c^1...FC_c^n$ is empty. To the best of our knowledge, existing association rule mining techniques [6] cannot be directly applied to mine these sequence association rules. Therefore, to bridge the gap, we develop a new mining algorithm by adapting the frequent closed subsequence mining technique [130].

We further develop a novel approach, called CAR-Miner, that incorporates our new mining algorithm for the problem of detecting exception-handling rules in the form of sequence association rules by analyzing source code. Apart from mining sequence association rules, CAR-Miner, developed based on our WebMiner framework addresses another challenge that is often faced by existing approaches [24,75,133], which mine rules from a limited data scope, i.e., from only a few example applications. Therefore, these approaches may not be able to mine rules that do not have enough supporting samples in those example applications, and hence the related defects remain undetected by these approaches. We show the usefulness of mined exception-handling rules by applying these rules on five applications to detect violations.

CAR-Miner makes the following major contributions:

- A general mining algorithm to mine sequence association rules of the form "$(FC_c^1...FC_c^n) \wedge FC_a \Rightarrow (FC_e^1...FC_e^m)$". Our new mining algorithm takes a step forward in the direction of developing new mining algorithms to address unique requirements in mining software engineering data, beyond being limited by existing off-the-shelf mining algorithms.

- An approach that incorporates the general mining algorithm to mine exception-handling rules that describe expected behavior when exceptions occur during program execution.

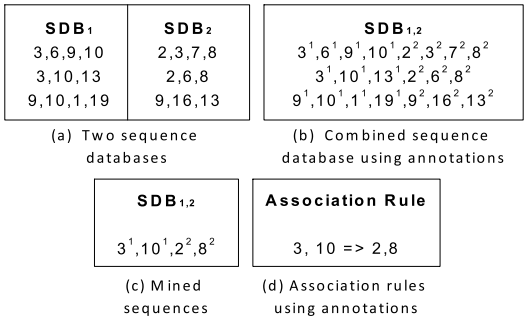


Figure 4.2: Illustrative examples of general algorithm.

- A technique for constructing a precise Exception-Flow Graph (EFG), which is an extended form of a Control-Flow Graph (CFG), that includes only those exception paths that can potentially occur during program execution.

- Two evaluations to show the effectiveness of our CAR-Miner approach. (1) CAR-Miner detects 294 real exception-handling rules in five different applications including 285 KLOC. (2) The top 50 exception-handling rules (top 10 real rules of each application) are used to detect a total of 160 real defects in these five applications, where 87 defects are new, not being detected by a previous related approach [133].

4.2.2 Sequence Association Rule Mining Algorithm

We next present a formal definition of general association rules and then describe sequence association rules required for characterizing exception-handling rules. Although we present our algorithm from the point-of-view of mining exception-handling rules, the algorithm is general and can be applied to other practical problems that fall into our problem domain.

Problem Domain

Let $F = \{FC_1, FC_2, ..., FC_k\}$ be the set of all possible distinct items. Let $I = \{FC_{i1}, FC_{i2}, ..., FC_{im}\}$ and $J = \{FC_{j1}, FC_{j2}, ..., FC_{jn}\}$ be two sets of items, where $I \subseteq F$ and $J \subseteq F$. Consider a sequence database as a set of tuples (sid, S_i, S_j), where sid is a sequence id, S_i is a sequence of items belonging to I, and S_j is a sequence of items belonging to J. In essence, S_i and S_j belong to two sequence databases, say SDB_1 and SDB_2, denoted as $S_i \in SDB_1$ and $S_j \in SDB_2$, respectively, and there is a **one-to-one** mapping between the two sequence databases. We define an association rule between sets of sequences as $X \Rightarrow Y$, where both X and Y are subsequences of $S_i \in SDB_1$ and $S_j \in SDB_2$, respectively. A sequence $\alpha = \langle a_1 a_2 ... a_p \rangle$ (where each a_s is an item) is defined as a subsequence of another sequence $\beta = \langle b_1 b_2 ... b_q \rangle$, denoted as $\alpha \sqsubseteq \beta$, if there exist integers $1 \leq j_1 < j_2 < ... < j_p \leq q$ such that $a_1 = b_{j1}$, $a_2 = b_{j2}$,..., $a_p = b_{jq}$.

General Algorithm

To the best of our knowledge, there are no existing mining techniques that can mine from sets of sequences such as SDB_1 and SDB_2 with resulting association rules as $X \Rightarrow Y$, where $X \sqsubseteq S_i \in SDB_1$ and $Y \sqsubseteq S_j \in SDB_2$. We combine both sequence databases in a novel way using annotations to build a single sequence database. These annotations help in deriving association rules in later stages. For example, consider two sequence databases shown in Figure 4.2a. Figure 4.2b shows a single sequence database using annotations combined from the two sequence databases. We next mine frequent subsequences from the combined database, denoted as $SDB_{1,2}$, using the frequent closed subsequence mining technique [130].

The frequent subsequence mining technique accepts a database of sequences such as $SDB_{1,2}$ and a minimum support threshold *min_sup*, and returns subsequences that appear at least *min_sup* times in the sequence database. Given a sequence s, it is considered as frequent if its support $sup(s) \geq min\_sup$. In our context, we are interested in frequent closed subsequences. A sequence s is a frequent closed sequence, if s is frequent and no proper super sequence of s is frequent. Figure 4.2c shows an example closed frequent subsequence from the combined sequence database. Since sequence mining preserves temporal order among items, we scan each closed frequent subsequence and transform the subsequence into an association rule of the form "$X \Rightarrow Y$" based on annotations (as shown in Figure 4.2d). We compute confidence values for each association rule using the formula as shown below:

Confidence ($X \Rightarrow Y$) = Support (X Y) / Support (X)

Although we explain our algorithm using two sequence databases SDB_1 and SDB_2, our algorithm can be applied to multiple sequence databases as well. These multiple sequence databases can also be combined into a single sequence database using the similar mechanism illustrated in Figure 4.2.

Sequence Association Rules

In our current approach, our target is to mine exception-handling rules in the form of association rules. Therefore, we collect two sequence databases for each function call FC_a: a normal function-call-sequence (NFCS) database and an exception function-call-sequence (EFCS) database. We apply our mining algorithm to generate sequence association rules of the form $FC_c^1...FC_c^n \Rightarrow FC_e^1...FC_e^m$, where $FC_c^1...FC_c^n \sqsubseteq S_i \in$ NFCS and $FC_e^1...FC_e^m \sqsubseteq S_j \in$ EFCS. Such an association rule describes that FC_a should be followed by the function-call-sequence $FC_e^1...FC_e^m$ in exception paths, when preceded by the function-call-sequence $FC_c^1...FC_c^n$. Since this association rule is specific to the function call FC_a, we append FC_a to the rule as $(FC_c^1...FC_c^n) \wedge FC_a \Rightarrow (FC_e^1...FC_e^m)$.

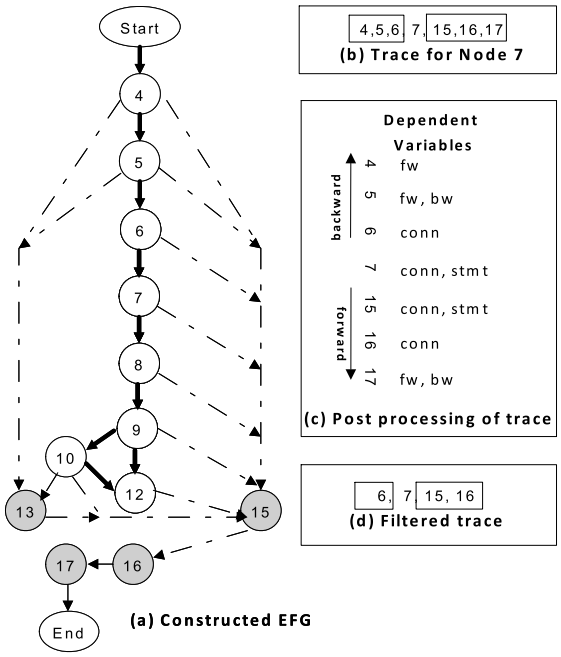


Figure 4.3: Illustrative examples of CAR-Miner approach.

4.2.3 Approach

CAR-Miner, developed based on our WebMiner framework, accepts an application under analysis and mines exception-handling rules for all function calls in the application. CAR-Miner next detects violations of the mined exception-handling rules. CAR-Miner includes all phases of our WebMiner framework. Initially, CAR-Miner parses the application to collect each function call, say FC_a, in the application from the call sites in the application. For example, CAR-Miner collects the function call `Statement.executeUpdate` as an FC_a from Line 1.9 in Scenario 1. We denote the set of all function calls as FCS. In the *search* phase, CAR-Miner collects code examples that reuse FC by interacting with a code search engine. In the *analyze* phase, CAR-Miner generates pattern candidates. In contrast to CFGs constructed by our WebMiner framework, CAR-Miner constructs Exception-Flow Graphs (EFG), which are an extended form of CFG. In the *mine* phase, CAR-Miner applies sequence association rule mining algorithm to generate programming rules that describe expected behavior when exceptions occur which invoking function calls such as FC. In the *apply* phase, CAR-Miner detects violations in the application under analysis as deviations from mined rules. We next present the details of all phases, except the *search* phase, which is the same as our WebMiner framework.

Analyze Phase

Exception-Flow Graph Construction. In the *analyze* phase, CAR-Miner analyzes code examples collected in the *search* phase and generates pattern candidates. Initially, CAR-Miner

constructs Exception-Flow Graphs (EFG), which are an extended form of CFGs. An EFG provides a graphical representation of all paths that might be traversed during the execution of a program, including exception paths. Construction of an EFG is non-trivial due to the existence of additional paths that transfer control to exception-handling blocks defined in the form of `catch` or `finally` in Java. We develop an algorithm inspired by Sinha and Harrold [113] for constructing EFGs with additional paths that describe exception conditions. Figure 4.3a shows the constructed EFG for Scenario 2, where each node is denoted with the corresponding line number of Scenario 2 in Figure 4.1.

Initially, CAR-Miner builds a CFG that represents flow of control during normal execution and augment the constructed CFG with additional edges that represent flow of control after exceptions occur. We refer to these additional edges as *exception* edges and all other edges as *normal* edges. In the figure, *normal* and *exception* edges are shown in solid and dotted lines, respectively. For example, an exception edge is added from Node 5 to Node 13 as the program can follow this path when `IOException` occurs while creating a `BufferedWriter` object. Since code inside a `catch` or a `finally` block gets executed after exceptions occur, we consider edges between the statements within `catch` and `finally` blocks also as exception edges. We show nodes related to function calls in normal paths such as those in a `try` block in white and function calls in exception paths such as those in a `catch` block in grey. Although function calls in a `finally` block belong to both normal and exception paths, we consider these paths as exception paths and show the associated nodes in grey. For simplicity, we ignore the control flow inside exception blocks.

In the constructed EFG, there is an exception edge from Node 5 to Node 13, however, there is no exception edge from Node 6 to Node 13. The reason is that Node 13 handles a checked exception `IOException`, which is never raised by function call `DriverManager.getConnection` of Node 6. Therefore, we prevent such infeasible control flow through a sound static analysis tool, called Jex [104]. Jex analyzes source code statically and provides possible exceptions raised by each function call. For example, Jex provides that `IOException` can be raised by `BufferedWriter.Constructor` but not `DriverManager.getConnection`. While adding *exception* edges, we add only those edges from a function call to a `catch` block where the exception handled by the `catch` block belongs to the set of possible exceptions thrown by the function call. This additional check helps reduce potential false positives by preventing infeasible exception paths. If the `catch` block handles `Exception` (the super class of all exception types), we add exception edges from each function call to the `catch` block. We consider a `finally` block as similar to a `catch` block that handles `Exception`, and add exception edges from each function call to the `finally` block.

Since collected code samples are partial, we use intra-procedural analysis for constructing EFGs. Furthermore, before constructing an EFG for a code sample, we also check whether the

code sample includes any $FC_a \in FCS$. If the code sample does not include any FC_a, we skip the EFG construction for that code sample.

Static Trace Generation. We next capture static traces that include actions that should be taken when exceptions occur while executing function calls such as $FC_a \in FCS$. For example, consider the FC_a "`Connection.createStatement`" and its corresponding Node 7 in the EFG. A trace generated for this node is shown in Figure 4.3b. The trace includes three sections: *normal function-call sequence* ($FC_c^1...FC_c^n$), FC_a, *exception function-call sequence* ($FC_e^1...FC_e^m$).

The $FC_c^1...FC_c^n$ sequence starts from the beginning of the body of the enclosing function (i.e., caller) of the FC_a function call to the call site of FC_a. The $FC_e^1...FC_e^m$ sequence includes the longest exception path that starts from the call site of FC_a and terminates either at the end of the enclosing function body or at a node in EFG whose outgoing edges are all normal edges. We generate such traces from code samples and input application for each $FC_a \in FCS$.

Trace Post-Processing. We next identify function calls in $FC_c^1...FC_c^n$ or $FC_e^1...FC_e^m$ that are not related to FC_a through data-dependency, and remove such function calls from each trace. Failing to remove such unrelated function calls can result in many false positives due to frequent occurrences of unrelated function calls as shown in the evaluation of PR-Miner [75]. For example, in the trace shown in Figure 4.3b, function calls in the normal function-call sequence related to Nodes 4 and 5 are unrelated to the FC_a of Node 7. Similarly, Node 17 in the exception function-call sequence is also unrelated to FC_a.

Figure 4.3c shows an example of our data-dependency analysis. Initially, we generate two kinds of relationships: var dependency of a variable and function association of a function call. The var dependency of a variable represents the set of variables on which a given variable is dependent upon. Similarly, a function association of a function call represents the set of variables on which a function call is associated with.

First, we compute the var-dependency relationship information from assignment statements. For example, in Scenario 2, we identify that the variable `res` is dependent on the variable `stmt` from Line 2.8 and is transitively dependent on `conn` as `stmt` is dependent on `conn` from Line 2.7. We compute the function-association relationship based on the var-dependency relationship. In particular, we identify that a function call is associated with all its variables including the receiver, arguments, and the return variable, and their transitively dependent variables. For example, applying the preceding analysis to the function call of Node 7, we identify that the associated variables are `conn` and `stmt`.

We use variables associated with each function call to identify function calls in the normal function-call sequence $FC_c^1...FC_c^n$ or the exception function-call sequence $FC_e^1...FC_e^m$ that are not related to FC_a. Starting from FC_a, we perform a backward traversal of the trace to

filter out function calls in $FC_c^1...FC_c^n$ and a forward traversal to filter out function calls in $FC_e^1...FC_e^m$. Assume that variables associated with FC_a are $\{V_a^1, V_a^2,\ldots, V_a^s\}$. Assume that variables associated with a function call, say FC_{ce}^k, in the normal or exception function-call sequence are $\{V_{ce}^1, V_{ce}^2,\ldots, V_{ce}^t\}$.

In each traversal, we compute an intersection of associated variable sets of FC_a and FC_{ce}^k. If the intersection $\{V_a^1, V_a^2,\ldots, V_a^s\} \cap \{V_{ce}^1, V_{ce}^2,\ldots, V_{ce}^t\} \neq \phi$, we keep the FC_{ce}^k function call (either in the normal or exception function- call sequence) in the trace; otherwise, we filter out the FC_{ce}^k function call from the trace. The rationale behind our analysis is that if the intersection is a non-empty set, it indicates that the FC_a is directly or indirectly related to the FC_{ce}^k function call. For example, the intersection of associated variables for Nodes 6 and 7 is non-empty. In contrast, the intersection of associated variables for Nodes 5 and 7 is empty. Therefore, we keep Node 6 in the trace and filter out Node 5 during backward traversal. Similarly, during forward traversal, we ignore Node 17 since the intersection is an empty set. The resulting trace of "4,5,6,7,15,16,17" is "6,7,15,16", where

```
6 : DriverManager.getConnection
7 : Connection.createStatement
15 : Statement.close
16 : Connection.close
```

Mine Phase

We apply our new mining algorithm described in Section 4.2.2 on the set of static traces collected for each FC_a. These static traces serve pattern candidates for our mining algorithm. We apply mining on the traces of each FC_a individually. The reason is that if we apply mining on all traces together, rules related to a FC_a with only a small number of traces can be missed due to rules related to other FC_a with a large number of traces.

In the phase of static trace mining, we first transform traces suitable for our mining algorithm. More specifically, as each trace includes a normal function-call sequence and an exception function-call sequence, we build two sequence databases with normal and exception function-call sequences, respectively, from all the traces of a FC_a function call.

We next apply our mining algorithm that initially annotates corresponding normal and exception function-call sequences and combines the annotated sequences into a single call sequence. The mining algorithm produces sequence association rules of the form $FC_c^1...FC_c^n \Rightarrow FC_e^1...FC_e^m$. As this sequence association rule is specific to FC_a, we add FC_a to the rule as $(FC_c^1...FC_c^n) \wedge FC_a \Rightarrow (FC_e^1...FC_e^m)$. The preceding sequence association rule describes that the function call FC_a should be followed by $FC_e^1...FC_e^m$ in exception paths only when preceded by $FC_c^1...FC_c^n$ in normal paths. In our approach, we use the frequent closed subsequence mining tool, called BIDE, developed by Wang and Han [130]. We used the *min_sup* value as 0.4,

which is set based on our initial empirical experience. We repeat the preceding process for each FC_a and rank all final sequence association rules based on their support values assigned by the frequent subsequence miner.

Apply Phase

To show the usefulness of our mined exception-handling rules, we apply these rules on the application under analysis to detect violations. Initially, from each call site of FC_a in the application, we extract the normal function-call sequence, say $C_c^1 C_c^2 \ldots C_c^a$, from the beginning of the body of enclosing function of FC_a to the call site of FC_a. If $FC_c^1 \ldots FC_c^n \sqsubseteq C_c^1 C_c^2 \ldots C_c^a$, then we extract the exception function-call sequence, say $C_e^1 C_e^2 \ldots C_e^b$, from the call site of FC_a to the end of the enclosing function body or to a node (in the EFG) whose outgoing edges are all normal edges. We do not report a violation if $FC_e^1 \ldots FC_e^m \sqsubseteq C_e^1 C_e^2 \ldots C_e^b$; otherwise, we report a violation in the application under analysis. We rank all detected violations based on a similar criterion used for ranking exception-handling rules.

4.2.4 Evaluation

We next describe the evaluation results of CAR-Miner with five real-world open source applications as subjects. We use the same subjects (and same versions) used for evaluating a related approach called WN-miner [133] for the ease of comparison with the data provided by the WN-miner developer. We used five out of eight subjects used in WN-miner since related versions of the remaining three subjects are not currently available. The detailed results of our evaluation are available at `http://research.csc.ncsu.edu/ase/projects/carminer/`. In our evaluations, we address the following research questions.

- RQ1: How high percentage of exception-handling rules mined by CAR-Miner represent real rules?
- RQ2: How high percentage of rule violations detected by CAR-Miner represent real defects?
- RQ3: How many new real rules and real defects are detected by CAR-Miner compared to an existing related WN-miner approach?
- RQ4: How many new defects that cannot be detected with simple association rules of the form "$FC_a \Rightarrow FC_e$" are detected by sequence association rules?

Table 4.1: Characteristics of subjects used in evaluating CAR-Miner.

Subject	Lines of code	Internal Info		External Info		# Code Examples	Time (in sec.)
		#Classes	#Functions	#Classes	#Functions		
Axion 1.0M2	24k	219	2405	58	217	47783 (7M)	1381
HsqlDB 1.7.1	30k	98	1179	80	264	78826 (26M)	2547
Hibernate 2.0 b4	39k	452	4321	174	883	88153 (27M)	1125
SableCC 2.18.2	22k	183	1551	21	76	47594 (15M)	1220
Ptolemy 3.0.2	170k	1505	9617	477	2595	70977 (21M)	1126

Subjects

Table 4.1 shows subjects and their versions used in our evaluations. Axion is a small and fast open source Java database engine and provides database management utilities for applications developed in Java. HsqlDB is one of the leading SQL database engines and is often used as a database in development environments. Hibernate is a large-scale open source application and is used as a middle ware for interacting with a database from Java applications. Hibernate allows Java programmers to develop persistent classes in an object-oriented idiom. SableCC is a parser generator that helps build compilers, interpreters, and other text parsers. Ptolemy is a software framework that studies modeling, simulation, and design of concurrent, real-time, embedded systems.

Column "Internal Info" shows the number of declared classes and functions of each application. Column "External Info" shows the number of external classes and their functions invoked by the application. Column "Code Examples" shows the number of code examples gathered by CAR-Miner to mine exception-handling rules. For example, CAR-Miner gathered 47783 code examples ($\approx$ 7 million LOC) from a code search engine for mining exception-handling rules of the Axion application. Column "Time" shows the amount of time taken by CAR-Miner in seconds for each application. The shown time includes the analysis time of the application and gathered code examples, and the time taken for detecting violations. The amount of processing time depends on the number of samples gathered for an application. All experiments were conducted on a machine with 3.0GHz Xeon processor and 4GB RAM.

RQ1: Mined Exception-Handling Rules

We next address the first question on whether the mined exception-handling rules represent real rules that can help detect defects in an application under analysis. Table 4.2 shows the classification of exception-handling rules mined by CAR-Miner. Column "Total" shows the total number of rules in each application. We classify these rules into three categories: real rules, usage patterns, and false positives. Real rules describe the behavior that must be satisfied while using function calls such as FC_a, whereas usage patterns suggest common ways of using FC_a. The violations of real rules and usage patterns can be defects and hints, respectively. A

Table 4.2: Classification of exception-handling rules.

Subject	#Total	Real Rules		Usage Patterns		False Positives	
		#	%	#	%	#	%
Axion	112	70	62.5	3	2.68	39	34.82
HsqlDB	127	89	70.08	3	2.36	35	27.56
Hibernate	121	86	71.07	1	0.82	34	28.09
SableCC	40	12	30	2	5	26	65
Ptolemy	94	37	39.36	5	5.32	52	55.32
AVERAGE			54.6		3.24		42.16

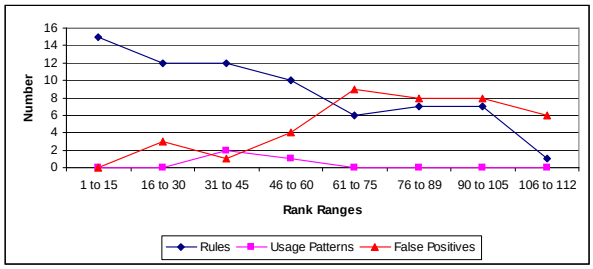


Figure 4.4: Distribution of classification categories with ranks for the Axion application.

hint, which was originally proposed by Wasylkowski et al. [131], helps increase readability and maintainability of source code of an application. We used the available on-line documentations, JML specifications[1], or the source code of the application for classifying mined exception-handling rules into these three categories. Our results show that real rules are 54.61% and false positives are 42.16%, averagely.

Although false positives are 42.16% on average among the total number of mined rules, our mining heuristics for ranking exception-handling rules help give higher priority to real rules than false positives. Figure 4.4 shows a detailed distribution of all extracted rules for the Axion application. In Figure 4.4, x-axis shows distribution of mined rules in different ranges (each range is of size 15) with respect to assigned ranks and y-axis shows the number of rules that are classified into the three categories for each range. The primary reason for selecting the Axion application is that the application is a medium-scale application that is amenable to a detailed analysis with reasonable effort. As shown in the figure, the number of false positives is quite low among the exception-handling rules ranked between 1 to 60. These results show the significance of our mining and ranking criteria. Our results in Table 4.2 also show that more exception-handling rules exist in applications such as Axion, HsqlDB, and Hibernate that deal with resources (such as databases or files) compared to other applications.

RQ2: Detected Violations

We next address the question on whether the detected violations represent real defects. Table 4.3 shows the violations detected in each application. Column "Total Violations" shows

[1] `http://www.eecs.ucf.edu/~leavens/JML/`

Table 4.3: Classification of detected violations.

Subject	#Total Violations	#Violations of first 10 rules	#Defects	#Hints	#FP
Axion 1.0M2	257	19	13	1	5
HsqlDB 1.7.1	394	62	51	0	10
Hibernate 2.0 b4	136	22	12	0	10
Sablecc 2.18.2	168	66	45	7	14
Ptolemy 3.0.2	665	95	39	1	55

Table 4.4: Status of detected defects in new versions of subject applications.

	# Defects	New Version	#Fixed	#Deleted	#Open
Axion 1.0M2	13	1.0M3	4	8	1
HsqlDB 1.7.1	51	1.8.0.9	2	9	40
Hibernate 2.0 b4	12	3.2.6	0	8	4
Sablecc 2.18.2	45	4-alpha.3	0	43	2
Ptolemy 3.0.2	39	3.0.2	0	0	39

the total number of violations detected in each application. The HsqlDB and Hibernate applications include test code as part of their source code. Since test code is often not written according to specifications, we excluded the violations detected in the test code of those applications from the results. Given a high number of violations in each application, we inspected the violations detected by the top 10 exception-handling rules and classified them into three categories: Defects, Hints, and False Positives.

Column "Violations of first 10 rules" shows the number of violations detected by the top ten exception-handling rules mined for each application. Column "Defects" shows the total number of violations that are identified as defects in each application. As we used the same versions (an earlier version than the latest version) used by the WN-miner approach for the ease of comparison, we verified whether the defects found by our approach are fixed, deleted, or still open in the latest version of each application. Column "New Version" of Table 4.4 shows the latest version used for our verification. The defect's sub-categories "Fixed" and "Open" indicate that the defects found by our approach in the earlier version are fixed or still open in the new version, respectively. We reported those open defects to respective developers for their confirmation. Sometimes, we find that the defective code such as function body with detected defects does not exist in the latest version. One reason could be the refactoring of such code, which can be considered as an indirect fix. We classified such defects as "Deleted" (shown in Table 4.4).

The results show that our CAR-Miner approach can detect real defects in the applications. The number of defects shown in Columns "Fixed" and "Deleted" provide further evidence that these defects detected by CAR-Miner are real since these defects are fixed directly or indirectly in newer versions of the applications. The initial response from the developers of HsqlDB is quite encouraging. The developers responded on the first ten defects that we reported, where

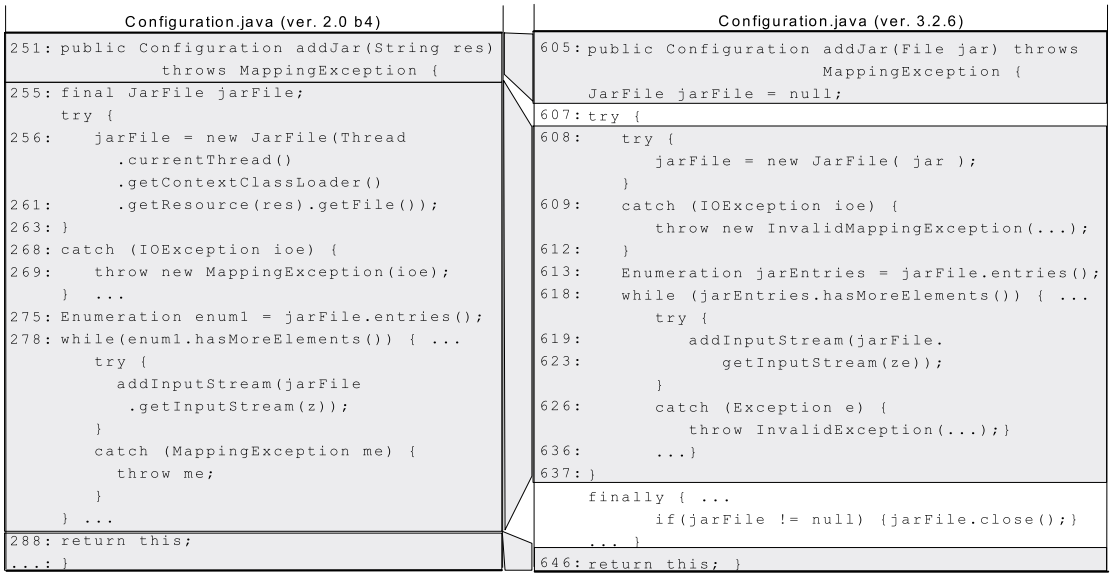


Figure 4.5: A fixed defect in the Hibernate application.

seven defects are *accepted* and only three defects are rejected. The bug reports for these ten defects are available in the HsqlDB Bug Tracker system[2] with IDs #1896449, #1896448, and #1896443[3]. Although the three rejected defects are violations of real rules, developers described that the violation-triggering conditions of these defects cannot be satisfied in the context of the HsqlDB application. For example, a rejected defect is a violation of real rule "`DatabaseMetaData.getPrimaryKeys` ⇒ `ResultSet.close`". The preceding rule describes that the `close` function call should be invoked on `ResultSet`, when `getPrimaryKeys` throws any exceptions. The response from the developers (Bug report ID: #1896448) for this defect is "*Although it can throw exceptions in general, it should not throw with HSQLDB. So it is fine.*", which describes that the violation-triggering condition cannot be satisfied in the context of HsqlDB.

We next show a real defect detected in the Hibernate application, and show how this defect is fixed in the latest version. Figure 4.5 shows a real defect in `Configuration.java` source file (Hibernate 2.0 b4). The defect violated the exception-handling rule "`JarFile.getInputStream` ⇒ `JarFile.close`", which indicates that when an exception occurs while executing the function `getInputStream`, the related `JarFile` resource should be closed. The figure also shows the fix done in the latest version (version 3.2.6) of the Hibernate application, where a new `finally` block is added with additional code for closing the `JarFile` resource. The described defect is not detected by the related approach WN-miner, as this exception-handling rule is extracted from gathered code examples and does not have supporting samples in the Hibernate application.

[2] `http://sourceforge.net/tracker/?group_id=23316&atid=378131`

[3] We reported multiple defects in the same source file as a single bug report.

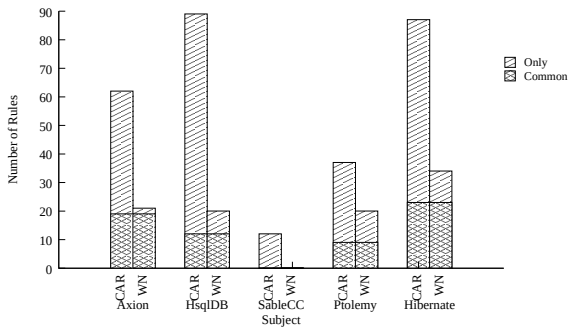


Figure 4.6: Comparison of real rules mined by CAR-Miner and WN-miner.

RQ3: Comparison with WN-miner

We next address the third question on whether our CAR-Miner approach performs better than the related WN-miner tool. As the WN-miner tool is not currently available, the WN-miner developer provided the mined specifications and static traces of their tool. We developed Perl scripts to detect violations of mined specifications in static traces as described by the WN-Miner developer [133]. We used the same criteria described in Section 4.2.4 for classifying rules and violations detected by their approach, respectively. We compared both mined exception-handling rules and detected violations.

Comparison of exception-handling rules. We next present the comparison results of exception-handling rules mined by both approaches. Figure 4.6 shows the results for the classification category "real rules" between WN-miner and CAR-Miner. For each subject and approach, the figure shows the total number of rules mined by each approach along with the number of common rules between the two approaches. For example, CAR-Miner detected a total of 70 rules for the Axion application. Among these 70 rules, 43 rules are newly detected by CAR-Miner and 27 rules are common between CAR-Miner and WN-miner. CAR-Miner failed to detect 2 real rules that were detected by WN-miner.

The primary reason for these two real rules not detected by CAR-Miner and detected by WN-miner is due to the *ranking* criterion used by WN-miner. WN-miner extracts rules "$FC_a \Rightarrow FC_e$" when FC_e appears at least once in exception-handling blocks such as `catch` and ranks those rules with respect to the number of times FC_e appears after FC_a among normal paths. As shown in their results, such a criterion can result in a high number of false positives such as "`Trace.trace` $\Rightarrow$ `Trace.printSystemOut`" in the HsqlDB application, where FC_e often appears after FC_a in normal paths and is used once in some `catch` block. CAR-Miner ignores such patterns due to their relatively low support among exception paths of FC_a.

The results show that CAR-Miner is able to detect most of the rules mined by WN-miner and also many new rules that are not detected by WN-miner. CAR-Miner performed better than WN-miner due to two factors: sequence association rules and increase in the data scope.

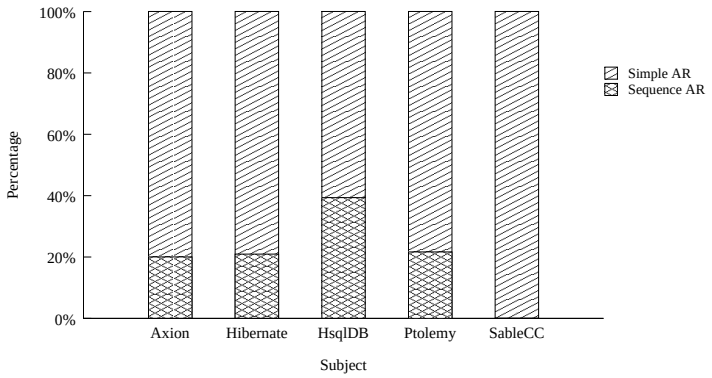


Figure 4.7: Percentage of sequence association rules.

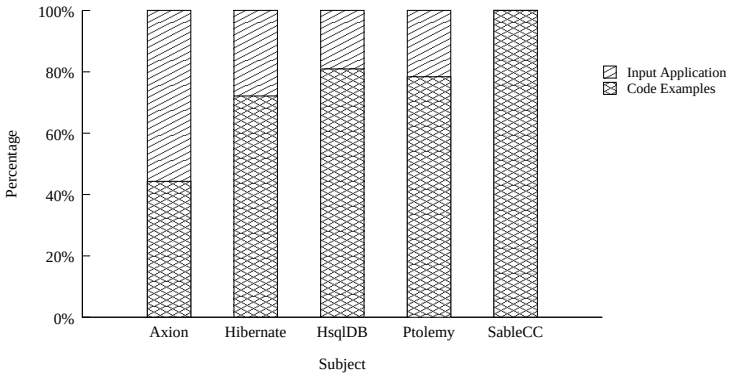


Figure 4.8: Percentage of rules mined only from code examples.

To further show the significance of these factors, we classified the real rules mined by CAR-Miner based on these two factors. Figure 4.7 shows the percentage of sequence association rules among all real rules. The results show that sequence association rules are 20.37% of all real rules on average mined for all applications.

Figure 4.8 shows the percentage of real rules that cannot be mined by analyzing only the application under analysis. For example, 44.28% of the real rules mined for the Axion application occur only from gathered code samples. Our results show that increase in the data scope to open source repositories helps detect new exception-handling rules that do not have sufficient supporting samples in the application. Furthermore, increase in the data scope also helps give higher priority to real rules than false positives.

Comparison of detected defects. We next present the number of real defects that were detected by CAR-Miner but not detected by WN-miner. To show that CAR-Miner can find new defects that were not detected by WN-miner, we identified the exception-handling rules that are mined only by CAR-Miner and not by WN-miner among top 10 shown in Table 4.3 and verified the defects detected by those rules. The results are shown in Table 4.5. Column "Total" shows the number of violations detected by the top 10 exception-handling rules. Column "Common" and "Only" show the number of defects commonly detected by CAR-Miner and

Table 4.5: Defects detected or missed by CAR-Miner.

Subject	# Defects			
	# Total	#Common	# Only	# Missed
Axion	13	0	13	1
HsqlDB	51	35	16	13
Hibernate	12	0	12	7
Sablecc	45	0	45	0
Ptolemy	39	38	1	11
TOTAL	160	73	87	32

Table 4.6: Defects detected by Sequence Association Rules.

	# Rules	# Violations	# Defects	# Hints	# False Positives
Axion	3	6	4	0	2
HsqlDB	6	14	8	0	6
Hibernate	4	10	8	0	2
Sablecc	0	0	0	0	0
Ptolemy	1	1	1	0	0

WN-miner, and defects that are detected by CAR-Miner only, respectively. Column "Missed" shows the number of defects detected by WN-miner only. The results show that CAR-Miner detected 87 new defects (among all applications) that were not detected by WN-miner. When inspecting all violations detected by CAR-Miner, we expect that the preceding number of new defects detected by CAR-Miner can be much higher. CAR-Miner missed 32 defects that were detected by WN-miner. These missed defects are due to the missing patterns as described in Section 4.2.4.

Significance of Sequence Association Rules

We next address the last research question on whether sequence association rules mined by CAR-Miner are helpful in detecting new defects that cannot be detected by simple association rules. Table 4.6 shows the number of sequence association rules that are used to detect real defects in all applications. The results show that these rules help detect 21 real defects among all applications.

We next describe a defect in the HsqlDB application to show the significance of sequence association rules, which cannot be mined by existing approaches such as WN-miner. The related code snippet from the `saveChanges` function of `ZaurusTableForm.java` is shown as below:

```
public boolean saveChanges()
{ ...
 try {
  PreparedStatement ps =
      cConn.prepareStatement(str);
  ps.clearParameters(); ...
```

```
    for (int j=0; j<primaryKeys.length; j++){
     ps.setObject(i + j + 1,
        resultRowPKs[aktRowNr][j]); }
    ps.executeUpdate();
   } catch (SQLException e) { ...
     return false;
   } ...
  }
```

CAR-Miner detected a defect in the preceding code example as the code example violated the exception-handling rule $FC_c^1 \wedge FC_a \Rightarrow FC_e^1$, where

FC_c^1 :`Connection.prepareStatement`
FC_a :`PreparedStatement.clearParameters`
FC_e^1 :`Connection.rollback`

The preceding rule describes that when an exception occurs after executing the function `clearParameters`, the `rollback` function should be invoked on the `Connection` object. Failing to invoke `rollback` can make the database state inconsistent. This result shows that sequence association rules are helpful in detecting new defects.

4.3 Alattin: Detecting Neglected Conditions

4.3.1 Motivation

In the preceding section, we presented CAR-Miner, which primarily focuses on mining programming rules that detect new exception-handling defects, thereby reducing false negatives. We next present another approach, called Alattin, developed based on our WebMiner framework, and focuses on reducing false positives among detected violations. As shown in empirical evaluations of existing approaches [4, 24, 38, 75, 131], majority of detected violations turn out to be false positives, making those approaches ineffective in practice.

To illustrate how detected violations turn out to be false positives, we use two code examples (shown in Figure 4.9) using the `next` method of the `Iterator` class. The `next` method throws `NoSuchElementException` when invoked on an `ArrayList` object without any elements. Programmers can avoid this exception by using either the condition check "P_1: `boolean-check` on `return` of `Iterator.hasNext` before `Iterator.next`" (shown in `printEntries1` from Example 1) or "P_2: `const-check` on `return` of `ArrayList.size` before `Iterator.next`" (shown in `printEntries2` from Example 2). In general, programmers use either P_1 or P_2 but not both, since using both P_1 and P_2 is redundant. Consider that a single pattern P_1 is mined from the data points. A static

```
Example 1:
00:String printEntries1(ArrayList<String> entries){
01: ...
02: Iterator it = entries.iterator();...
03: if (it.hasNext()) {
04:    String last = (String) it.next();... }
05:}

Example 2:
00:String printEntries2(ArrayList<String> entries){
01: ...
02: if (entries.size() > 0) {
03:    Iterator it = entries.iterator();...
04:    String last = (String) it.next();... }
05:}
```

Figure 4.9: Two code examples using the `next` method of the `Iterator` class.

defect-detection technique reports a violation in `printEntries2`, since the method does not satisfy P_1. However, the code example does not include any defect on using `Iterator.next`, since `printEntries2` satisfies P_2; therefore, the detected violation turns out to be a false positive.

In our empirical investigation of false positives generated by existing approaches [4,24,38,75, 131], we identify that a major reason for such a large number of false positives is that the focus of these existing approaches is to mine *single* patterns (such as P_1) or *conjunctive* patterns (such as $P_1 \wedge P_2$) (more details are presented in Section 4.3.5). The conjunctive pattern $P_1 \wedge P_2$ describes that both P_1 and P_2 often appear together among the data points (e.g., code examples). We identify that these single or conjunctive patterns alone cannot describe the *nearly complete* behavior among data points, resulting in false positives. The reason why single or conjunctive patterns are not sufficient is that programmers write source code in different ways to achieve the same programming task (as shown in Examples 1 and 2). For example, the pattern $P_1 \oplus P_2$[4] describes both the condition checks that can be used before the `next` method. Furthermore, using the pattern $P_1 \oplus P_2$ does not result in violations in `printEntries1` and `printEntries2`, thereby reducing false positives. We focus on mining patterns that describe nearly complete rather than complete behavior, since the patterns that describe complete behavior cannot help detect violations as deviations from those patterns, resulting in *false negatives*.

To reduce both false positives and false negatives among detected violations and to infer patterns that describe nearly complete behavior, we propose a novel approach, called *Alattin*, that includes new mining algorithms and a technique that detects neglected conditions (described next) using patterns mined by our mining algorithms. In particular, our algorithms mine patterns in four pattern formats: conjunctive (*And* or $\wedge$), disjunctive (*Or* or $\vee$), exclusive-disjunctive (*Xor* or $\oplus$), and combinations of these patterns (referred to as *Combo* patterns).

[4]The symbol $\oplus$ represents the exclusive-or relationship.

We use *Alternative Patterns* to collectively refer to patterns of all four formats and refer to individual patterns such as P_1 and P_2 in $P_1 \wedge P_2$ as alternatives.

In general, mining *Or* and *Xor* patterns is more challenging than mining *And* patterns, since *Or* and *Xor* patterns do not follow the Apriori principle [53] in data mining. Given an input database of itemsets for applying mining techniques, the Apriori principle states that if an itemset is frequent, then all its subsets should also be frequent. Existing mining techniques [22] that target at mining *And* patterns use this principle for pruning the search space. For example, if an itemset P_1 is not frequent, then any super *And* itemset of P_1 such as $P_1 \wedge P_2$ cannot be frequent, and hence can be pruned. However, the Apriori principle does not hold for mining *Or* and *Xor* patterns. For example, although the itemset P_1 is not frequent, its super *Or* itemset such as $P_1 \vee P_2$ can be frequent, since $P_1 \vee P_2$ is supported by more itemsets in the input database compared to P_1 or P_2 individually.

In this section, we show the benefits and limitations of these four pattern formats with respect to false positives and false negatives by applying these pattern formats to the problem of detecting neglected conditions. *Neglected conditions*, also referred to as missing paths, are known to be an important category of software defects and are considered to be one of the primary reasons for many fatal issues such as security or buffer overflow vulnerabilities [24]. As shown by a recent study [24], 66% (109/167) of defect fixes applied in the Mozilla Firefox project are due to neglected conditions. In particular, neglected conditions (related to an API call) refer to (1) missing conditions that check the arguments or receiver of the API call before the API call or (2) missing conditions that check the return values or receiver of the API call after the API call. In Alattin, we mine patterns that describe necessary condition checks related to an API call in these four formats and use those condition checks for detecting neglected conditions in applications under analysis.

Alattin makes the following main contributions:

- An empirical investigation of four pattern formats: conjunctive (*And* or $\wedge$), disjunctive (*Or* or $\vee$), exclusive-disjunctive (*Xor* or $\oplus$), and combinations of these patterns (referred to as *combo* patterns) in software engineering data.
- New mining algorithms for efficiently mining patterns in *Or*, *Xor*, and *Combo* pattern formats.
- A technique that applies patterns of these four pattern formats for detecting neglected conditions around individual API calls in an application under analysis.
- Two evaluations to demonstrate the effectiveness of Alattin. Our evaluation results show that the best pattern-mining approach (in terms of reducing both false positives and false negatives) is to first mine *And* patterns for API methods and next mine *Combo* patterns. Among violations detected by *Combo* patterns, the best violation-detection approach is to assign higher priority to the violations of API methods with *And* patterns compared

```
01:public Object evaluate(Object val) { ...
02:   if (val != null && val instanceof Collection) {
03:      Collection coll = (Collection) val;
04:      Iterator i = coll.iterator();
05:      if(!coll.isEmpty()) {
06:         for (; i.hasNext();) {
07:            Object obj = i.next();
08:            if(obj instanceof Node) {
09:               Node node = (Node) obj;
10:               //...
11:            } } } }
12:   return new Double(sum);
13:}
```

Figure 4.10: A code example using `Iterator.next` collected from Google code search.

IS1:	1 3 4
IS2:	4 1
IS3:	2 3
IS4:	1
IS5:	2
IS6:	4 1

1 : boolean check on return of Iterator.hasNext before Iterator.next
2 : const check on return of ArrayList.size before Iterator.next
3 : null check on argument of ArrayList.constructor after Iterator.next
4 : boolean check on return of Iterator.hasNext after Iterator.next

Input Database (ISD)

Figure 4.11: An example input database ISD.

to the violations of API methods without *And* patterns. The primary reason is that the former violations are more likely to be real defects compared to the latter violations.

4.3.2 Example

We next use an illustrative example to describe Alattin on how we collect the data that describes necessary condition checks around API calls. We also show how our proposed four pattern formats affect the number of false negatives and false positives among detected violations. Consider that an application under analysis uses the `Iterator.next` method as shown in the `printEntries2` method in Figure 4.9.

Initially, we collect relevant code examples that invoke the `Iterator.next` method by constructing queries to Google code search [48]. These relevant code examples are required for mining patterns that describe necessary condition checks around the `Iterator.next` method. Figure 4.10 shows a code example collected from Google code search. We next construct control-flow graphs (CFG) for collected code examples and perform two traversals (backward and forward) of the CFG from the node corresponding to the `Iterator.next` method. In the backward traversal, we collect the condition checks on the receiver and argument objects preceding the call site of the `Iterator.next` method. Similarly, in the forward traversal, we collect the condition checks on the receiver and return objects succeeding the call site of the `Iterator.next` method. For the current code example, our backward and forward traversals gather the following condition checks.

`a: boolean-check on the return of Iterator.hasNext before Iterator.next`

```
b: boolean-check on the return of Collection.isEmpty before Iterator.next
c: instance-check on the return of Iterator.next with org.w3c.dom.Node
d: boolean-check on the return of Iterator.hasNext after Iterator.next
```

Condition check "a" describes the condition check performed *before* the call site of the `next` method, whereas Condition check "d" describes the condition check performed *after* the call site of the `next` method. The reason for two condition checks is that the `next` method (Statement 7) is invoked in a `for` loop. Section 4.3.4 presents more details on how we exploit program dependencies while performing backward and forward traversals. The preceding set of condition checks collected from the code example forms an itemset in the itemset database ISD, used as input for mining patterns. We analyze all collected code examples to generate various itemsets and use different mining algorithms for mining the patterns in formats: *And*, *Or*, *Xor*, and *Combo*. Section 4.3.3 presents more details on our mining algorithms.

Figure 4.11 shows a sample itemset database ISD with six itemsets. This ISD includes four distinct items labeled with IDs 1 through 4. The figure also shows the condition check corresponding to each item. We next apply our mining algorithms for mining different pattern formats. The patterns mined by our algorithms with a minimum support threshold value *min_sup* of 0.4 are shown below:

- *And Pattern:* "$1 \wedge 4$", support: 0.5
- *Or Pattern:* "$1 \vee 2 \vee 3 \vee 4$", support: 1.0
- *Xor Pattern:* "$1 \oplus 2$", support 1.0; "4", support: 0.5
- *Combo Pattern:* "$(1 \wedge 4) \oplus 2$", support: 0.83

Among the itemsets shown in ISD, Items 1 and 4 often appear together with an $\wedge$ relation, since the methods `hasNext` and `next` are often used in a loop (as shown in Statements 6 and 7 in Figure 4.10). Although the *And* pattern captures this behavior, the *And* pattern cannot capture the relation with Item 2, resulting in false positives when applied on code examples such as `printEntries2` in Figure 4.9. On the other hand, the *Or* pattern does not result in false positives, since the pattern includes all items. However, the *Or* pattern does not help in detecting violations, resulting in false negatives. Although the *Xor* pattern can perform better than the *Or* pattern, the *Xor* pattern may not detect violations in code examples, which include only Item 1 and do not include Item 4. As shown in this example, the *Combo* pattern describes the nearly complete behavior and also helps reduce both false positives and false negatives.

Along with the challenges faced while mining *Or* and *Xor* patterns, *Combo* patterns pose additional challenges in choosing the suitable operator while combining items. For example, the suitable operator for combining items "1" and "4" is $\wedge$, although "$1 \vee 4$" results in a higher

support value than "1 $\wedge$ 4". We describe these challenges in subsequent sections and present our algorithms for mining these patterns.

In summary, this example illustrates the existence of pattern formats *And*, *Or*, and *Xor*, and also shows that no single pattern format alone can help in describing the necessary condition checks around API calls.

4.3.3 Mining Algorithms for Alternative Patterns

We next describe four pattern formats that we use in our empirical study. We first present formal definitions for the four pattern formats and next describe our algorithms for mining the pattern in four formats with illustrative examples.

Formal Definitions

Let $M = \{m_1, m_2, ..., m_k\}$ be the set of all possible distinct *items*. For example, an m_j represents a condition check such as "`boolean-check` on return of `Iterator.hasNext` before `Iterator.next`". Consider an ItemSet Database ISD as $\{is_1, is_2, ...,is_l\}$, where each itemset is_j includes different sets of elements such as $\{m_1, m_2, ..., m_a\}$ from the set of all possible distinct elements.

Definition 4.3.1 Pattern Candidate. A pattern candidate pc is a single item $m_k \in M$ or a combination of two elements associated by a logical operator $op \in \{\wedge, \vee, \oplus\}$. Each element in the combination is an item $m_i \in M$ or another pattern candidate.

The preceding definition is a recursive definition, which defines that a pattern candidate can be either a simple or nested pattern candidate. For example, a simple pattern candidate $pc_k : m_i \wedge m_j$ is a combination of two items m_i and m_j with the operator $op \in \{\wedge\}$. On the other hand, a nested pattern candidate $pc_l : m_i \vee pc_k$ is a combination of an item m_i and the preceding pattern candidate pc_k with the operator $op \in \{\vee\}$. We use notations $pc_k.left$ and $pc_k.right$ to refer to the left and right *child* pattern candidates, respectively, and refer to pc_k as their *parent* pattern candidate. Furthermore, we use the notation $pc_k.op$ to refer to the operator op of the pattern candidate pc_k. We classify a pattern candidate by its format, especially using its operator.

Definition 4.3.2 *And* Pattern Candidate. An *And* pattern candidate is a pattern candidate where the operator $op \in \{\wedge\}$ and all the child pattern candidates are also *And* pattern candidates.

Definition 4.3.3 *Or* Pattern Candidate. An *Or* pattern candidate is a pattern candidate where the operator $op \in \{\vee\}$ and all the child pattern candidates are also *Or* pattern candidates.

Definition 4.3.4 *Xor* Pattern Candidate. A *Xor* pattern candidate is a pattern candidate where the operator $op \in \{\oplus\}$ and all the child pattern candidates are also *Xor* pattern candidates.

Definition 4.3.5 *Combo* Pattern Candidate. A *Combo* pattern candidate is a pattern candidate where the operator $op \in \{\wedge, \vee, \oplus\}$.

The category of *Combo* pattern candidates subsumes the categories of *And*, *Or*, and *Xor* pattern candidates. An example combo pattern candidate is "$pc_1 \oplus pc_2$", where pc_1 and pc_2 are "$m_i \wedge m_j$" and "$m_k \wedge m_l$", respectively.

To compute frequent patterns among pattern candidates, we use a threshold value, referred to as *min_sup*, that describes the minimum support for a pattern candidate to be classified as a frequent pattern. Algorithm 4, `IsSupportedBy`, describes how we compute support values for the preceding pattern formats. In particular, `IsSupportedBy` accepts a pattern candidate pc_k and an itemset is_j, and returns `true`, if is_j supports pc_k, and otherwise returns `false`. Initially, `IsSupportedBy` checks whether pc_k is a single item and returns `true` or `false` based on whether pc_k is contained in is_j. Otherwise, `IsSupportedBy` recursively computes whether $pc_k.left$ and $pc_k.right$ of pc_k are supported by the itemset is_j, and uses the operator $pc_k.op$ for checking whether is_j supports pc_k (Lines 4 - 15). We compute the support value of a pattern candidate pc_k, referred to as `Support(`pc_k`)`, based on the number of the itemsets (in ISD) that return `true` for the algorithm `IsSupportedBy`.

Definition 4.3.6 Frequent Pattern (FP). A pattern candidate pc_k is considered as a frequent pattern, if `Support(`pc_k`)` $\geq$ *min_sup*.

A frequent pattern fp_k is considered as an *And*, *Or*, *Xor*, or *Combo* pattern based on the operator $fp_k.op$.

Mining Algorithms

We next present our algorithms for mining preceding pattern formats. In particular, we present algorithms for mining *Or*, *Xor*, and *Combo* patterns, since mining *And* patterns can be achieved by well-known approaches such as a frequent itemset miner [22]. We first explain our algorithm for mining *Or* and *Xor* patterns, and next describe the algorithm for mining *Combo* patterns.

Mining Or and Xor Patterns. In general, mining *Or* and *Xor* patterns is more challenging than mining *And* patterns, since these patterns do not follow the Apriori principle [53]. The Apriori principle states that if an itemset is frequent, then all its subsets should also be frequent. Existing mining techniques that target at mining *And* patterns use this principle for pruning

Algorithm 4 IsSupportedBy (pc_k, is_j)

Require: PatternCandidate pc_k, ItemSet is_j
Ensure: `true`, if is_j supports pc_k
Ensure: `false`, if is_j does not support pc_k

```
1: if pc_k is a SingleItem then
2:     return pc_k ∈ is_j
3: else
4:     bool lefts = isSupportedBy(pc_k.left, is_j)
5:     bool rights = isSupportedBy(pc_k.right, is_j)
6:     if pc_k.op == ∨ then
7:         return lefts ∨ rights
8:     end if
9:     if pc_k.op == ∧ then
10:        return lefts ∧ rights
11:    end if
12:    if pc_k.op == ⊕ then
13:        return lefts ⊕ rights
14:    end if
15: end if
```

the search space. For example, if an itemset pc_1 is not frequent, then any super itemset of pc_1 such as $pc_1 \wedge pc_2$ cannot be frequent, and hence can be pruned. However, the Apriori principle does not hold for mining *Or* and *Xor* patterns. For example, even if the itemset pc_1 is not frequent, then its super itemset such as $pc_1 \vee pc_2$ can be frequent. To address this issue, we propose a greedy algorithm based on the following property for pruning the search space of patterns. This property is inspired by Nanavati et al. [92] and is applicable to both *Or* and *Xor* patterns.

Property 1 The support of an *Or* or *Xor* pattern candidate pc_k, represented as `Support(`pc_k`)`, formed from two pattern candidates $pc_k.left$ and $pc_k.right$ should have higher value than `Support(`$pc_k.left$`)` and `Support(`$pc_k.right$`)`.

The rationale behind the preceding property is based on our objective to mine patterns that describe nearly complete behavior. For example, a pattern candidate $pc_k = pc_i \vee pc_j$, whose support value is less than `Support(`pc_i`)` and `Support(`pc_j`)` is not useful compared to individual pattern candidates in achieving our objective. Algorithm 5, `MineXorOr`, describes our greedy algorithm for mining *Or* and *Xor* patterns. `MineXorOr` accepts itemset database ISD, *min_sup*, and *ptype* $\in \{\vee, \oplus\}$ as inputs.

Initially, `MineXorOr` identifies all distinct items in the itemset database ISD using the function `ComputeDistinct` (Line 2). Among these distinct items, `MineXorOr` checks whether the sup-

Algorithm 5 MineXorOr(ISD, min_sup, $ptype$)

Require: ISD, min_sup, $ptype$
Ensure: Set<PC> $PCSet$

```
1:  PCSet = φ
2:  Set<M> distinctItems = ComputeDistinct(ISD)
3:  for all m_i ∈ distinctItems do
4:      if Support(m_i, ISD) >= min_sup then
5:          PCSet+ = m_i
6:      end if
7:  end for
8:  Set<PC> CurrSet = distinctItems
9:  loop
10:     Set<PC> NextSet = φ
11:     Set<PC> ChildSet = φ
12:     Set<PC> PWSet = ComputePairwise(CurrSet, ptype)
13:     for all pc_j ∈ PWSet do
14:         sval = Support(pc_j, ptype)
15:         if (sval < min_sup || sval ≤ Support(pc_j.left) || sval ≤ Support(pc_j.right))
            then
16:             Continue
17:         end if
18:         NextSet+ = pc_j
19:         PCSet+ = pc_j
20:     end for
21:     if NextSet.size() ≤ 1 then
22:         break
23:     end if
24:     NextSet = ApplyGreedy(CurrSet, NextSet)
25:     for all pc_j ∈ CurrSet do
26:         if pc_j ∉ {pc_k.left, pc_k.right}, ∀pc_k ∈ NextSet then
27:             NextSet+ = pc_j
28:         end if
29:     end for
30:     CurrSet = NextSet
31: end loop
32: return PCSet
```

port of any of these items is greater than min_sup and adds those items to $PCSet$ (Lines 3 to 7). Next, `MineXorOr` uses various iterations, where pairwise combinations are computed using the function `ComputePairwise` from the elements of the previous iteration stored in $CurrSet$ (Lines 9 to 31). For example, consider Or patterns. For the `CurrSet` $= \{pc_1, pc_2, pc_3\}$, `ComputePairwise` returns a set with three elements, i.e., `PWSet` $= \{pc_1 \vee pc_2,\ pc_1 \vee pc_3,\ pc_2 \vee pc_3\}$. The algorithm

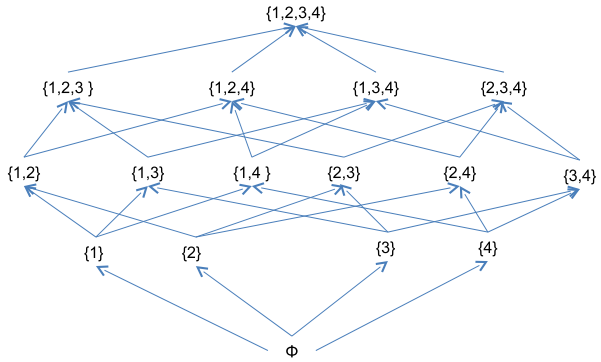


Figure 4.12: All possible itemsets with four distinct items.

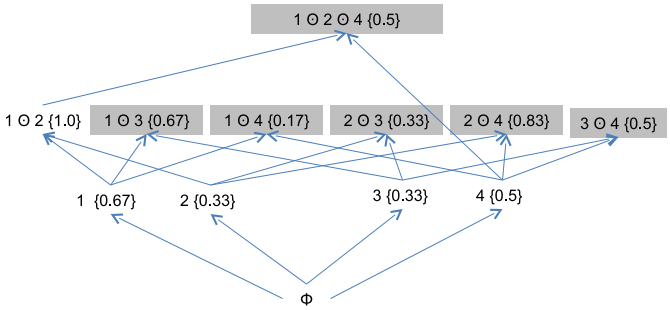


Figure 4.13: Mining *Xor* patterns.

next identifies the elements in `PWSet`, whose support values are greater than *min_sup* and satisfy Property 1. `MineXorOr` next chooses pattern candidates (from $NextSet$) that participate in the next iteration by greedily choosing one parent pattern candidate in $NextSet$ with the highest support value for each pattern candidate in $CurrSet$ using the function `ApplyGreedy` (Line 24). The rationale behind our greedy approach is based on our empirical investigation that the real patterns describing necessary condition checks around API calls often have higher support values compared to other patterns. Finally, `MineXorOr` identifies those pattern candidates in $CurrSet$ whose parent pattern candidates do not belong to $NextSet$ (Lines 25 - 29). `MineXorOr` adds such pattern candidates to $NextSet$ as well, since these pattern candidates can still be helpful when combined with other pattern candidates in $NextSet$.

We next explain the algorithm in detail using the itemset database shown in Figure 4.11. The itemset database includes four distinct items. Figure 4.12 shows all possible pattern candidates that can be derived using the preceding four distinct items. This figure shows the complete search space of pattern candidates. We first explain mining *Xor* patterns and next explain mining *Or* patterns.

Mining *Xor* Patterns. Figure 4.13 shows how `MineXorOr` prunes the search space and generates patterns "$1 \oplus 2$" (support 1.0) and "4" (support: 0.5). The value shown in braces next to each pattern candidate indicates support value of that pattern candidate. The pattern

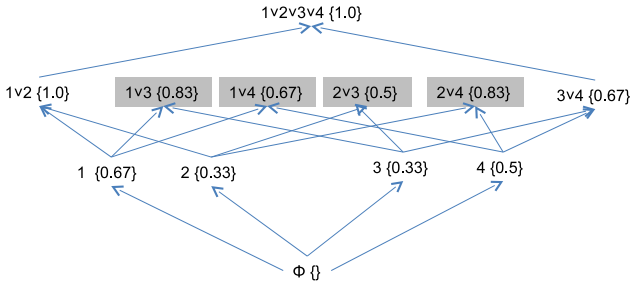


Figure 4.14: Mining *Or* patterns.

candidates shown in gray are pruned by `MineXorOr` due to three factors. First, the support value of pattern candidate is lower than *min_sup*. For example, $\texttt{Support}(2 \oplus 3) = 0.33$, which is lower than *min_sup*. Second, the pattern candidate does not satisfy Property 1. For example, the pattern candidate "$1 \oplus 3$" does not satisfy Property 1, since $\texttt{Support}(1 \oplus 3) \leq \texttt{Support}(1)$. Third, the pattern candidate is not the candidate with the highest support value among the parent pattern candidates of each child pattern candidate. For example, the pattern candidate "$2 \oplus 4$" is pruned, since the pattern candidate "$1 \oplus 2$" has a higher support value than "$2 \oplus 4$". Since all parent pattern candidates of "4" are pruned away, `MineXorOr` adds this candidate to $NextSet$ for the next iteration and computes further pattern candidates such as "$1 \oplus 2 \oplus 4$".

Mining *Or* Patterns. Figure 4.14 shows how `MineXorOr` prunes the search space and generates the *Or* pattern "$1 \vee 2 \vee 3 \vee 4$" (support: 1.0). Similar to *Xor* patterns, the pattern candidates are pruned due to the same three factors.

Mining Combo Patterns. We next describe how we mine combo patterns. The algorithm for mining *Combo* patterns includes two phases. Phase 1 mines *And* Patterns and Phase 2 mines *Combo* patterns using the output of Phase 1. We next explain each phase in detail.

Phase 1. Algorithm 6, `MineComboP1`, shows Phase 1 of mining *Combo* patterns. In particular, `MineComboP1` computes pairwise *And* combinations of all pattern candidates in $CurrSet$ and checks whether new pattern candidates have higher support values than *min_sup* (Lines 9 - 11). If yes, `MineComboP1` computes the support of *Xor* combination of the two candidates, shown as $\texttt{Support}(pc_j.left \oplus pc_j.right)$ in Line 12. If the preceding support value is also higher than *min_sup*, then `MineComboP1` ignores the *And* combination. The rationale behind this decision is that, if both "$pc_j.left \wedge pc_j.right$" and "$pc_j.left \oplus pc_j.right$" have higher values than *min_sup*, then the suitable combination of $pc_j.left$ and $pc_j.right$ is "$pc_j.left \vee pc_j.right$".

Figure 4.15 shows the output of `MineComboP1` with the itemset database ISD shown in Figure 4.11. As shown in the figure, Phase 1 produces three pattern candidates as output: "$1 \wedge 4$", "2", and "3", which are passed as inputs to Phase 2.

Phase 2. Phase 2 of mining *Combo* patterns is similar to Algorithm 5 for mining *Xor* and *Or* patterns. The major difference is to choosing a suitable operator, $op \in \{\vee, \oplus\}$, when

Algorithm 6 MineComboP1(ISD, min_sup)

Require: ISD, min_sup, $ptype$
Ensure: Set<PC> $PCSet$
1: Set<M> $distinctItems$ = ComputeDistinct(ISD)
2: Set<PC> $CurrSet = distinctItems$
3: $PCSet = distinctItems$
4: **loop**
5: Set<PC> $NextSet = \phi$
6: Set<PC> $PWSet$ = ComputePairwise($CurrSet$, "And")
7: **for all** $pc_j \in PWSet$ **do**
8: $sval = Support(pc_j, ptype)$
9: **if** $sval < min\_sup$ **then**
10: **Continue**
11: **end if**
12: **if** $Support(pc_j.left \oplus pc_j.right) \geq min\_sup$ **then**
13: **Continue**
14: **end if**
15: $NextSet+ = pc_j$
16: $PCSet- = pc_j.left$
17: $PCSet- = pc_j.right$
18: $PCSet+ = pc_j$
19: **end for**
20: **if** $NextSet.size() \leq 1$ **then**
21: break
22: **end if**
23: $CurrSet = NextSet$
24: **end loop**
25: **return** $PCSet$

combining two pattern candidates during the computation of pairwise combinations. Given two pattern candidates pc_i and pc_j, Phase 2 chooses the $\vee$ operator if "`Support`$(pc_i \wedge pc_j) \geq min\_sup$ && `Support`$(pc_i \oplus pc_j) \geq min\_sup$"; otherwise, Phase 2 chooses the $\oplus$ operator. The rationale behind this decision is the same as the reason given in Phase 1. Figure 4.16 shows the output of Phase 2 resulting in the pattern "$(1 \wedge 4) \oplus 2$" (support: 0.83).

4.3.4 Approach

Alattin, developed based on our WebMiner framework, accepts an application under analysis and detects neglected conditions around APIs reused by the application. Alattin includes all four phases of our WebMiner framework. More specifically, Alattin scans the application and gathers API classes and methods reused by the application. In the *search* phase, Alattin collects

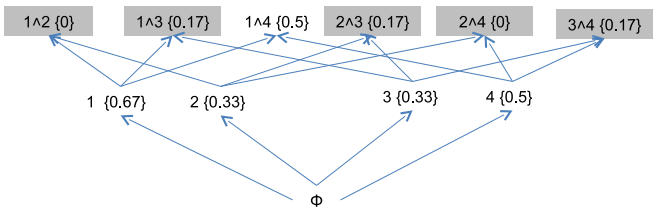


Figure 4.15: Phase 1 of mining *Combo* patterns.

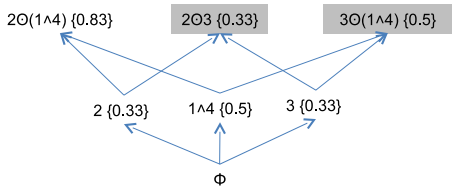


Figure 4.16: Phase 2 of mining *Combo* patterns.

relevant code examples that reuse those API classes and methods. In the *analyze* phase, Alattin analyzes collected code examples or application under analysis to generate pattern candidates suitable for mining. In the *mine* phase, Alattin applies mining algorithms on pattern candidates to mine patterns. In the *apply* phase, Alattin detects violations of mined patterns in the application under analysis. We next explain all phases except the *search* phase, which is the same as our WebMiner framework. We use notations C_i and F_i to denote a class or a method used by the application under analysis, respectively.

Analyze Phase

In the *analyze* phase, Alattin analyzes collected code examples or application under analysis statically to generate pattern candidates suitable for mining. These pattern candidates include condition checks that are performed before and after invoking an F_i method. To identify these condition checks on method calls, Alattin has to associate condition checks in the conditional expressions of `If` or `While` statements with the related method calls. We use the `Iterator.next` method and its relevant code example in Figure 4.10 as a running example for explaining Phase 2.

Initially, Alattin constructs CFGs for each code example. More details on constructing CFGs are described in the *analyze* phase of our WebMiner framework (Section 2.3). In particular, Alattin constructs CFGs for each code example with two kinds of nodes: control (CT) and non-control (NT) nodes. Control nodes represent control-flow statements such as *if*, *while*, and *for*, which control the flow of the program execution. Non-control nodes represent other statements such as method calls or type casts. For example, Statement 5 in the code example (Figure 4.10) is a control node and Statement 9 is a non-control node. When encountering a control node, say CT_i (i indicates the statement id), Alattin also extracts all variables, say

$\{V_1, V_2, ..., V_n\}$, that participate in the conditional expression of that node and the condition checks on those variables. For example, the control node CT_2 includes the {(`val`, `null-check`), (`val`, `instance-check`)} pairs. If the control node includes comparisons with expressions such as method calls, our approach stores those method calls also as additional information within the control node. When encountering a non-control node such as a method call, Alattin extracts variables such as {`receiver`, `argument1`, ..., `argumentN`} associated with the method call.

Alattin next traverses CFG and associates gathered condition checks with their related method calls such as `Iterator.hasNext`. The traversal phase includes two kinds of traversals: backward and forward. Alattin performs a backward traversal from the call site such as NT_7 of the F_i method to collect condition checks on the receiver and argument objects preceding the call site. Similarly, Alattin performs a forward traversal to collect condition checks on the receiver and return objects after the call site of the F_i method. In each traversal, Alattin exploits program dependencies for associating condition checks with method calls. Failing to consider these program dependencies may result in programming rules that are not semantically related as shown in the limitations of the PR-Miner [75] and DynaMine [78] approaches. To exploit program dependencies, Alattin uses the concept of *dominance* with a combination of *control-flow* and *data-flow* dependencies.

Definition: *A node N* ***dominates*** *another node M in a control flow graph (represented as N dom M) if every path from the starting node of the CFG to M includes N.*

Initially, Alattin identifies the dominant CT_i nodes for each NT_k node. For example, the control node CT_6 dominates the non-control node NT_7. Alattin computes the intersection between the variable set associated with the CT_i node, say $\{V_1, V_2, ..., V_n\}$, and the receiver or argument variables of the NT_k node, say {`receiver`, `argument1`, ..., `argumentN`}. If the intersection $\{V_1, V_2, ..., V_n\} \cap$ {`receiver`, `argument1`, ..., `argumentN`} $\neq \emptyset$, Alattin checks whether the NT_k node is dependent on the CT_i node, i.e., whether there exists at least one variable of NT_k node involved in the CT_i node and is not redefined in the path between CT_i and NT_k nodes. If the NT_k node is dependent on the CT_i node, Alattin adds the condition check to the pattern candidate. For example, the extracted condition check for nodes CT_6 and NT_7 in the code example is "`boolean-check` on return of `Iterator.hasNext` before `Iterator.next`", which indicates that a `boolean-check` must be done on the return variable of the `hasNext` method before the call site of `Iterator.next`. In our experience, we found that there can be various code examples without any condition checks around an F_i method. Failing to consider these code examples can assign incorrect support values to mined patterns. To address this issue, we add an *Empty Pattern Candidate* to the input database ISD for each such code example.

Mine Phase

In the *mine* phase, Alattin uses our mining algorithms (described in Section 4.3.3) to mine patterns in all four pattern formats: *And*, *Or*, *Xor*, and *Combo* patterns. Alattin applies our mining algorithms on pattern candidates of each F_i method individually. The reason is that if we apply mining algorithms on all pattern candidates together, the patterns related to an F_i method with a few pattern candidates can be missed due to patterns (related to other M_j methods) with a large number of pattern candidates. For each F_i, Alattin mines patterns in all four pattern formats. We used a *min_sup* threshold value of 0.4 based on our empirical experience [121].

Apply Phase

In the *apply* phase, Alattin detects violations of mined patterns in the application under analysis statically. More specifically, Alattin gathers condition checks around each call site of an F_i method in the application under analysis. Alattin constructs an itemset is_j using gathered condition checks. For each mined pattern pc_k in all patterns of four formats, Alattin uses `IsSupportedBy` (Algorithm 4) to check whether the itemset is_j supports the mined pattern pc_k. If the itemset does not support the mined pattern, Alattin reports a violation. For each detected violation, Alattin assigns a support value as the same value as the support value of the associated mined pattern used to detect the violation.

4.3.5 Evaluation

We conducted two evaluations to assess the effectiveness of Alattin. We use the APIs provided by three Java default API libraries to show the existence of alternative patterns. We next use four popular applications to show the benefits and limitations of alternative patterns with respect to false positives and false negatives among detected violations. The details of subjects and results of our evaluations are available at `https://sites.google.com/site/asergrp/projects/alattin/`. We next present research questions addressed in our evaluations.

Research Questions

In our evaluations, we address the following research questions.

- RQ1: How high percentage of *And*, *Or*, *Xor*, and *Combo* patterns represent real programming rules, respectively? Since real programming rules are required for detecting violations in applications under analysis, this research question helps to show the pattern formats that are suitable for detecting violations.

- RQ2: How low percentage of false negatives and false positives exist among violations detected using *And*, *Or*, *Xor*, and *Combo* patterns, respectively? Since false positives are one of the common issues faced by existing static defect-detection techniques, this research question helps to show that the patterns that describe nearly complete behavior (such as *Or* or *Combo*) help reduce the number of false positives with no or low increase of false negatives.

Subject Applications

We next present subject applications used in our evaluations. In our evaluations, we used three Java default API libraries and four popular open source libraries. Table 4.7 shows the characteristics of the subject applications. Columns "Classes" and "Methods" show the number of classes and methods, respectively. For mining patterns of three Java default API libraries, we gathered 49858, 5555, and 15052 code examples for Java Util, Java Transaction, and Java SQL, respectively. Column "KLOC" shows the kilo lines of code in each subject application.

The Java Util package[5] includes the collections framework and other popular utilities used by many different applications. Java Transactions[6] and Java SQL[7] are industry standards for developing multi-tier server-side Java applications. Hibernate[8] and HsqlDB[9] abstract relational databases to use an object-oriented methodology. Columba[10] is an open source email-client application written in Java. Columba provides a user-friendly graphical interface and is suitable for internationalization support. The BCEL library[11], developed by Apache, is mainly used to analyze, create, and manipulate Java class files. We selected these applications, since these applications are popular open source applications and are used as subjects in evaluating previous related approaches [122, 133].

RQ1: Alternative Patterns

We next address the first research question of whether alternative patterns exist in real applications and how high percentage of those patterns represent real programming rules. To address this question, we configured Alattin, which by default accepts an application under analysis and mines patterns for third-party APIs, to accept a set of classes and methods directly. In this mode of operation, Alattin mines patterns (programming rules) for the APIs of the given classes and methods.

[5] `http://java.sun.com/j2se/1.4.2/docs/api/java/util/package-summary.html`
[6] `http://java.sun.com/javaee/technologies/jta/`
[7] `http://java.sun.com/j2se/1.4.2/docs/api/java/sql/package-summary.html`
[8] `http://www.hibernate.org/`
[9] `http://hsqldb.org/`
[10] `http://sourceforge.net/projects/columba/`
[11] `http://jakarta.apache.org/bcel/`

Table 4.7: Subject applications and their characteristics.

Application	**#Classes**	**#Methods**	**KLOC**
Java Util APIs	19	144	-
Java Transaction APIs	7	37	-
Java SQL APIs	14	93	-
Hibernate	478	4334	118
HsqlDB	143	1178	41
Columba	1500	7674	136
BCEL	357	2691	32
Total	2518	16151	327

Table 4.8: Alternative patterns mined by Alattin.

Application	***And* Patterns**				***Or* Patterns**			
	Total	#RR	#PR	#FP	Total	#RR	#PR	#FP
Java Util	40	34	0	6	51	25	19	7
Java Sql	26	26	0	0	24	21	3	0
Java Transaction	3	3	0	0	9	2	4	3
	***Xor* Patterns**				***Combo* Patterns**			
	Total	#RR	#PR	#FP	Total	#RR	#PR	#FP
Java Util	54	35	11	8	50	32	11	7
Java Sql	33	30	3	0	24	21	3	0
Java Transaction	8	2	4	2	8	2	4	2

RR: Real Rules, PR: Partial Rules, FP: False Positives

Table 4.8 shows the patterns mined in all four formats: *And*, *Or*, *Xor*, and *Combo*. For each pattern format, Columns "Total", "RR", "PR", and "FP" show the total number of mined patterns, real rules, partial rules, and false positives, respectively. *Real rules* describe properties that must be satisfied when using an API method. In real rules, all alternatives are real properties. In contrast to real rules, some alternatives in *Partial rules* do not represent real properties. The reason for introducing partial rules is that partial rules are as effective as real rules in reducing false-positive defects; however, partial rules can increase false-negative defects due to false-positive alternatives among mined patterns. *False positives* represent mined patterns where none of the alternatives represents real properties. To mine patterns in all four pattern formats, Alattin took 13, 1, and 1 seconds for Java Util, Java Sql, and Java Transaction, respectively. All experiments were conducted on a machine with 2.2GHz Intel processor and 3GB RAM. We used available on-line documentations, JML specifications[12], or source code of applications for classifying mined patterns into these three categories.

Figure 4.17 shows the percentages of real, partial, and false-positive rules among mined patterns of each pattern format. In summary, a high percentage of *And*, *Or*, *Xor*, and *Combo* patterns represent real rules. We also found that *Or*, *Xor*, and *Combo* patterns include new real rules that do not exist among *And* patterns. Since existing approaches mine only *And* patterns, our results show that new defects can be detected using *Or*, *Xor*, and *Combo* patterns. The

[12] `http://www.eecs.ucf.edu/~leavens/JML/`

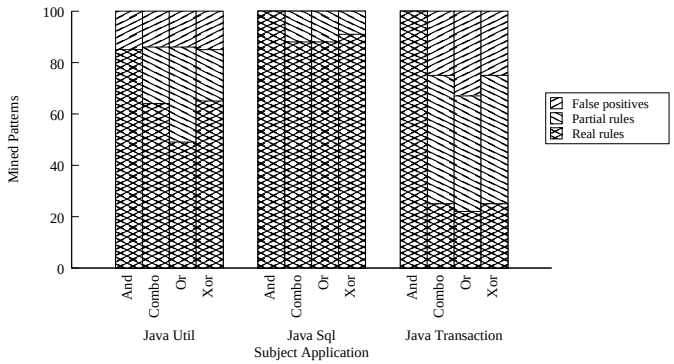


Figure 4.17: Classification of mined patterns

figure also shows that, except *And* patterns, all other pattern formats include a considerable percentage of partial rules. Therefore, although these three pattern formats can help reduce false positives, these three pattern formats can result in false negatives among detected violations. Furthermore, *Or* patterns have higher percentage of partial rules compared to *Xor* and *Combo* patterns, indicating that *Or* patterns result in *more* false negatives compared to *Xor* and *Combo* patterns.

We next present example patterns in each pattern format for the `read` method of the `java.util.jar.JarInputStream` class, which extends the `ZipInputStream` class. This class is used for reading the contents of a Jar file from any input stream such as `FileInputStream`. This class includes three methods: `getNextEntry`, `getNextJarEntry`, and `read`. The `getNextEntry` method reads the next Zip file entry, represented as an instance of the `ZipEntry` class, and positions the stream at the beginning of the entry data in the Jar file. On the other hand, the `getNextJarEntry` method reads the next Jar file entry, represented as an instance of the `JarEntry` class, and positions the stream at the beginning of the entry data. Indeed, the `JarEntry` class extends the `ZipEntry` class and includes additional methods such as `getAttributes` for reading the attributes specific to the Jar file. In general, programmers use either `getNextEntry` or `getNextJarEntry` for iterating through the entries in the Jar file and for reading the contents using the `read` method. Furthermore, if there is only one entry to read from the Jar file, the `read` method is used directly without using either `getNextEntry` or `getNextJarEntry`. Figure 4.18 shows an example usage of `getNextEntry` and `read` methods. This code example is extracted from Apache's Jakarta Cactus project[13].

Figure 4.19 shows the patterns mined for the `read` method in all four formats. The *And* pattern includes only one alternative P_1, which describes that there should be a condition check with "-1" on the return value of the `read` method. Here, "-1" indicates that the end of the entry is reached. The *Or* Pattern includes two alternatives "$P_1 \vee P_2$", where P_2 indicates that there should be a `null-check` on the return of the `getNextJarEntry` method. The reason that the *And* pattern could not mine the alternative P_2 is that there are various scenarios

[13] `http://jakarta.apache.org/cactus/`

```
00: ...
01: JarInputStream in;
02: ZipEntry ze; ...
03: while ((ze = in.getNextEntry()) != null){
04:    if(thePath.equals(zipEntry.getName())){
05:       ByteArrayOutputStream buffer = new ByteArrayOutputStream();
06:       byte[] bytes = new byte[2048];
07:       int bytesRead;
08:       while((bytesRead = in.read(bytes)) != -1){
09:          buffer.write(bytes, 0, bytesRead);
10:       }
11:       return new ByteArrayInputStream(buffer.toByteArray());
12:    }
13: } ...
```

Figure 4.18: Alternative patterns mined for the `read` method of the `JarInputStream` class.

where P_1 and P_2 are not used together. For example, when P_2 is used, programmers often get the size of the buffer to be read using the `getSize` method of `JarEntry`, which is the return type of the `getNextJarEntry` method. Therefore, programmers often do not explicitly check the return value of the `read` method, when the `getNextJarEntry` method is used. Figure 4.19 also shows that there are two *Xor* patterns. Interestingly, the second pattern "$P_2 \oplus P_3$" shows that programmers often use either `getNextJarEntry` or `getNextEntry`, but not both together, since the related pattern "$P_2 \vee P_3$" is not being mined. However, the *Xor* pattern alone could not mine the relation among all alternatives P_1, P_2, and P_3. The *Combo* pattern addresses this issue via mining the pattern "$P_1 \vee (P_2 \oplus P_3)$", and shows the relation among all alternatives.

RQ2: False Positives and False Negatives

We next address the second research question of whether alternative patterns help reduce false positives among detected violations. We also address whether these patterns introduce no or a low percentage of false negatives among detected violations. To address this question, we used the four subject applications (Hibernate, Columba, BCEL, and HsqlDB) shown in Table 4.7. In particular, we mined patterns in all four formats from these applications under analysis and apply mined patterns on those applications to detect violations. We next inspected detected violations to classify violations as real defects or false positives based on available specifications such as JML and call sites of related API methods in source code of these subject applications. In our inspection, we ignored the violations related to API methods whose all pattern formats include only one alternative, since our objective is to show the benefits and limitations of alternative patterns. The primary reason is that such patterns do not help show benefits of alternative patterns, since those patterns have the same number of false positives or false negatives in all pattern formats. To compute false negatives, we need a baseline that shows the number of defects exist in subject applications. Since such a baseline does not exist for these

Method: `JarInputStream.read (byte[], int, int)`
***A*. *And* Pattern**
Pattern: "P_1", SUP(P_1): 0.63
P_1: "`const-check` on the return of `JarInputStream.read` with -1"
***B*. *Or* Pattern**
Pattern: "$P_1 \vee P_2$", SUP($P_1 \vee P_2$): 0.67
P_1: "`const-check` on the return of `JarInputStream.read` with -1"
P_2: "`null-check` on the return of `JarInputStream.getNextJarEntry()` before `JarInputStream.read`"
***C*. *Xor* Patterns**
Pattern: "P_1", SUP(P_1): 0.63
P_1: "`const-check` on the return of `JarInputStream.read` with -1"
Pattern: "$P_2 \oplus P_3$", SUP($P_2 \oplus P_3$): 0.52
P_2: "`null-check` on the return of `JarInputStream.getNextJarEntry()` before `JarInputStream.read`"
P_3: "`null-check` on the return of `JarInputStream.getNextEntry()` before `JarInputStream.read`"
***D*. *Combo* Pattern**
Pattern: "$P_1 \vee (P_2 \oplus P_3)$ ", SUP($P_1 \vee (P_2 \oplus P_3)$): 0.67
P_1: "`const-check` on the return of `JarInputStream.read` with -1"
P_2: "`null-check` on the return of `JarInputStream.getNextJarEntry()` before `JarInputStream.read`"
P_3: "`null-check` on the return of `JarInputStream.getNextEntry()` before `JarInputStream.read`"
Figure 4.19: Alternative patterns mined for the `read` method of the `JarInputStream` class.

applications, we identified all distinct real defects detected using patterns in all pattern formats and used those defects as a baseline for computing false negatives among violations detected using each pattern format.

Table 4.9 shows detected violations in all subject applications. Column "Real Defects" shows the total number of distinct defects detected using all pattern formats in each application. We used these defects as a baseline for computing the number of false negatives among violations detected using each pattern format. For each pattern format, Columns "Total", "RD", "FN", and "FP" show the total number of violations, real defects, false negatives (their percentage), and false positives (their percentage), respectively. We next summarize our findings for each pattern format with respect to real defects, false negatives, and false positives.

Real defects and False negatives. Figure 4.20 shows comparison between real defects and false negatives for the four pattern formats in each subject application. The figure shows that *Or*, *Xor*, and *Combo* patterns helped detect new defects that are not detected using *And* patterns. For example, in the Columba application, *Or* patterns detected 41 real defects, whereas *And* patterns detected only 26 real defects. The reason for new defects is due to the new patterns mined using the *Or*, *Xor*, and *Combo* pattern formats.

Regarding false negatives, our results show that the violations detected using *And* patterns include a high percentage of false negatives in 3 out of 4 applications. Since *And* patterns represent patterns that can be mined by existing approaches, the results show the ineffectiveness

Table 4.9: Analysis of violations detected in subject applications.

Application	**# Real Defects**	***And* Patterns**						***Or* Patterns**					
		Total	#RD	#FN	%	#FP	%	Total	#RD	#FN	%	#FP	%
Columba	49	117	26	23	47	91	78	113	41	8	16.3	72	63.7
Hibernate	22	93	14	8	36	71	76	177	17	5	22.7	160	90.4
Hsqldb	6	13	6	0	0	7	53.8	5	5	1	16.7	0	0
BCEL	1	2	0	1	100	2	100	13	1	0	0	12	92.3
		***Xor* Patterns**						***Combo* Patterns**					
		Total	#RD	#FN	%	#FP	%	Total	#RD	#FN	%	#FP	%
Columba	49	164	49	0	0	115	73	144	47	2	4	97	67
Hibernate	22	214	21	1	4.5	193	90.2	195	19	3	13.6	176	90.3
HsqlDB	6	11	6	0	0	5	45.5	10	6	0	0	4	40
BCEL	1	20	1	0	0	19	95	16	1	0	0	15	93.8

RD: Real Defects, FN: False Negatives, FP: False Positives

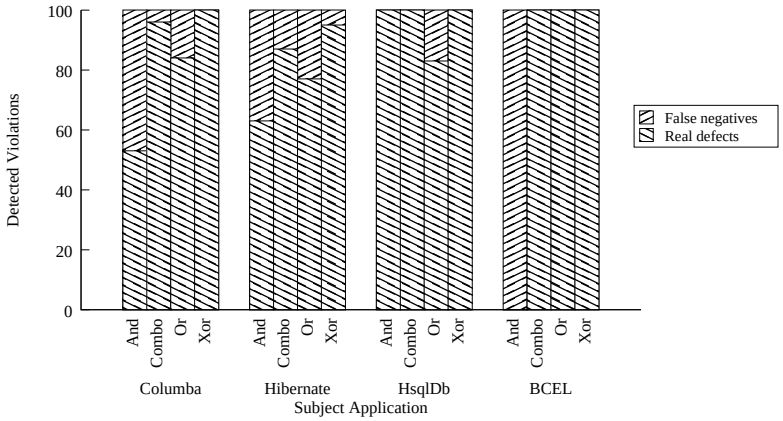


Figure 4.20: Real defects and false negatives among detected violations.

of existing approaches in detecting defects in applications under analysis. Among *Or*, *Xor*, and *Combo* patterns, violations detected using *Or* patterns have a higher number of false negatives compared to the violations detected using *Xor* and *Combo* patterns. For example, in the Columba application, violations detected using *Or* patterns include 8 (16.3%) false negatives compared to 0 (0%) and 2 (4%) false negatives among violations detected using *Xor* and *Combo* patterns, respectively. The primary reason is that *Or* patterns often include partial rules (as shown in Figure 4.17), resulting in false negatives among detected violations. The results show that *Xor* patterns are quite effective in detecting defects compared to all three other pattern formats: *And*, *Or*, and *Combo*. However, *Combo* patterns are also shown to be effective than *Or* patterns and have similar effectiveness as that of *Xor* patterns.

False positives. Figure 4.21 shows the number of false positives among violations detected using patterns in each pattern format. Initially, we expected that *Or* and *Combo* patterns help reduce a high percentage of false positives among violations detected using *And* and *Xor* patterns. However, contrary to our expectation, the number of false positives is high among

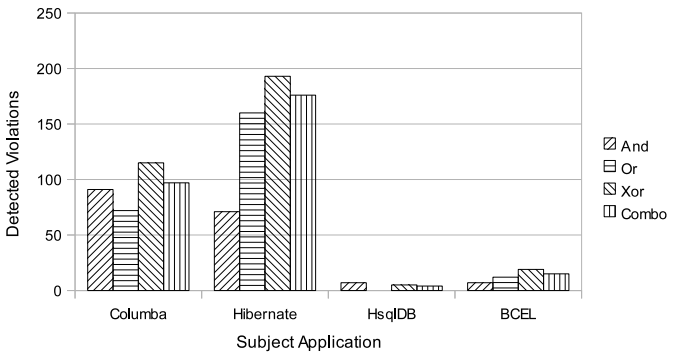


Figure 4.21: False positives among detected violations.

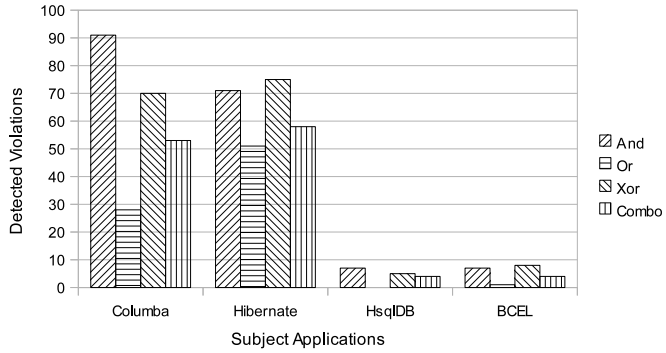


Figure 4.22: False positives among detected violations of API methods with *And* patterns.

violations detected using *Or* and *Combo* patterns. For example, in the Hibernate application, the numbers of false positives are 71, 160, 193, and 176 among violations detected using *And*, *Or*, *Xor*, and *Combo* patterns, respectively.

In our manual analysis of these false positives, we identified an interesting phenomenon: the majority of false positives is related to the API methods that do not have any patterns mined using the *And* pattern format and have new patterns mined using one or more of the *Or*, *Xor*, and *Combo* pattern formats. To illustrate this scenario, we classified all false positives among detected violations into two categories: *FPAnd* and *FPWithOutAnd*. *FPAnd* includes all false positives detected using patterns (*Or*, *Xor*, and *Combo* patterns) related to API methods that have mined patterns using the *And* pattern format. In contrast, *FPWithOutAnd* includes all false positives detected using patterns (*Or*, *Xor*, and *Combo* patterns) related to API methods that *do not* have mined patterns using the *And* pattern format. Figures 4.22 and 4.23 show the classification of false positives for categories *FPAnd* and *FPWithOutAnd*, respectively. Figure 4.22 shows that *Or* patterns help significantly reduce the number of false positives among detected violations compared to *And* and *Xor* patterns. Although *Combo* patterns help reduce false positives, these patterns are not as effective as *Or* patterns. The primary reason is that most of the *Combo* patterns are similar to *Xor* patterns based on our algorithm described in Section 4.3.3.

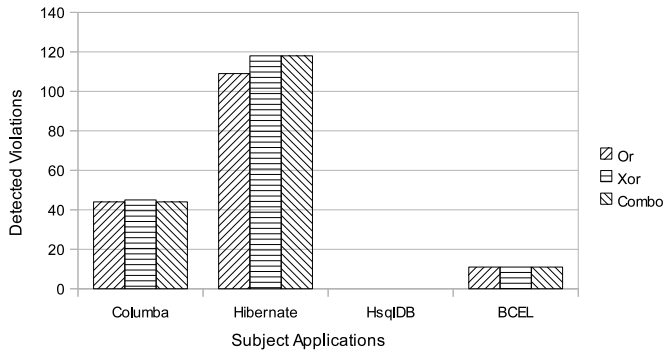


Figure 4.23: False positives among detected violations of API methods without *And* patterns.

Figure 4.23 shows the classification of false positives for API methods without *And* patterns. As shown in the figure, neither *Or* nor *Combo* patterns help reduce false positives among violations detected using *Xor* patterns. The primary reason is that most of these mined patterns are false positives, resulting in false positives among their detected violations. We next summarize our findings.

Summary

In summary, based on our results, comparing to *Or*, *Xor*, and *Combo* patterns, *And* patterns are not effective in detecting defects and result in both false positives and false negatives among detected violations. Although *Xor* patterns are effective in detecting defects, these patterns result in a large number of false positives among detected violations. On the other hand, *Or* patterns are effective in reducing false positives, but, result in false negatives as shown in our results. *Combo* patterns can perform reasonably well with respect to both false positives and false negatives. However, *Or* or *Combo* patterns often result in false-positive patterns for those API methods without any *And* patterns. Therefore, based on our empirical results, the best pattern-mining approach (in terms of reducing both false positives and false negatives) is to first mine *And* patterns for API methods and next mine *Combo* patterns. Among violations detected using *Combo* patterns, the best violation-detection approach is to assign higher priority to the violations of API methods with *And* patterns compared to the violations of API methods without *And* patterns. The primary reason is that the former violations are more likely to be real defects compared to the latter violations.

4.4 Related Work

In the literature, there exist various approaches that mine specifications from static or dynamic traces and use mined specifications for detecting violations in an application under analysis. We first discuss approaches common to both CAR-Miner and Alattin, and next discuss approaches related to CAR-Miner and Alattin, individually.

Engler et al. [38] proposed a general approach for finding defects in C code by applying statistical analysis to rank deviations from programmer beliefs inferred from source code. Their approach requires users to define rule templates. PR-Miner developed by Li and Zhou [75] uses frequent itemset mining to mine programming rules from C code and detect their violations. DynaMine developed by Livshits and Zimmermann [78] uses association rule mining to extract simple rules from software revision histories for Java code and detect defects related to rule violations. PR-Miner or DynaMine may suffer from issues of a high number of false positives since their rule elements are not necessarily associated with program dependencies. CodeWeb [88] mines association rules from source code as library reuse patterns. CodeWeb mines association rules such as application classes inheriting from a library class often create objects of another class. Wasylkowski et al. [131] mines rules that include pairs of API calls and detect violations. Perracotta [140] mines patterns such as $(ab)^*$ and includes techniques for handling imperfect traces. Schäfer et al. [109] mine association rules that describe usage changes in library evolution. All these preceding approaches mine simple association rules that are often not sufficient to characterize complex real rules as shown in our evaluations. In contrast, our approaches include new mining algorithms that can mine more complex rules in the form of sequence association rules or alternative patterns, thereby reducing both false negatives and false positives.

Finally, DeLine and Fähndrich [30] proposed an approach that allows programmers to manually specify resource management protocols that can be statically enforced by a compiler. However, their approach requires manual effort from programmers and also requires the knowledge of the *Vault* specification language to specify domain-specific protocols. In contrast, our approaches do not require any manual effort or the knowledge of any specific specification languages. We next discuss approaches related to CAR-Miner and Alattin, individually.

Detecting exception-handling defects. WN-miner by Weimer and Necula [133] extracts simple association rules of the form "$FC_a \Rightarrow FC_e$", when FC_e is found at least once in exception-handling blocks (i.e., `catch` or `finally` blocks). Their approach mines and ranks these rules based on the number of times FC_e appears after FC_a in normal paths. Due to their ranking criteria, their approach cannot mine rules that include a FC_e function call such as `Connection.rollback`, where FC_e can appear *only* in exception paths. Acharya and Xie [2] later proposed a similar approach for detecting API error-handling defects in C code. Our approach significantly differs and improves upon these previous approaches as we mine sequence association rules of the form "$(FC_c^1 ... FC_c^n) \wedge FC_a \Rightarrow (FC_e^1 ... FC_e^m)$" that can characterize more exception-handling rules. Our approach also addresses the problem of lacking enough supporting samples for these rules in the application under analysis by expanding the data scope to open source repositories through a code search engine.

Our approach is also related to other approaches that analyze exception behavior of programs. Fu and Ryder [43] proposed an exception-flow analysis that computes chains of semantically related exception-flow links across procedures. Our approach uses intra-procedural analysis for constructing exception-flow graphs. The Jex [104] tool statically analyzes exception flow in Java code and provides a precise set of exceptions that can be raised by a function call. We use Jex in our approach to prevent infeasible exception edges in a constructed EFG. Fu et al. [44] present a *def-use*-based approach that helps gather error-recovery code-coverage information. Our approach is different from their approach as our approach detects defects that violate mined rules rather than focusing on coverage of exception-handling code.

Detecting neglected conditions. Another related approach to our Alattin approach is the approach developed by Chang et al. [24] that applies frequent subgraph mining on C code to mine condition rules and to detect neglected conditions. Both Alattin and their approach target at the same type of defects: neglected conditions. Alattin significantly differs from Chang et al.'s approach in three main aspects. First, their approach cannot mine infrequent alternatives. Second, their approach is limited on a much smaller scale of code repositories (in fact, only one project code base) than Alattin, which exploits a CSE to search for relevant code examples from open source code available on the web. Third, the scalability of their approach is heavily limited by its underlying graph mining algorithms, which are known to suffer from scalability issues. In contrast, Alattin uses our new ImMiner algorithm based on frequent itemset mining, being much more scalable.

Williams and Hollingsworth [135] incorporate an API call return value checker for C code, which checks that a value returned by an API call is checked before being used. This type of return-value checking before use falls into a subset of the types of rules being mined by Alattin. Different from their tool, Alattin does not require or rely on version histories, which may not include the types of defect fixing (required by their tool) related to the rules being mined. Acharya et al. [4] developed a tool to mine interface details (such as an API call's return values on success or failure and error flags) from model-checker traces for C code, and then mine interface robustness properties for defect detection. Similar to the tool of Williams and Hollingsworth [135], Acharya et al.'s tool mines only a subset of neglected conditions (e.g., return-value checking before use) mined by Alattin. In addition, as shown by Acharya et al. [4], only the interface details of 22 out of 60 POSIX API functions can be successfully mined by their tool, whereas Alattin exploits a CSE to alleviate the issue by collecting relevant API call usages from the web. Furthermore, these approaches cannot mine alternative patterns targeted by Alattin.

Javert [46] uses a pattern-based specification miner to mine smaller patterns such as $(ab)^*$, called *micro patterns*, and then compose these patterns into larger specifications. Their ap-

proach does not require the user to provide any templates. Similar to their approach, our approach also does not require the user to provide any templates. In future work, we plan to compose larger specifications from the alternative rules mined by Alattin.

Although ours is the first approach to propose new pattern formats such as disjunctive and exclusive-disjunctive pattern formats for mining software engineering data, a few approaches have been proposed (in the data mining research area) that target at mining patterns in these preceding formats for other applications such as market basket analysis [5]. Zhao et al. [142] proposed an approach, called BLOSOM, that targets at mining itemset patterns in four pattern formats: conjunctive, disjunctive, conjunction of disjunctive, and disjunction of conjunctive. In contrast to their BLOSOM approach, our Alattin approach additionally proposes the exclusive-disjunctive pattern format and includes a greedy technique for handling the search space of pattern candidates. In future work, we plan to adopt some properties used by their approach for handling the search space of pattern candidates while mining disjunctive patterns. Nanavati et al. [92] proposed an approach for mining disjunctive association rules. Given a minimum confidence *min_conf*, their approach uses concepts from propositional logic for pruning the association rules that do not have confidence higher than *min_conf*. Shimizu and Miura [111] proposed algorithms for mining disjunctive sequence patterns. In contrast to these two preceding approaches, our Alattin approach targets at mining itemset patterns in disjunctive and exclusive-disjunctive pattern formats.

4.5 Chapter Summary

In this chapter, we presented two approaches, called CAR-Miner and Alattin, that are developed based on our WebMiner framework. We followed a problem-driven methodology in developing these two approaches. More specifically, we empirically investigated problems in the software engineering domain and identified required types of patterns for addressing those problems. We further developed new mining algorithms for mining these required types of patterns, rather than being constrained by available mining algorithms from the data mining community.

In our two approaches, CAR-Miner mines programming rules as sequence association rules, focuses on reducing false negatives by detecting new exception-handling defects in applications under analysis. In contrast to CAR-Miner, Alattin focuses on another sub-direction of reducing false positives among detected violations. In future work, we plan to further expand our research by investigating broader types of problems, patterns, mining algorithms, and defects.

Chapter 5

Improving Software Quality via Dynamic Test Generation

5.1 Introduction

Unit testing is a widely adopted mechanism for assuring high quality of developed software. Unit testing helps detect defects at an early stage, reducing the effort required in fixing those defects. In recent years, there has been a significant improvement in the field of automatic test generation [29,49,59,63,67,70,97,126,128,137]. Automatic test generation, a white-box testing technique, improves software quality by automatically generating test inputs using dynamic program analysis for achieving full or at least high coverage of the code under test; achieving high code coverage with passing test cases often gives high confidence of the quality of the code under test. For example, consider the code example shown in Figure 5.1. There exist approaches such as bounded-exhaustive approaches [137], that can generate sequences upto a bounded length to exhaustively test code under test such as `BinarySearchTree`. A major issue with most of the existing approaches is that these existing approaches can handle only single classes, as shown in their evaluations, where most of the subjects are simple data structures. However, in practice, real-world applications that reuse existing libraries such as .NET libraries are more complex and often include multiple classes.

In particular, code under test reusing existing libraries include nested classes from those libraries, posing additional challenges in effectively generating test inputs. The primary reason is that reuse of API classes or methods from existing libraries increases the amount of code under test, since the test generation techniques requires the knowledge of reused API classes or methods as well. To give more illustrative example, consider the code example shown in Figure 5.2. The code example shows an implementation of Depth First Search algorithm from the QuickGraph [100] application. To automatically generate test inputs for the `UndirectedDFS` class,

```
00:public class BinarySearchTree {
01:   private Node root; //root node
02:   private int size;  //number of nodes in the tree
03:   public void AddToBST(Int32 info_0) {
04:      if (root == null) {
05:         root = new Node(info_0);
06:      } else {
07:         ...
08:      }
09:      size++;
10:   }
11:   public bool RemoveFromBST(Int32 info_0) {
12:      Node parent = null;
13:      Node current = root;
14:      while (current != null) {
15:         ...
16:      }
17:      if (current == null)
18:         return false;
19:      ...
20:      return true;
21:   }}
```

Figure 5.1: An implementation of binary search tree.

a test generation technique not only requires the implementation of `UndirectedDFS`, but also requires the knowledge of other classes such as `IVertexAndEdgeListGraph` or `VertexEventArgs`.

In this chapter, we address the preceding problems from a novel perspective of how knowledge available from existing code bases can be leveraged to assist test generating techniques. In particular, we focus on a specific problem of generating method sequences in testing object-oriented code. These method sequences (in short as *sequences*) create and mutate objects. These sequences help generate target object states (in short as *target states*) of the receiver or arguments of the method under test (MUT). These target states help achieve high coverage by covering `true` or `false` branches in the code under test. As a real example for a target state, the `Compute` MUT is shown in Figure 5.2. The MUT performs a depth-first search on an undirected graph. A target state for reaching Statements 8, 14, or 22 requires that a graph object used in test execution has a non-empty set of vertices and edges. It is quite challenging to automatically generate desirable method sequences due to a large search space of possible method sequences, and the valid sequences constitute only a small portion of the entire search space. A recent empirical study [61] using a random test-generation tool [97] shows that nearly 50% of branches that are not covered in the code under test are due to lack of target sequences. These results show the significance of addressing the problem of generating target sequences for achieving high structural coverage such as branch coverage in object-oriented unit testing. Furthermore, object-oriented programming features such as inheritance, nested classes, and generics pose additional challenges in effectively generating target sequences.

```
//UndirectedDFS: Short form of UndirectedDepthFirstSearch
00:class UndirectedDFS {
01:  IVertexAndEdgeListGraph VisitedGraph; ...
02:  public UndirectedDFS(IVertexAndEdgeListGraph g) {
03:    ...
04:  }
05:  public void Compute(IVertex s) {
06:    //init vertices
07:    foreach(IVertex u in VisitedGraph.Vertices) {
08:      Colors[u]=GraphColor.White;
09:      if (InitializeVertex != null)
10:      InitializeVertex(this, new VertexEventArgs(u));
11:    }
12:    //init edges
13:    foreach(IEdge e in VisitedGraph.Edges) {
14:      EdgeColors[e]=GraphColor.White; }
15:    //use start vertex
16:    if (s != null) {
17:      if (StartVertex != null)
18:        StartVertex(this,new VertexEventArgs(s));
19:      Visit(s); }
20:    // visit vertices
21:    foreach(IVertex v in VisitedGraph.Vertices) {
22:      if (Colors[v] == GraphColor.White) {
23:        if (StartVertex != null)
24:          StartVertex(this,new VertexEventArgs(v));
25:        Visit(v); }
26:    }
27:  }
28:}
```

Figure 5.2: A method under test from the QuickGraph library [100].

To generate target states, there exist four major categories of sequence-generation approaches: bounded-exhaustive [67,137], evolutionary [59,128], random [29,63,97], and heuristic approaches [126]. Bounded-exhaustive approaches generate sequences exhaustively up to a small bound of sequence length. However, generating target states often requires longer sequences beyond the small bound that can be handled by the bounded-exhaustive approaches. On the other hand, evolutionary approaches accept an initial set of sequences and evolve those sequences to produce new sequences that can generate target states. These approaches use *fitness* [77], a metric computed toward reaching a target state, as a guidance for producing new sequences. However, the generation is still a random process and shares the same characteristics as random approaches discussed next.

Random approaches use a random mechanism to combine individual method calls to generate sequences. Although random approaches are shown as effective as systematic approaches theoretically [34], random approaches still face challenges in practice to generate sequences for achieving target states. The reason is that there is often a low probability of generating required

sequences at random for achieving target states. To illustrate the challenges faced by random approaches, we applied a state-of-the-art random approach, called Randoop [97], on the MUT shown in Figure 5.2. Randoop achieved branch coverage of 31.8% (7 out of 22 branches). The reason for low coverage is that the random mechanism of Randoop is not able to generate a graph with vertices and edges.

A heuristic approach for automatic sequence generation is used by a recent approach based on dynamic symbolic execution (DSE) [25, 49, 68, 70], called Pex [126]. Initially, Pex identifies constructors and other methods of a class under test that set values to different fields, hopefully helping generate desirable target state. Using the identified constructors and methods, Pex generates method-sequence skeletons (in short as *skeletons*), which are basically method sequences with symbolic values for primitive types. These skeletons can be considered as a general form of sequences, where symbolic values are used instead of constant values for primitive types. Pex computes concrete values for the symbolic values in skeletons based on constraints collected from branch statements in the code under test. Based on our experience of applying Pex on industrial code bases, we observed that many branches in the code under test are not covered due to lack of proper skeletons. For example, Pex achieved branch coverage of 45.5% (10 out of 22 branches) on the MUT shown in Figure 5.2. The reason for low coverage is that Pex is also not able to generate a graph with vertices and edges.

We developed three approaches to effectively address the problem of generating method sequences: *MSeqGen* [125], *DyGen* [118], and *Seeker* [124]. *MSeqGen* mines method sequences statically from existing code bases and generalizes those sequences to enhance two state-of-the-art test-generation approaches: random testing and dynamic symbolic execution. *MSeqGen* represents the first step towards a new direction of leveraging data mining techniques in addressing a testing problem, serving as a synergy between these two major research areas. In contrast to MSeqGen, *DyGen* mines dynamic traces recorded during program executions and generates regression tests from mined sequences. MSeqGen mines sequences and uses those sequences to assist random or DSE-based approaches, whereas DyGen is a complete approach that generates regression tests from dynamic traces. DyGen also includes additional techniques such as distributed setup for generating tests for large code bases. Although both MSeqGen and DyGen are shown effective compared to related approaches [97, 126], a major issue with both these approaches is that they are not effective when code bases that use classes required for generating target sequences are not available. To address this issue, we developed *Seeker*, that complements MSeqGen and DyGen, and generates method sequences based on implementation. Seeker includes novel techniques that combine static and dynamic analyses. In our evaluations, we compare Seeker and MSeqGen, and discuss their benefits and limitations. Among these three approaches, MSeqGen and DyGen are based on our WebMiner framework, whereas Seeker is independent of our WebMiner framework.

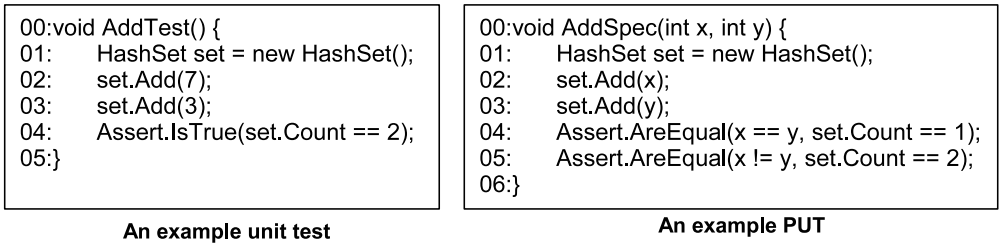


Figure 5.3: A unit test and a parameterized unit test.

5.2 Background

In this section, we present two existing test generation approaches Pex [126] and Randoop [97] that are used as representative approaches for DSE-based and random approaches, respectively, in the rest of the dissertation. We also present dynamic code coverage that refers to the coverage information collected by Pex.

5.2.1 Pex

Pex [126] is a DSE-based approach. Initially, Pex explores an MUT with default or random inputs. During exploration, Pex collects constraints on inputs from the predicates in branch statements. Pex negates collected constraints and uses a constraint solver to generate new inputs that guide future program explorations along different paths. To generate target sequences, Pex uses a simple heuristic-based approach that generates fixed sequences based on static information of constructors and other methods (of classes under test) that set values to member fields, hopefully helping produce desired object states.

5.2.2 Parameterized Unit Test (PUT)

PUTs [127] are a recent advance in software testing. Unlike conventional unit tests that do not accept any parameters, PUTs accept parameters. In particular, separate two major concerns. First, PUTs allow to describe the behavior of method under test for all test arguments. Second, conventional unit tests can be regenerated by instantiating PUTs. Figure 5.3 shows an example unit test and a PUT. As shown, a unit test does not accept any parameters, whereas PUT accepts two parameters `x` and `y`.

5.2.3 Dynamic Code Coverage

Dynamic code coverage or block coverage refers to the coverage information collected by Pex. Since Pex performs code instrumentation dynamically at runtime, Pex only knows about the code that was already executed. In addition to code loaded from binaries on the disk, the .NET

environment in which we perform our experiments allows the generation of additional code at runtime via *Reflection-Emit*.

5.2.4 Randoop

Randoop [97] is a random approach that generates sequences incrementally by randomly selecting method calls. For each randomly selected method call, Randoop uses random values and previously generated sequences for primitive and non-primitive arguments, respectively. For each generated test input, Randoop avoids reusing or extending previously generated sequences that throw uncaught exceptions.

5.3 Formal Definitions

We next present definitions used in the rest of this chapter. We also present a formal definition for the problem of generating target sequences. Consider that $CSet$ and $MSet$ represent all classes and methods, respectively, of the code under test and other referenced assemblies. Consider that $Prim$ and $PrimVal$ represent the set of all primitive types such as `int` or `bool`, and primitive values, respectively. Each method $M_i \in MSet$ is represented as a tri-tuple $\langle recv(M_i), arg(M_i), ret(M_i)\rangle$, where $recv(M_i) \in null \vee CSet$, $arg(M_i) = \{a_i^1, a_i^2, \ldots, a_i^n\}$: $\forall 1 \leq j \leq n \;\; a_i^j \in CSet \cup Prim$, and $ret(M_i) \in CSet \cup Prim \cup \{void\}$ represent the receiver, argument, and return values, respectively, of M_i. Since $a_i^j \in CSet \cup Prim$, the arguments of M_i can be either primitive or non-primitive. Here, the receiver object can be `null` for static methods.

Definition 5.3.1 Method Sequence (MCS). A method sequence is a sequence of method calls $(M_1, M_2, \ldots, M_r)$, where $\forall 1 \leq i \leq r,\ M_i \in MSet \wedge (recv(M_i) = null \vee ret(M_s)$ where $1 \leq s < i) \wedge (\forall 1 \leq j \leq |arg(M_i)|,\ a_i^j \in PrimVal \vee ret(M_k)$ where $1 \leq k < i)$.

For each method call M_i in MCS, the receiver object $recv(M_i)$ should be `null` (for static method calls) or the return object $ret(M_s)$ of another method call M_s that precedes M_i within the sequence. Furthermore, each M_i in MCS can have either primitive or non-primitive arguments. For primitive arguments, the preceding definition describes that MCS should include values such as `true` for the `bool` type. For non-primitive arguments, they should be return values of some preceding method calls within the sequence. For example, consider the following sample sequence.

```
00: VEProvider s0 = new VEProvider();
01: Vertex s1 = new Vertex(0);
02: AdjacencyGraph ag = new AdjacencyGraph(s0, false);
```

```
03: ag.AddVertex(s1);
04: ag.AddEdge((IVertex)s1, (IVertex)s1);
05: UDFSAlgorithm ud = new UDFSAlgorithm(ag);
06: ud.Compute((IVertex)null);
```

In the preceding sequence, the non-primitive argument `s1` of `AddVertex` in Statement 3 is the return value of another preceding method call `new Vertex()` in Statement 1. Our definition ensures that method sequences are self-contained, and executable code can be generated directly from those sequences.

Definition 5.3.2 Class Sequence (CCS). A class sequence is a sequence of method calls $(M_1, M_2, \ldots, M_r)$, where $\forall 1 \leq i \leq r,\ M_i \in MSet\ \wedge\ (M_1\ is\ a\ constructor\ \wedge\ recv(M_s) = ret(M_1)$ where $2 \leq s \leq r)$.

In contrast to MCS, in CCS, the receiver objects of all method calls are the same. Each CCS starts with a constructor and all other methods are invoked on the object created by the first method call. An example CCS is shown below:

```
00: VEProvider s0; Vertex s1;
01: AdjacencyGraph ag = new AdjacencyGraph(s0, false);
02: ag.AddVertex(s1);
03: ag.AddEdge((IVertex)s1, (IVertex)s1);
```

Definition 5.3.3 Skeleton (SKT). A skeleton is a sequence of method calls $(M_1, M_2, \ldots, M_r)$, where $\forall 1 \leq i \leq r,\ M_i \in MSet\ \wedge\ (recv(M_i)\ =\ null \vee ret(M_s)$ where $1 \leq s\ <\ i)\ \wedge\ (\forall 1 \leq j \leq |arg(M_i)|,\ a_i^j\ \in\ Prim\ \vee ret(M_k)$ where $1 \leq k < i)$.

The definition for SKT is similar to MCS, except that the values of primitive-type arguments are not required (represented by $a_i^j\ \in\ Prim$) for SKT. Consider $M_u \in MSet$ as a method under test. Consider that a subset $npa_u \subseteq arg_u$ of all the arguments are non-primitive arguments. Consider that $anp_u = npa_u \cup \{rec_u\}$ denotes all non-primitive objects of M_u.

Definition 5.3.4 Object state (OS). An object state of a non-primitive object $np_i \in anp_u$ is represented as an $MCS\ (M_1, M_2, \ldots, M_r)$ where $np_i = ret(M_k)$, where $1 \leq k \leq r$.

We represent an object state as a method sequence, following Henkel and Diwan [62], who use a similar representation for mapping Java classes to algebras.

Definition 5.3.5 Object state set (OSS). An object state set of method under test M_u is a set of method sequences for all non-primitive objects $np_i \in anp_u$.

Consider that a test collection T includes a set of test inputs $\{t_1, t_2, \ldots, t_h\}$ generated for M_u. Each test input t_i includes an OSS and covers `true` or `false` branches of conditional statements in the method under test M_u.

Definition 5.3.6 Target branch (TB). A target branch is a `true` or `false` branch (of a conditional statement in M_u) that is not covered by any test input $t_i \in T$.

Consider `Compute` in Figure 5.2 as M_u. Consider the following test input as the only test input (in the test collection T) that is generated for `AddEdge`.

```
00: AdjacencyGraph ag = new AdjacencyGraph();
01: UndirectedDFS udfs = new UndirectedDFS(ag);
02: ag.Compute(null);
```

The `true` branch of Statement 7 in Figure 5.2 is considered as a target branch, since the preceding test input cannot cover this branch. We next present our approaches that focus on generating method sequences for covering the target branches.

5.4 MSeqGen: Method Sequence Generation via Mining Source Code

5.4.1 Motivation

Since existing approaches are not effective in generating sequences, in this section, we propose a novel approach, called *MSeqGen*. MSeqGen addresses the significant problem of sequence generation from a novel perspective of how these method calls are used together in practice. The key insight of MSeqGen is that the information of how the method calls are used in practice helps generate sequences that can achieve target states. To gather such usage information of method calls, our MSeqGen approach mines code bases that are already using the object types such as receiver or argument object types of the MUT. For a MUT, these code bases include source code of the application that the MUT belongs to and test code for that application. In addition, code bases also include other applications using the receiver or argument object types of the MUT available in both proprietary and open source code (available on the web).

To the best of our knowledge, MSeqGen is the first one that addresses the significant problem of sequence generation by leveraging the information of how method calls are used in practice. MSeqGen mines code bases to extract sequences related to receiver or argument object types of a MUT. MSeqGen uses extracted sequences to assist both random and DSE-based approaches in achieving higher structural coverage. More specifically, MSeqGen addresses three major issues in extracting sequences from code bases. First, code bases are often large and complete analysis of these code bases can be prohibitively expensive. To address this issue, MSeqGen searches

for code portions relevant to receiver or argument object types of the MUT and analyzes those relevant code portions only (*search* phase of our WebMiner framework). Second, constant values in sequences extracted from code bases can be different from values required to achieve target states. To address this issue, our approach converts extracted sequences into skeletons by replacing constant values for primitive types with symbolic values. We refer to this process of converting sequences into skeletons as *sequence generalization*. Third, extracted sequences individually may not be useful in achieving target states. Our approach addresses this issue by combining extracted sequences randomly to generate new sequences that may produce target states.

In summary, MSeqGen makes the following major contributions:

- The first approach that leverages the information of how method calls are used in practice to address the significant problem of sequence generation in object-oriented unit testing.
- A technique for generalizing extracted sequences (i.e., converting sequences into skeletons) to assist DSE-based approaches. Generalization helps address issues where constant values in extracted sequences are different from values required to achieve target states.
- A technique for generating new sequences by randomly combining extracted sequences. These new sequences try to address the issue where extracted sequences individually are not sufficient to achieve target states.
- An implementation of the MSeqGen approach and its evaluation upon two state-of-the-art industrial testing tools: Randoop and Pex. Both Pex and Randoop were shown to find serious defects in industrial code bases [97, 126]. MSeqGen represents a significant, successful step towards addressing complex testing problems in industrial practice, targeting at complex desirable sequences from multiple classes rather than sequences on single classes such as data structures heavily focused by previous approaches [59, 128].
- Empirical results from two evaluations show that MSeqGen can effectively assist state-of-the-art random and DSE-based approaches in achieving higher branch coverage. Using MSeqGen, we show that a random approach achieves 8.7% (with a maximum of 20.0% for one namespace) higher branch coverage and a DSE-based approach achieves 17.4% (with a maximum of 22.5% for one namespace) higher branch coverage than without using our approach. Such an improvement is significant since the branches that are not covered by these state-of-the-art approaches are generally quite difficult to cover.

5.4.2 Example

We next explain our MSeqGen approach with an illustrative example shown in Figure 5.2. The figure shows a MUT, called `Compute`, taken from the QuickGraph library [100]. The MUT

requires two non-primitive objects: `IVertexAndEdgeListGraph` and `IVertex`. The MUT requires an object of `IVertexAndEdgeListGraph` (which represents a graph) since the constructor of the receiver object of the MUT has the argument of type `IVertexAndEdgeListGraph`. The MUT accepts a vertex in the graph as argument and computes a depth-first search of the graph. To achieve high structural coverage of the MUT, the minimal requirement is that the graph object should include vertices and edges. We used both Randoop and Pex to generate unit tests for the MUT. Randoop achieved branch coverage of 31.8% (7 of 22). The reason for low branch coverage is that the random mechanism of Randoop is not able to generate a graph object with vertices and edges.

To generate test inputs using Pex, we created a PUT that includes `UndirectedDFS` as a parameter. Since the constructor of `UndirectedDFS` accepts an interface `IVertexAndEdgeListGraph` as argument, Pex can automatically generate a new class implementing the `IVertexAndEdgeListGraph` interface. However, such a new class may not support the (implicit) contracts associated with the interface implementation. Therefore, we provided minimal assistance to Pex by describing which implementing classes can be used for interfaces. For example, we feed to Pex the information that it can use the `AdjacencyGraph` class as an implementing class for the `IVertexAndEdgeListGraph` interface. Pex achieved branch coverage of 45.5% on the MUT. Although Pex achieved higher branch coverage than Randoop, the coverage is still low (only 45.5%). Similar to Randoop, Pex was not able to generate a graph object with vertices and edges.

We next describe how MSeqGen can assist Randoop and Pex by extracting sequences from existing code bases. We need sequences for objects of three classes[1]: `UndirectedDFS`, `IVertexAndEdgeListGraph`, and `IVertex`. We need a sequence for an object of the `UndirectedDFS` class to construct a desirable receiver object state. We also need sequences for objects of classes implementing the `IVertexAndEdgeListGraph` and `IVertex` interfaces. We collected a set of applications (code bases of 3.9 MB of .NET assembly code) from an open source C# repository[2] that reuse classes of the QuickGraph library. MSeqGen analyzes these code bases and extracts sequences that produce objects of these classes. MSeqGen extracted 5 sequences for the `AdjacencyGraph` class that implements the `IVertexAndEdgeListGraph` interface, and 11 sequences for the `Vertex` class that implements the `IVertex` interface. Figures 5.4 and 5.5 show two example class sequences (refer to section 5.3 for the definition of class sequence) for creating objects of the `AdjacencyGraph` and `Vertex` classes. The sequence for `AdjacencyGraph` satisfies our minimal requirement that the resulting graph should include vertices and edges. It is challenging to generate these sequences automatically, especially due to the large number of possible sequence combinations. In contrast, MSeqGen can easily extract such sequences from code bases.

[1]We use classes to collectively denote both classes and interfaces.

[2]`http://www.codeplex.com/`

```
01: VertexAndEdgeProvider vo; //requires as input
02: bool bVal; //requires as input
03: AdjacencyGraph agObj = new AdjacencyGraph(vo,bVal);
04: IVertex source = agObj.AddVertex();
05: IVertex target = agObj.AddVertex();
06: IVertex vertex3 = agObj.AddVertex();
07: IEdge edgObj1 = agObj.AddEdge(source,target);
08: IEdge edgObj2 = agObj.AddEdge(target,vertex3);
09: IEdge edgObj3 = agObj.AddEdge(source,vertex3);
```

Figure 5.4: A class sequence (CCS) for producing an `AdjacencyGraph` object with vertices and edges.

```
AdjacencyGraph agObj; //requires as input
IVertex vObj = agObj.AddVertex();
```

Figure 5.5: A class sequence (CCS) for producing an `IVertex` object.

One issue with extracted sequences is that these sequences can include additional classes. For example, Statement 3 of Figure 5.4 shows that the sequence requires another object of the class `VertexAndEdgeProvider`. MSeqGen automatically identifies such additional classes and gathers sequences that produce objects of these classes. MSeqGen extracted one sequence for the `VertexAndEdgeProvider` class from existing code bases. Section 5.4.3 presents more details on how MSeqGen addresses challenges for extracting these sequences.

We next use Randoop with additional sequences extracted by MSeqGen. Randoop generated new test inputs incorporating sequences extracted by MSeqGen. The new test inputs achieved branch coverage of 86.4% (19 of 22) of the `Compute` method. When we use our extracted sequences to assist Pex, Pex also achieved the same branch coverage for the `Compute` method. The remaining not-covered branches are due to lack of event handlers that need to be registered with `UndirectedDFS`. This example describes our MSeqGen approach and highlights the significance of using sequences from existing code bases in achieving higher branch coverage with both random and DSE-based approaches.

5.4.3 Approach

Figure 5.6 shows a high-level overview of our MSeqGen approach that is developed based on our WebMiner framework. MSeqGen includes an additional phase, called *sequence generalization*, along with the three major phases of our WebMiner framework: *search*, *analyze*, and *apply*. In particular, MSeqGen accepts an application under test and identifies classes and interfaces, declared or used by the application under test. These applications under test can also be libraries. We refer to extracted classes for which class sequences need to be collected as target classes, denoted by $\{TC_1, TC_2, ..., TC_m\}$. MSeqGen also accepts a set of existing code bases, denoted by $\{CB_1, CB_2, ..., CB_n\}$, that already use these target classes. In our prototype implemented for the MSeqGen approach, these code bases are in the form of .NET assemblies. Initially, in the *search* phase, MSeqGen searches for relevant method bodies by using target

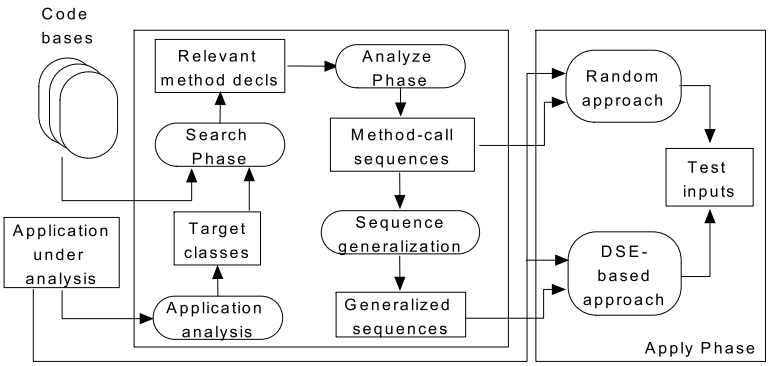


Figure 5.6: Overview of our MSeqGen approach.

```
00:public void Sort(VertexAndEdgeProvider vo) {
01:   AdjacencyGraph g = new AdjacencyGraph(vo, true);
02:   Hashtable iv = new Hashtable();
03:   int i = 0;  //adding vertices
04:   IVertex a = g.AddVertex();
05:   iv.Add(a);
06:   IVertex b = g.AddVertex();
07:   iv.Add(b);
08:   IVertex c = g.AddVertex();
09:   iv.Add(c);
10:   g.AddEdge(a,b); //adding edges
11:   g.AddEdge(a,c);
12:   g.AddEdge(b,c);
13:   //TSAlgorithm: TopologicalSortAlgorithm
14:   TSAlgorithm topo = new TSAlgorithm(g);
15:   topo.Compute(); ...
16:}
```

Figure 5.7: A relevant method body for classes `AdjacencyGraph`, `VertexAndEdgeProvider`, `Hashtable`, and `TopologicalSortAlgorithm`.

classes as keywords. In the *analyze* phase, MSeqGen constructs control-flow graphs for these method bodies and extracts class sequences that produce objects of target classes. MSeqGen extracts class sequences by traversing these control-flow graphs. In the *apply* phase, these extracted class sequences are used to assist random and DSE-based approaches. For DSE-based approaches, MSeqGen includes an additional phase, called *sequence generalization*, before the *apply* phase. In the *sequence generalization* phase, MSeqGen converts extracted class sequences into skeletons by replacing constant values for primitive types with symbolic values. MSeqGen does not include the *mine* phase, since MSeqGen attempts to reuse all the class sequences extracted from existing code bases. We next explain each phase of MSeqGen in detail using the illustrative example shown in Figure 5.2.

Search Phase

MSeqGen uses the *search* phase, since code bases are often large and analyzing complete code bases can be prohibitively expensive. To avoid analyzing complete code bases, MSeqGen uses a keyword search to identify relevant method bodies including target classes. In particular, we use a text-based search, where the text is derived by decompiling .NET assemblies taken as inputs. We consider that a method body is relevant to a target class TC_j, if the method body includes the name of the TC_j target class. For example, we use `AdjacencyGraph` as a keyword and search for method bodies including that keyword. Figure 5.7 shows an example method body including the `AdjacencyGraph` keyword. Since our search is primarily a text-based search, the *search* phase also returns irrelevant method bodies such as method bodies that include `AdjacencyGraph` as a variable name or a word in comments. We filter out such irrelevant method bodies in subsequent phases.

We use intra-procedural analysis to analyze only such relevant method bodies. We use intra-procedural analysis since this analysis is more scalable than inter-procedural analysis. Although intra-procedural analysis is less precise than inter-procedural analysis, we address this precision issue by using an iterative strategy explained in subsequent sections.

Analyze Phase

We next analyze each relevant method body statically and construct a control-flow graph (CFG). Our CFG includes four types of statements: method calls, object creations, typecasts, and field accesses. The rationale behind choosing these statements is that these statements result in generating objects of target classes. While constructing a CFG, we identify the nodes (in the constructed CFG) that produce the target classes such as `AdjacencyGraph` and mark them as *nodes of interest*. For example, the node corresponding to Statement 1 in Figure 5.7 is marked as a node of interest for the target class `AdjacencyGraph`. We also filter out irrelevant method bodies identified during the code searching phase if their related CFGs do not contain any nodes of interest.

We next extract class sequences from a CFG using nodes of interest. For each node of interest related to a target class TC_j, we gather a path from the node of interest to the end of the CFG. In the case of loops, we consider the nodes inside a loop as a group of nodes that is executed either once or not. Considering these nodes once can help identify the sequence inside the loop. We also annotate these nodes to store the additional information that these nodes (and their associated method calls) exist inside loops. This additional information is used in subsequent phases while generating code based on extracted sequences.

Often, an extracted sequence can include a few method calls that are unrelated to the target class TC_j. To extract class sequences from extracted sequences, we use data-dependency

analysis to filter out such unrelated method calls from the extracted sequence. We start with the method call (in short as *base method call*) associated with a node of interest and filter out method calls that do not share the same receiver object as the base method call. Our data-dependency analysis results in a class sequence that creates and mutates an object of a target class TC_j. For example, Figure 5.4 shows a sequence gathered from the code example in Figure 5.7. MSeqGen extracts several classes sequences for different classes from the same code example. For example, if the set of target classes also includes classes `Hashtable` and `TSAlgorithm`, MSeqGen automatically extracts one class sequence for each of these classes as shown below from the code example.

```
Sequence for Hashtable:
    IVertex a,b,c; //requires as input
    Hashtable iv = new Hashtable();
    iv.Add(a);
    iv.Add(b);
    iv.Add(c);
Sequence for TSAlgorithm:
    AdjacencyGraph g; //requires as input
    TSAlgorithm tsObj = new TSAlgorithm(g);
    tsObj.compute();
```

One issue with extracted class sequences is that these sequences can include additional non-primitive types. For example, the sequence for `AdjacencyGraph` (shown in Figure 5.4) requires non-primitive type `VertexAndEdgeProvider`. To achieve target states, we need new sequences for generating these additional non-primitive types. In principle, call sites in code bases including sequences for a TC_j target class also include the sequences for generating related additional non-primitive types. However, in practice, often these call sites do not include sequences for these additional non-primitive types due to two factors. (1) A sequence for an additional non-primitive type is available in another method body and is not found by our approach as it uses intra-procedural analysis for extracting sequences. (2) A sequence for an additional non-primitive type does not exist in the current code base CB_i (such as a library) and expects a reusing application to provide a necessary sequence.

We address this issue by extracting new class sequences for additional non-primitive types by using an iterative strategy. More specifically, we first extract class sequences for the initial set of target classes and collect all additional classes for which new class sequences need to be extracted. We next extract class sequences for these additional classes and collect more new additional classes. We repeat this process either till no new additional classes are collected or we reach a fixed number of iterations accepted as a configuration parameter, denoted by *NUM_ITERATIONS*. A high value for *NUM_ITERATIONS* can help collect more sequences;

A. Class Definition:
```
00:class MyClass {
01:   private int testMe;
02:   private String ipAddr;
03:}
```

B. MUT:
```
00:public void Mut1(MyClass mc, String IPAddress) {
01:   if(mc.getTestMe() > 100) {
02:      if(IsAValidIPAddress(IPAddress)) { ... }
03:   }
04:}
```

C. Method-call sequence (MCS):
```
00:MyClass mcObj = new MyClass();
01:mcObj.SetTestMe(10);
02:mcObj.SetIpAddr("127.0.0.1");
```

D. Skeleton:
```
00:int symvar = *, string ipaddr = *;
01:MyClass mcObj = new MyClass();
02:mcObj.SetTestMe(symvar);
03:mcObj.SetIpAddr(ipaddr);
```

Figure 5.8: An illustrative example for method sequence generalization.

however, a high value can require more time for collecting those sequences. In our approach, we use five as the value of *NUM_ITERATIONS*, which is set based on our initial empirical experience.

Sequence Generalization

Before the *apply* phase, MSeqGen generalizes class sequences into skeletons (SKT) to address an issue that constant values in extracted sequences can be different from values required to achieve target states. We refer to the process of converting sequences into skeletons (which are sequences with symbolic values instead of concrete values for primitive types) as *sequence generalization*. For example, consider a simple MUT and an example sequence (denoted as MCS) shown in Figures 5.8a to 5.8c, respectively. The sequence cannot directly achieve the `true` branch of the MUT since the value of `testMe` is set to 10. To address this issue, we generalize extracted sequences. More specifically, we replace constant values of primitive types in extracted sequences with symbolic values. Figure 5.8d also shows the skeleton, where a symbolic variable `symvar` of type `int` is taken as input for the sequence. This `symvar` variable replaces the constant value 10 in the MCS. When this skeleton is used along with a DSE-based approach, the DSE-based approach initially generates a concrete random value for the `symvar` symbolic variable and gathers the constraint (> 100) in the MUT through dynamic execution.

The DSE-based approach next solves the constraint to generate another concrete value for `symvar` such as 200 that satisfies the gathered constraint.

Although DSE-based approaches are effective in practice, it is challenging for these approaches to generate concrete values for variables that require complex values such as doubles, IP addresses, or URLs. In such cases, constant values in extracted sequences are useful in quickly covering those related branches such as the `true` branch in Statement 2 (Figure 5.8b) of the MUT. To address this issue, we preserve constant values in sequences along with the newly introduced symbolic values by using a symbolic `boolean` value as a switch between symbolic and constant values.

Apply Phase

In the *apply* phase, MSeqGen uses extracted class sequences to assist random and DSE-based approaches. For DSE-based approaches, MSeqGen uses generalized sequences. However, in some cases, these extracted class sequences individually are not sufficient to achieve target states. MSeqGen tries to address this issue by generating new method sequences (MCS) from extracted class sequences by combining extracted class sequences randomly. For example, consider two target classes T_i and T_j, where T_j requires an object of T_i and a MUT requires an object of T_j. Consider that MSeqGen identified two method bodies, denoted by MD_1 and MD_2, in code bases relevant to both T_i and T_j. Consider that MSeqGen extracted class sequences S_i^1 and S_j^1 for target classes T_i and T_j from MD_1, respectively. Similarly, MSeqGen extracted class sequences S_i^2 and S_j^2 for target classes T_i and T_j from MD_2, respectively. The target class T_i has class sequences S_i^1 and S_i^2, and the target class T_j has class sequences S_j^1 and S_j^2. Given these sequences, MSeqGen can generate some or all of four different combinations of these sequences for generating objects of T_j. These new sequences may further help achieve target states in the MUT.

5.4.4 Evaluation

We conducted three different evaluations to show the effectiveness of our MSeqGen approach. In our evaluations, we used two popular .NET applications: QuickGraph [100] and Facebook [41]. Our empirical results show that MSeqGen handles large code bases and extracts sequences that can help achieve target states. Our empirical results also show that MSeqGen can effectively assist random and DSE-based approaches in achieving higher branch coverage. The details of subjects and results of our evaluation are available at `http://research.csc.ncsu.edu/ase/projects/mseqgen/`. All experiments were conducted on a machine with 1.6GHz Xeon processor and 1GB RAM. We next present research questions addressed in our evaluations.

Research Questions

In our evaluations, we address the following research questions.

- RQ1: Can MSeqGen handle large code bases in gathering class sequences for target classes of subject applications?
- RQ2: How much higher code coverage of the code under test is achieved by a random approach with the assistance of MSeqGen compared to without the assistance of MSeqGen?
- RQ3: How much higher code coverage of the code under test is achieved by a DSE-based approach with the assistance of MSeqGen compared to without the assistance of MSeqGen?

Subject Applications

We used two popular .NET applications for evaluating MSeqGen: QuickGraph [100] and Facebook [41]. QuickGraph is a C# graph library that provides various directed/undirected graph data structures. QuickGraph also provides algorithms such as depth-first search, breadth-first search, and A* search [28]. QuickGraph includes 165 classes and interfaces with 5 KLOC. Facebook is a popular social network website that connects people with friends and others whom they work, study, and live around. In our evaluation, we use a Facebook developer toolkit that provides APIs necessary for developing Facebook applications. The Facebook developer toolkit includes 285 classes and interfaces with 40 KLOC.

RQ1: Gathering Class Sequences

We next address the first research question on whether MSeqGen can handle large code bases in gathering sequences for target classes of the QuickGraph and Facebook applications. For QuickGraph and Facebook, we use code bases including 3.85 MB and 5 MB of .NET assembly code, respectively. MSeqGen extracted 167 sequences for QuickGraph with a maximum length of 12 method calls for the `AdjacencyGraph` class. MSeqGen took 5.2 minutes for analyzing code bases related to QuickGraph. For Facebook, MSeqGen extracted 355 sequences with a maximum length of 51 method calls for the `Hashtable` class. Although the sequence extracted for `Hashtable` is long, this sequence includes method calls such as `Add` for multiple times. MSeqGen took 4.5 minutes for analyzing code bases related to Facebook and to gather these sequences. Our results show that MSeqGen can mine large code bases for gathering sequences to help achieve target states.

RQ2: Assisting Random Approach

We next address the second research question on whether MSeqGen helps increase branch coverage achieved by a state-of-the-art random approach, called Randoop [97]. To address this research question, we first run Randoop on QuickGraph and Facebook applications, and generate test inputs. Randoop generates test inputs in the form of sequences of method calls. We execute generated test inputs and measure branch coverage using a coverage measurement tool, called NCover[3]. This measured coverage forms a baseline for comparing Randoop with and without the assistance from MSeqGen. In our evaluation, we use default configurations provided by the Randoop developers. For each namespace of the subject application, we ran Randoop for a maximum of 130 seconds.

To assist Randoop with our extracted sequences, we synthesize static method bodies that include our gathered sequences and return objects of target classes of our subject applications. For example, if a target class TC_j has four sequences, we synthesize four static method bodies where each method body returns an object of TC_j by executing a gathered sequence for TC_j. If a sequence for TC_j requires other objects of non-primitive or primitive types (whose values are not known in gathered sequences due to static analysis), we add those non-primitive and primitive types as arguments for the method bodies. For primitive types, Randoop randomly generates some values. For non-primitive types, Randoop randomly generates a new sequence or selects some other method body (synthesized by MSeqGen) that produces that non-primitive type. We gather newly generated test inputs that include the method bodies synthesized by MSeqGen and add these new test inputs to existing tests to measure the increase in the branch coverage.

Table 5.1 shows the results of our evaluation with both subject applications. The table shows the results for all namespaces of the subject applications. As we include test code available with subject applications in code bases used for extracting sequences, we show branch coverage achieved by the test code alone in Column "T". Column "R" shows branch coverage achieved by Randoop. Column "R + M" shows branch coverage achieved by Randoop with the assistance of our MSeqGen approach. Column "Increase in Branch Coverage" shows additional branch coverage achieved with the assistance from our MSeqGen approach. As shown in our results, "R + M" achieved higher coverage than Randoop and test code (except for namespaces `facebook` and `facebook.Utility`). There are two primary reasons for lower coverage of "R + M" for these two namespaces: the random mechanism of Randoop and limitations of our current implementation. First, due to the random mechanism used by Randoop, various method calls used in test code that contributed to higher coverage achieved by the test code are not used by Randoop in generating test inputs. Second, our current implementation does not handle

[3] `http://www.ncover.com/`

Table 5.1: Evaluation results showing higher branch coverage achieved by Randoop with the assistance of MSeqGen. `T: Test code, R: Randoop, M: MSeqGen`

Application	# of **classes**	**Test Code T**	**Random R**	**R + M**	**% Increase in Branch coverage**
QuickGraph.Algorithms	104	18.4	63.3	63.3	-
QuickGraph.Algorithms.Search	11	40.3	33.3	47.6	**14.3**
QuickGraph.Algorithms.ShortestPath	4	0	29.3	30.2	0.9
QuickGraph.Algorithms.Visitors	11	0	86.4	86.4	-
QuickGraph.Collections	19	11.2	74.0	83.3	9.3
QuickGraph.Exceptions	3	40.0	100.0	100.0	-
QuickGraph.Predicates	9	8.6	43.1	48.3	5.2
QuickGraph.Providers	1	100.0	80.0	100.0	**20.0**
QuickGraph.Representations	3	43.1	35.1	49.0	**13.9**
facebook	25	48.9	14.0	23.3	9.3
facebook.Components	3	0	30.7	30.7	-
facebook.desktop	14	0	18.5	21.0	2.5
facebook.Forms	4	0	11.1	11.1	-
facebook.Properties	1	31.3	37.5	37.5	-
facebook.Schema	216	6.1	20.8	24.8	4.1
facebook.Types	1	0	100.0	100.0	-
facebook.Utility	8	49.1	22.6	37.7	**15.1**
facebook.web	12	0	3.3	4.5	1.2
AVERAGE					**8.7**

several features such as inheritance or C# generics. Therefore, our implementation could not capture some sequences due to their use of these features. In future work, we plan to extend our implementation to support these features. Overall, our results show that there is a considerable increase of 8.7% on average[4] (with a maximum of 20%) in branch coverage achieved by Randoop with assistance from MSeqGen.

We next provide examples to describe scenarios where MSeqGen can assist random approaches. We also describe scenarios where MSeqGen cannot assist random approaches. We use a MUT, called `AddEdge`, in the `BidirectionalGraph` class of the `QuickGraph.Representations` namespace (shown in Figure 5.9). Although Randoop generated three test inputs (in the form of sequences) for the `AddEdge` MUT, Randoop achieved low branch coverage of 40.0% (2 out of 5 branches). The reason for not achieving high coverage for the `AddEdge` MUT is that the `AddEdge` MUT requires a specific receiver object state. To reach Statement 8 of the MUT, the `VertexInEdges` field should include the new vertices represented by `src` and `tg` that are passed as arguments. With the sequences extracted by MSeqGen, Randoop achieved a branch coverage of 80.0% (4 out of 5 branches). As our sequences are extracted from code bases that include usage scenarios on how these method calls are used in real practice, our sequences helped achieve high coverage for the `AddEdge` MUT.

[4]We compute average from those namespaces that have a non-zero increase in the branch coverage

```
00:class BidirectionalGraph { ...
01:   public IEdge AddEdge(IVertex src, IVertex tg) {
02:      // look for the vertex in the list
03:      if (!VertexInEdges.ContainsKey(src))
04:         throw new VertexNotFoundException ("Could not find source");
05:      if (!VertexInEdges.ContainsKey(tg))
06:         throw new VertexNotFoundException ("Could not find target");
07:      // create edge
08:      IEdge e = base.AddEdge(src, tg);
09:      VertexInEdges[target].Add(e);
10:      return e;
11:   }
12:}
```

Figure 5.9: A MUT `AddEdge` in the `BidirectionalGraph` class of QuickGraph.

Although Randoop achieved higher branch coverage with the assistance from MSeqGen, the test inputs generated by Randoop did not cover the `true` branch of Statement 5 to reach Statement 6. The reason is that our sequences do not include a usage scenario where the `AddEdge` MUT is invoked with one vertex in `VertexInEdges` and the other vertex not in `VertexInEdges`. Such usage scenarios rarely exist in code bases that are used for extracting sequences as these usage scenarios are related to testing the MUT for negative cases rather than reusing the MUT in real practice. However, a more systematic approach such as a DSE-based approach can cover such not-covered branches with the assistance from MSeqGen.

RQ3: Assisting DSE-based Approaches

We next address the third research question on whether MSeqGen can help increase branch coverage achieved by a DSE-based approach. To address this research question, we use a state-of-the-art DSE-based approach called Pex [126]. Pex accepts PUTs as input and generates conventional unit tests from these PUTs using DSE. As PUTs are not available with our subject applications, we generated PUTs for each public method in our subject applications using the *PexWizard* tool. PexWizard is a tool provided with Pex and this tool automatically generates PUTs for each public method in the application given as input. A PUT generated for the `Compute` MUT (Figure 5.2) is shown below.

```
00:[PexMethod]
01:public void Compute01(
02:    [PexAssumeUnderTest]UndirectedDFS target,
03:    [PexAssumeUnderTest]Vertex s) {
04: target.Compute(s);
05: Assert.Inconclusive("this test has to be reviewed");
06:}
```

Table 5.2: Evaluation results showing higher branch coverage achieved by Pex with the assistance of MSeqGen. `# C: number of classes, P: Pex, M: MSeqGen`

Application	**# C**	**P**	**P + M**	**Increase %**
QuickGraph.Algorithms	104	8.2	30.6	**22.5**
QuickGraph.Algorithms.Search	11	0	13.9	**13.9**
QuickGraph.Algorithms.ShortestPath	4	1.9	1.9	-
QuickGraph.Algorithms.Visitors	11	50.0	50.0	-
QuickGraph.Collections	19	14.9	29.0	**14.1**
QuickGraph.Exceptions	3	60.0	60.0	-
QuickGraph.Predicates	9	31.0	31.0	-
QuickGraph.Representations	1	2.7	21.6	**19.2**
AVERAGE				**17.4**

The receiver object and argument objects required for the `Compute` MUT are accepted as arguments for the PUT. Pex generates skeletons for the non-primitive arguments by using a heuristic-based approach (Section 5.2). For this evaluation, we used only the QuickGraph application. The reason is that Pex does not terminate in generating unit tests for the Facebook application. In future work, we plan to investigate the issues with Pex and apply Pex on the Facebook application. To provide a baseline for showing the effectiveness of MSeqGen, we first applied Pex on PUTs generated for the QuickGraph application. We executed generated unit tests and measured branch coverage achieved by these unit tests for different namespaces in the QuickGraph application. In our evaluation, we use default configurations of Pex.

We next used our extracted sequences to assist Pex. Pex provides a feature called *factory* methods, which allow programmers to provide assistance to Pex in generating non-primitive object types. We used this feature by converting our extracted sequences into factory methods. One issue with factory methods is that the current Pex allows only one factory method for a non-primitive object type. As MSeqGen can extract multiple sequences for creating an object of a non-primitive type, we combine all sequences related to a non-primitive type into one factory method by using a `switch` statement. We next apply Pex on the subject application with new factory methods created based on our extracted sequences. We again generate unit tests using Pex and measure new branch coverage.

Table 5.2 shows our results by applying Pex with and without our sequences on the QuickGraph application. On average, MSeqGen helped increase the branch coverage by 17.4% (with a maximum increase of 22.5% for one namespace). Although there is a considerable increase in branch coverage with the assistance from MSeqGen, overall Pex still achieved low branch coverage. This result is due to a limitation with the current Pex that cannot automatically identify implementing classes for interfaces and use their related factory methods. Often, factory methods created by MSeqGen accept interfaces as arguments. Therefore, Pex is not able to identify relevant factory methods for interfaces, although factory methods for their imple-

menting classes are created by MSeqGen. In future work, we plan to address this limitation and we expect that our results can be much better after addressing this limitation of Pex.

We next present example scenarios where MSeqGen is quite useful in achieving higher branch coverage with Pex. We use the `TopologicalSortAlgorithm` class in the `QuickGraph.Algorithms` namespace as an illustrative example. Without the assistance from MSeqGen, Pex did not achieve any coverage of the `TopologicalSortAlgorithm` class as Pex was not able to generate any sequences for creating objects of the `TopologicalSortAlgorithm` class. The reason for not able to generate any sequences is that the constructor of `TopologicalSortAlgorithm` accepts an interface as input. Using the factory methods generated by MSeqGen, Pex achieved a branch coverage of 57.9% (11 out of 19 branches). Our results show that MSeqGen can assist DSE-based approaches in achieving higher code coverage than without using MSeqGen.

5.5 DyGen: Regression Test Generation via Mining Dynamic Traces

5.5.1 Motivation

Software maintenance is an important phase of the software development life cycle. Software maintenance involves maintaining programs that evolve during their life time. One important aspect of software maintenance is to make sure that the changes made in the new version of software do not introduce any new defects in the existing functionality. Regression testing is a testing methodology that aims at exposing such defects, referred to as regression faults, introduced in the new version of software. Rosenblum and Weyuker [105] describe that the majority of software maintenance costs is spent on regression testing. The basis of regression testing is unit tests that achieve a high code coverage and are created on a stable version of software. It is essential to have unit tests that achieve high code coverage, since many types of defects such as functional defects are difficult to be detected without executing the relevant portions in the code under test. These unit tests created on one version of software are executed on the further versions of software to expose regression faults.

Although regression testing is our ultimate goal, in this chapter, we address the main challenge of generating unit tests that achieve a high code coverage on a given version of software. In particular, we propose a novel approach, called *DyGen*, that generates sequences from dynamic traces recorded during (typical) program executions. We use *dynamic* traces as opposed to *static* traces, since dynamic traces are more precise than static traces. These recorded dynamic traces include two aspects: realistic scenarios expressed as sequences and concrete values passed as arguments to those method calls. Since dynamic traces include both sequences and concrete argument values, these traces can directly be transformed into unit tests. However, such a naive transformation results in a large number of *redundant* unit tests that often do not

achieve high structural coverage due to *two* major issues. We next explain these two major issues of naive transformation and describe how DyGen addresses those issues.

First, since dynamic traces are recorded during program executions, we identify that many of the recorded traces are duplicates. The reason for duplicates is that the same sequence can get invoked multiple times. Therefore, a naive transformation results in a large number of redundant unit tests. To address this issue, DyGen uses a combination of static and dynamic analyses and filters out duplicate traces.

Second, unit tests generated with the naive transformation tend to exercise only *happy paths* (such as paths that do not include error-handling code in the code under test) and often do not achieve high structural coverage of the code under test. To address this issue, DyGen transforms recorded dynamic traces into PUTs [127] rather than Conventional Unit Tests (CUT). DyGen next uses Dynamic Symbolic Execution (DSE) [25] [49] [68] [70] to automatically generate a small set of CUTs that achieve high coverage of the code under test defined by the PUT. Section 5.2 provides more details on how DSE generates CUTs from PUTs. DyGen uses Pex [126], a DSE-based approach for generating CUTs from PUTs. However, DyGen is not specific to Pex and can be used with any other test-input generation engine.

DyGen addresses *two* major challenges faced by existing DSE-based approaches in effectively generating CUTs from PUTs. First, DSE-based approaches face a challenge in generating concrete values for parameters that require complex values such as floating point values or URLs. To address this challenge, DyGen uses naive transformation on each trace to generate a CUT, which is effectively an instantiation of the corresponding PUT. DyGen uses this CUT to seed the exploration of the corresponding PUT, which DyGen generates as well. Using seed tests helps not only to address the preceding challenge in generating complex concrete values, but also helps in increasing the efficiency of DSE while exploring PUTs. Second, in our evaluations (and also in practice), we identify that even after minimization of duplicate traces, the number of generated PUTs and seed tests can still be large, and it would take a long time (days or months) to explore those PUTs with DSE on a single machine. To address this challenge, DyGen uses a distributed setup that allows parallel exploration of PUTs.

In generated unit tests, DyGen infers test assertions based on the given version of software. More specifically, DyGen executes generated CUTs on the given version of software, captures the return values of method calls, and generates test assertions from these captured return values. These test assertions help detect regression defects by checking whether the new version of software also returns the same values. In summary, DyGen makes the following major contributions:

- A scalable approach for automatically generating regression tests (that achieve high structural coverage of the code under test) via mining dynamic traces from program executions and without requiring any manual efforts.

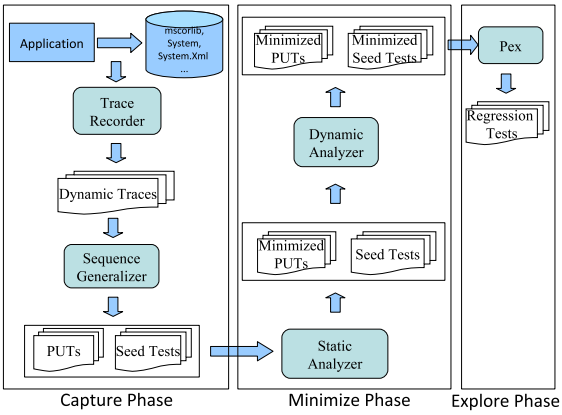


Figure 5.10: A high-level overview of DyGen.

- A technique to filter out duplicate dynamic traces by using static and dynamic analyses, respectively.
- A distributed setup to address scalability issues via parallel exploration of PUTs to generate CUTs.
- Three large-scale evaluations to show the effectiveness of our DyGen approach. In our evaluations, we show that DyGen recorded ≈1.5 GB C# source code (including 433,809 traces) of dynamic traces from applications using two core libraries of the .NET framework. From these PUTs, DyGen eventually generated 501,799 regression tests, where each test exercises a unique path, that together covered 27,485 basic blocks, which represents an increase of 24.3% over the number of blocks covered by the originally recorded dynamic traces.

5.5.2 Approach

Figure 5.10 shows the high-level overview of our DyGen approach. DyGen includes three major phases: *capture*, *minimize*, and *explore*. In the capture phase, DyGen records dynamic traces from (typical) program executions. DyGen next transforms these dynamic traces into PUTs and seed tests. Among recorded traces, we identify that there are many duplicate traces, since the same sequence of method calls can get invoked multiple times during program executions. Consequently, the generated PUTs and seed tests also include duplicates. For example, in our evaluations, we found that 84% of PUTs and 70% of seed tests are classified as duplicates by our minimize phase. To address this issue, in the minimize phase, DyGen uses a combination of static and dynamic analyses to filter out duplicate PUTs and seed tests, respectively. In the explore phase, DyGen uses Pex to explore PUTs to generate regression tests that achieve high coverage of the code under test.

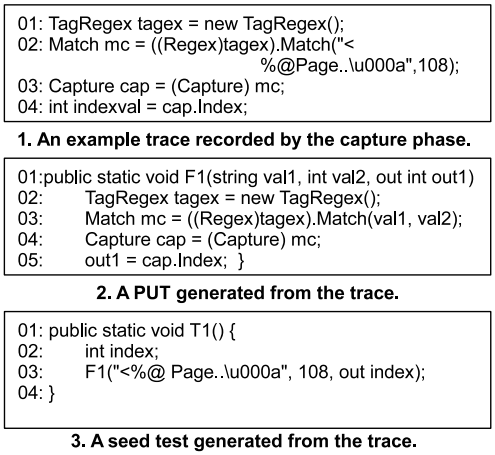


Figure 5.11: A dynamic trace and generated PUT and CUT from the trace.

Capture Phase

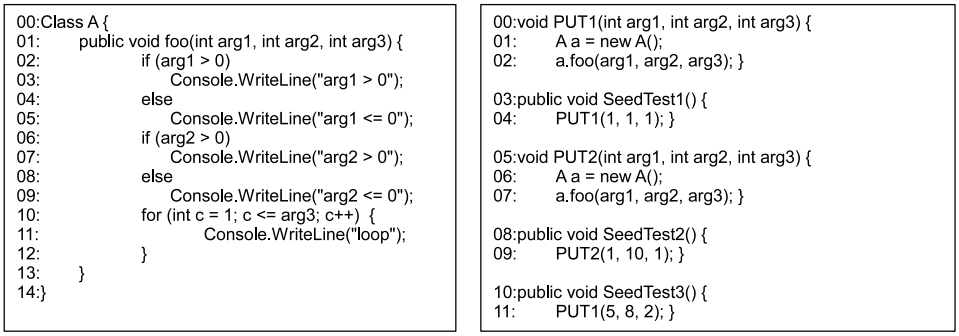
In the capture phase, DyGen records dynamic traces from program executions. The capture phase uses a profiler that records method calls invoked by the program during execution. The capture phase records both the method calls invoked and the concrete values passed as arguments to those method calls. Figure 5.11(1) shows an example dynamic trace recorded by the capture phase. Statement 2 shows the concrete value "`<% Page..\u000a`" passed as an argument for the `Match` method.

DyGen uses a technique similar to Saff et al. [106] for transforming recorded traces into PUTs and seed tests. To generate PUTs, DyGen identifies all constant values and promotes those constant values as parameters. Furthermore, DyGen identifies return values of method calls in the PUT and promotes those return values as `out` parameters for the PUT. In C#, these `out` parameters represent the return values of a method. DyGen next generates seed tests that include all concrete values from the dynamic traces. Figures 5.11(2) and 5.11(3) show the PUT and the seed test, respectively, generated from the dynamic trace shown in Figure 5.11(1).

The generated PUT includes two parameters and one `out` parameter. The `out` parameter is the return value of the method `Capture.Index`. These `out` parameters are later used to generate test assertions in regression tests. The figure also shows a seed test generated from the dynamic trace. The seed test includes concrete values of the dynamic trace and invokes the generated PUT with those concrete values.

Minimize Phase

In the minimize phase, DyGen filters out duplicate PUTs and seed tests. The primary reason for filtering out duplicates is that exploration of duplicate PUTs or execution of duplicate seed tests is redundant and can also lead to scalability issues while generating regression tests. We use PUTs and seed tests shown in Figure 5.12 as illustrative examples to explain the minimize



```
00:Class A {
01:     public void foo(int arg1, int arg2, int arg3) {
02:             if (arg1 > 0)
03:                 Console.WriteLine("arg1 > 0");
04:             else
05:                 Console.WriteLine("arg1 <= 0");
06:             if (arg2 > 0)
07:                 Console.WriteLine("arg2 > 0");
08:             else
09:                 Console.WriteLine("arg2 <= 0");
10:             for (int c = 1; c <= arg3; c++)  {
11:                     Console.WriteLine("loop");
12:             }
13:     }
14:}
```

```
00:void PUT1(int arg1, int arg2, int arg3) {
01:     A a = new A();
02:     a.foo(arg1, arg2, arg3); }

03:public void SeedTest1() {
04:     PUT1(1, 1, 1); }

05:void PUT2(int arg1, int arg2, int arg3) {
06:     A a = new A();
07:     a.foo(arg1, arg2, arg3); }

08:public void SeedTest2() {
09:     PUT2(1, 10, 1); }

10:public void SeedTest3() {
11:     PUT1(5, 8, 2); }
```

Figure 5.12: Two PUTs and associated seed tests generated by the capture phase.

phase. The figure shows a method under test `foo`, two PUTs, and three seed tests. We use these examples primarily for explaining our minimize phase. Our actual PUTs are much more complex than these illustrative examples with an average PUT size of 21 method calls (Section 5.5.3). We first present our criteria for a duplicate PUT and a seed test and next explain how we filter out such duplicate PUTs and seed tests.

Duplicate PUT: We consider a PUT, say P_1, as a duplicate of another PUT, say P_2, if both P_1 and P_2 have the same sequence of Microsoft Intermediate Language (MSIL)[5] instructions.

Duplicate Seed Test: We consider a seed test, say S_1, as a duplicate of another seed test, say S_2, if both S_1 and S_2 exercise the same execution path. This execution path refers to the path that starts from beginning of the PUT that is called by the seed test, and goes through all (transitive) method calls performed by the PUT.

DyGen uses static analysis to identify duplicate PUTs. Consider the method bodies of `PUT1` and `PUT2`. DyGen considers `PUT2` as a duplicate of `PUT1`, since both the PUTs include the same sequence of MSIL instructions. Since `PUT2` is a duplicate of `PUT1`, DyGen automatically replaces the `PUT2` method call in `SeedTest2` with `PUT1`.

After eliminating duplicate PUTs, DyGen uses dynamic analysis for filtering out duplicate seed tests. To identify duplicate seed tests, DyGen executes each seed test and monitors its execution path in the code under test. For example, `SeedTest1` follows the path "$3 \rightarrow 7 \rightarrow 11$" in the `foo` method. DyGen considers `SeedTest2` as a duplicate of `SeedTest1`, since `SeedTest2` also follows the same path "$3 \rightarrow 7 \rightarrow 11$" in the `foo` method. Consider another unit test `SeedTest3` shown in Figure 5.12. DyGen does not consider `SeedTest3` as a duplicate of `SeedTest1`, since `SeedTest3` follows the path "$3 \rightarrow 7 \rightarrow 11 \rightarrow 11$" (since `SeedTest3` iterates the loop in Statement 10 two times).

Explore Phase

In the explore phase, DyGen uses Pex to generate regression tests from PUTs. Although seed tests generated in the capture phase can be considered as regression tests, most seed tests tend

[5] `http://msdn.microsoft.com/en-us/library/c5tkafs1(VS.71).aspx`

to exercise common happy paths such as paths that do not include error-handling code in the code under test. In only a few rare scenarios, seed tests may exercise the paths related to error-handling code, if such scenarios happen during the recorded program executions. Therefore, these seed tests do not achieve high coverage of the corner cases and error handling of the code under test.

To address this issue, DyGen uses Pex to explore generated PUTs. Inspired by Patrice et al. [50], Pex can leverage seed inputs in the form of conventional unit tests. Using seed tests increases the effectiveness of Pex, and potentially any other DSE-based approaches, in two major ways. First, with seed tests, Pex executes those seed tests and internally builds an execution tree with nodes for all conditional control-flow statements executed along the paths exercised by the seed tests. Pex starts exploration from this pre-populated tree. In each subsequent iteration of the exploration, Pex tries to extend this tree as follows: a formula is constructed that represents the conjunction of the branch conditions of an already known path prefix, conjoined with the negation of a branch condition of a known suffix; the definitions of all derived values are expanded so that conditions only refer to the test inputs as variables. If the formula is satisfiable, and test inputs can be computed by the constraint solver, then by executing the PUT with those test inputs, Pex learns a new feasible path and extends the execution trees with nodes for the suffix of the new path. Without any seed tests, Pex starts exploration with an empty execution tree, and all nodes are discovered incrementally. Therefore, using seed tests significantly reduces the amount of time required in generating a variety of tests with potentially deep execution paths from PUTs. Second, seed tests can help cover reach certain paths that are hard to be covered without using those tests. For example, it is quite challenging for Pex or any other DSE-based approach to generate concrete values for variables that require complex values such as IP addresses, URLs, or floating point values. In such scenarios, seed tests can help provide desired concrete values to reach those paths.

Pex generated 86 regression tests for the PUT shown in Figure 5.11(2). Figure 5.13 shows three sample regression tests generated by Pex. In Regression tests 1 and 2, Pex automatically annotated the unit tests with expected exceptions `ArgumentNullException` and `ArgumentOutOfRangeException`, respectively. Since the PUT (Figure 5.11(2)) includes an `out` parameter, Pex generated assertions in regression tests (such as Statement 3 in Regression test 3) based on actual values captured while generating the test. These expected exceptions or assertions serve as test oracles in regression tests.

When a PUT invokes code containing loops, an exhaustive exploration of all execution paths via DSE may not terminate. While Pex employs search strategies to achieve high code coverage quickly even in the presence of loops, Pex or any other DSE-based approaches may still take a long time (days or months) to explore PUTs with DSE on a single machine. To address this issue, DyGen uses an enhanced distributed setup originally proposed in our previous

```
00:[PexRaisedException(typeof(ArgumentNullEx))]
01:public static void F102() {
02:       int i = default(int);
03:       F1 ((string)null, 0, out i);
04: }
```

a. Regression Test 1

```
00:[PexRaisedException(typeof(ArgumentOutOfRangeEx))]
01:       public static void F110() {
02:                 int i = default(int);
03:                 F1("", 1, out i);
04: }
```

b. Regression Test 2

```
00: public static void F103() {
01:       int i = default(int);
02:       F1 ("\0\0\0\0\0\0\0<\u013b\0", 7, out i);
03:       PexAssert.AreEqual<int>(0, i);
04: }
```

c. Regression Test 3

Figure 5.13: Regression tests generated by Pex by exploring the PUT shown in Figure 5.11(2).

work [126]. Our distributed setup allows to launch multiple Pex processes on several machines. Once started, our distributed setup is designed to run forever in iterations. Each subsequent iterations increase bounds imposed on the exploration to guarantee termination. For example, consider the *timeout* parameter that describes when to stop exploring a PUT. In the first iteration, DyGen sets three minutes for the timeout parameter. This value indicates that DyGen terminates exploration of a PUT after three minutes. In the first iteration, DyGen explores all PUTs with these bounded parameters. In the second iteration, DyGen doubles the values of these parameters. For example, DyGen sets six minutes for the timeout parameter in the second iteration. Doubling the parameters gives more time for Pex in exploring new paths in the code under test. To avoid Pex exploring the same paths that were explored in previous iterations, DyGen maintains a pool of all generated tests. DyGen uses the tests in the pool generated by previous iterations as seed tests for further iterations. For example, tests generated in Iteration 1 are used as seed tests in Iteration 2. Based on the amount of time available for generating tests, tests can be generated in further iterations.

5.5.3 Evaluation

We conducted three evaluations to show the effectiveness of DyGen in generating regression tests that achieve high coverage of the code under test. Our empirical results show that DyGen is scalable and can automatically generate regression tests for large real-world code bases without any manual efforts. In our evaluations, we use two core .NET 2.0 libraries[6] as main subjects. We next describe the research questions addressed in our evaluation and present our evaluation results.

Research Questions

We address the following three research questions in our evaluations.

- RQ1: Can DyGen handle large real-world code bases in automatically generating regression tests that achieve high coverage of the code under test?

[6] `http://msdn.microsoft.com/en-us/library/ms229335.aspx`

Table 5.3: Ten .NET libraries used in our evaluations.

.NET libraries	Short name	KLOC	# public classes	# public methods
mscorlib	mscorlib	178	1316	13199
System	System	149	947	8458
System.Windows.Forms	Forms	226	1403	17785
System.Drawing	Drawing	24	223	2823
System.Xml	Xml	122	270	5426
System.Web.RegularExpressions	RegEx	10	16	162
System.Configuration	Config	17	105	773
System.Data	Data	126	298	5464
System.Web	Web	202	1140	11487
System.Transactions	Trans	9.5	39	405
TOTAL		1063	5757	65982

- RQ2: How much higher coverage is achieved by using seed tests compared to without using seed tests?
- RQ3: How much higher coverage is achieved by generating new regression tests via more machine power?

Subject Code Bases

We used two core .NET 2.0 libraries as the main subjects in our evaluations. Since these libraries are the core libraries, it is paramount for the .NET product group to maintain and continually enrich a comprehensive regression test suite, in order to ensure that future product versions preserve the existing behavior, and to detect breaking changes. Table 5.3 shows the two libraries (mscorlib and System) used in our evaluations and their characteristics such as the number of classes and methods. Column "Short name" shows short names (for each library) that are used to refer to those libraries. The table also shows statistics of eight other libraries of .NET 2.0. Although these other eight libraries are not our primary targets for generating regression tests, they were exercised as well by the recorded program executions. In our evaluations, we use these additional eight libraries also while presenting our coverage results. The table shows that these libraries include 1,063 KLOC with 5,757 classes and 65,982 methods.

Evaluation Setup

In our evaluations, we used nine machines that can be classified into three configuration categories. On each machine, we launched multiple Pex processes. The number of processes launched on a machine is based on the configuration of the machine. For example, on an eight core machine, we launched seven Pex processes. Each Pex process was exploring one class

Machine Configuration	# of mc	# of pr
Xeon 2 CPU @ 2.50 GHz, 8 cores, 16 GB RAM	1	7
Quad core 2 CPU @ 1.90 GHz, 8 cores, 8 GB RAM	2	7
Intel Xeon CPU @2.40 GHz, 2 cores, 1 GB RAM	6	1

(a)

Mode	# of Tests	# of blocks	% of incr from base
WithoutSeeds Iteration 1	248,306	21,920	0%
WithoutSeeds Iteration 2	412,928	23,176	4.8%
WithSeeds Iteration 1	376,367	26,939	21.8%
WithSeeds Iteration 2	501,799	27,485	24.3%

(b)

Figure 5.14: (a) Three categories of machine configurations used in our evaluations. (b) Generated regression tests.

(including multiple PUTs) at a time. Table 5.14(a) shows all three configuration categories. Columns "# of mc" and "# of pr" show the number of machines of each configuration and the number of Pex processes launched on each machine, respectively.

Since we used .NET libraries in our evaluations, the generated tests may invoke method calls that can cause external side effects and change the machine configuration. Therefore, while executing the code during exploration of PUTs or while running generated tests, we created a sand-box with the "Internet" security permission. This permission represents the default policy permission set for the content from an unknown origin. This permission blocks all operations that involve environment interactions such as file creations or registry accesses by throwing `SecurityException`. We adopted sand-boxing after some of the Pex generated tests had corrupted our test machines. Since we use a sand-box in our evaluations, the reported coverage is lower than the actual coverage that can be achieved by our generated regression tests.

To address our research questions, we first created a base line in terms of the code coverage achieved by the seed tests, referred to as *base coverage*. In our evaluations, we use block coverage (Section 5.2) as a coverage criteria. We report our coverage in terms of the number of blocks covered in the code under test. We give only an approximate upper bound on the number of reachable basic blocks, since we do not know which blocks are actually reachable from the given PUTs for several reasons: we are executing the code in a sand-box, existing code is loaded from the disk only when it is used and new code may be generated at runtime.

We next generated regression tests in four different modes. In Mode "*WithoutSeeds Iteration 1*", we generated regression tests without using seed tests for one iteration. In Mode "*WithoutSeeds Iteration 2*", we generated regression tests without using seed tests for two iterations. The regression tests generated in Mode "WithoutSeeds Iteration 2" are a super set of the regression tests generated in Mode "WithoutSeeds Iteration 1". In Mode "*WithSeeds Iteration 1*", we generated regression tests with using seed tests for one iteration. Finally,

in Mode "*WithSeeds Iteration 2*", we generated regression tests with using seed tests for two iterations. Modes "WithoutSeeds Iteration 1" and "WithSeeds Iteration 1" took one and half day for generating tests, whereas Modes "WithoutSeeds Iteration 2" and "WithSeeds Iteration 2" took nearly three days, since these modes correspond to Iteration 2.

RQ1: Generated Regression Tests

We next address the first research question of whether DyGen can handle large real-world code bases in automatically generating regression tests. This research question helps show that DyGen can be used in practice and can address scalability issues in generating regression tests for large code bases. We first present the statistics after each phase in DyGen and next present the number of regression tests generated in each mode.

In the capture phase, DyGen recorded 433,809 dynamic traces and persisted them as C# source code, resulting in ≈1.5 GB of C# source code. The average trace length includes 21 method calls and the maximum trace length includes 52 method calls. Since our capture phase transforms each dynamic trace into a PUT and a seed test, the capture phase resulted in 433,809 PUTs and 433,809 seed tests.

In the minimize phase, DyGen uses static analysis to filter out duplicate PUTs. Our static analysis took 45 minutes and resulted in 68,575 unique PUTs. DyGen uses dynamic analysis to filter out duplicate seed tests. Our dynamic analysis took 5 hours and resulted in 128,185 unique seed tests. These results show that there are a large number of duplicate PUTs and seed tests, and show the significance of our minimize phase. We next measured the block coverage achieved by these 128,185 unique seed tests in the code under test and used this coverage as *base coverage*. These tests covered 22,111 blocks in the code under test.

Table 5.14(b) shows the number of regression tests generated in each mode along with the number of covered blocks. The table also shows the percentage of increase in the number of blocks compared to the base coverage. As shown in results, in Mode "WithSeeds Iteration 2", DyGen achieved 24.3% higher coverage than the base coverage. Table 5.4 shows more detailed results of coverage achieved for all ten .NET libraries. Column ".NET libraries" shows libraries under test. Column "Maximum Coverage" shows an approximation of the upper bound (in terms of number of blocks) of achievable coverage in each library under test. In particular, this column shows the sum of all blocks in all methods that are (partly) covered by any generated test. However, we do not present the coverage results of our four modes as percentages relative to these upper bounds, since these upper bounds are only approximate values, whereas the relative increase of achieved coverage can be measured precisely. Column "Base Coverage" shows the number of blocks covered by seed tests for each library. Column "WithOutSeeds Iteration 1" shows the number of blocks covered ("# blocks") and the percentage of increase in the coverage ("% increase") with respect to the base coverage in this mode. Similarly, Columns

Table 5.4: Comparison of coverage achieved for ten .NET libraries used in our evaluation.

.NET libraries	**Maximum Coverage**	**Base Coverage**	**WithOutSeeds Iteration 1**		**WithOutSeeds Iteration 2**		**WithSeeds Iteration 1**		**WithSeeds Iteration 2**	
	# blocks	# blocks	# blocks	% incr	# blocks	% incr	# blocks	% incr	# blocks	% incr
mscorlib	20437	12827	13063	1.84	13620	6.18	14808	15.44	15018	17.08
System	7786	4651	4062	-12.67	4243	-8.77	5907	27.00	6039	29.84
Forms	2815	1730	1572	-9.13	1774	2.54	1782	3.01	1865	7.80
Drawing	850	570	580	1.75	591	3.68	618	8.42	625	9.65
Xml	2770	1229	1390	13.10	1462	18.96	1959	59.40	2045	66.40
RegEx	854	351	330	-5.98	520	48.15	754	114.81	771	119.66
Config	392	263	297	12.93	297	12.93	302	14.83	306	16.35
Data	865	301	380	26.25	422	40.20	562	86.71	569	89.04
Web	253	154	211	37.01	212	37.66	212	37.66	212	37.66
Trans	59	35	35	0.00	35	0.00	35	0.00	35	0.00
TOTAL/ AVG	**37081**	**22111**	**21920**	**<0**	**23176**	**4.80**	**26939**	**21.80**	**27485**	**24.30**

"WithOutSeeds Iteration 2", "WithSeeds Iteration 1", and "WithSeeds Iteration 2" show the results for the other three modes.

Since we use seed tests during our exploration in Modes "WithSeeds Iteration 1" or "WithSeeds Iteration 2", the coverage achieved is either the same or higher than the base coverage. However, DyGen has achieved significant higher coverage than base coverage for libraries mscorlib and System (in terms of the number of additional blocks covered). The primary reason is that most of the classes in these libraries are stateless and do not require environment interactions.

Although our generated tests achieved higher coverage (24.3%) than the seed tests, we did not achieve full overall coverage of our subject code bases (i.e. 100% coverage of all methods stored in the code bases on disk). There are three major reasons for not achieving full coverage. First, using a sand-box reduces the amount of executable code. Second, our recorded dynamic traces do not invoke all public methods of the libraries under analyses. In future work, we plan to address this issue by generating PUTs for all public methods that are not covered. Third, the code under test includes branches that cannot be covered with the test scenarios recorded during program executions. To address this issue, we plan to generate new test scenarios from existing scenarios by using evolutionary techniques [128]. In summary, the results show that DyGen can handle large real-world code bases and can generate large number of regression tests that achieve high coverage of the code under test.

RQ2: Using Seed Tests

We next address the second research question of whether seed tests help achieve higher code coverage compared to without using seed tests. To address this question, we compare the coverage achieved by generated tests in Modes "WithoutSeeds Iteration 2" and "WithSeeds

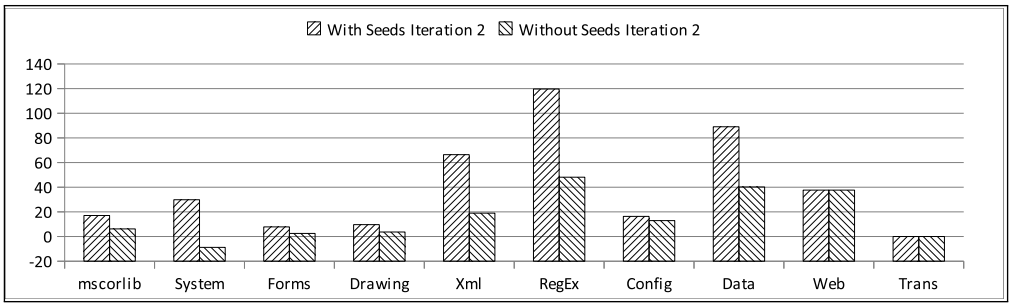


Figure 5.15: Comparison of coverage achieved by Mode "WithSeeds Iteration 2" and Mode "WithoutSeeds Iteration 2".

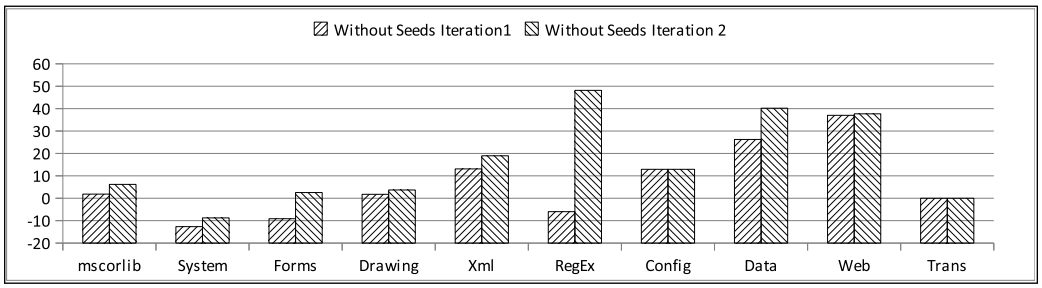


Figure 5.16: Comparison of code coverage achieved by Modes "WithoutSeeds Iteration 1" and "WithoutSeeds Iteration 2".

Iteration 2". Figure 5.15 shows comparison of the coverage achieved in these two modes. The x-axis shows the library under test and y-axis shows the percentage of increase in the coverage with respect to the base coverage. As shown, Mode "WithSeeds Iteration 2" always achieved higher coverage than Mode "WithoutSeeds Iteration 2". On average "WithSeeds Iteration 2" achieved 18.6% higher coverage than "WithoutSeeds Iteration 2". The table also shows that there is a significant increase in the coverage achieved for the `System.Web.RegularExpressions` (`RegEx`) library. In Section 5.5.2, we described one of the major advantages of seed tests is that seed tests can help cover certain paths that are hard to be covered without using those tests. The `System.Web.RegularExpressions` library is an example for such paths since this library requires complex regular expressions to cover certain paths in the library. It is quite challenging for Pex or any other DSE-based approach to generate concrete values that represent regular expressions. The increase in the coverage for this library shows that concrete values in the seed tests help achieve higher coverage. In summary, the results show that seed tests help achieve higher coverage compared to without using seed tests.

RQ3: Using More Machine Power

We next address the third research question of whether more machine power helps achieve more coverage. This research question helps show that additional coverage can be achieved in further iterations of DyGen. To address this question, we compare coverage achieved in Mode

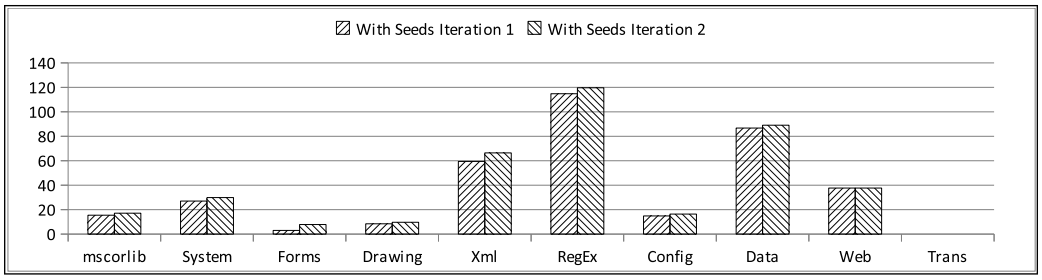


Figure 5.17: Comparison of code coverage achieved by Modes "WithSeeds Iteration 1" and "Withseeds Iteration 2".

"WithoutSeeds Iteration 1" with Mode "WithoutSeeds Iteration 2", and Mode "WithSeeds Iteration 1" with Mode "WithSeeds Iteration 2" (shown in Table 5.4).

Figure 5.16 shows the comparison of coverage achieved in Modes "WithoutSeeds Iteration 1" and "WithoutSeeds Iteration 2". On average, Mode "WithoutSeeds Iteration 2" achieved 5.73% higher coverage than Mode "WithoutSeeds Iteration 1". This result shows that DyGen can achieve additional coverage in further iterations. However, the coverage from Mode "WithoutSeeds Iteration 1" to Mode "WithoutSeeds Iteration 1" is not doubled. The primary reason is that it gets harder to cover new blocks in further iterations.

Figure 5.17 shows the comparison of coverage achieved in Modes "WithSeeds Iteration 1" and "WithSeeds Iteration 2". On average, Mode "WithSeeds Iteration 2" achieved 2.0% higher coverage than Mode "WithSeeds Iteration 1". The increase in coverage from Mode "WithSeeds Iteration 1" to Mode "WithSeeds Iteration 2" is less than the increase in the coverage from Mode "WithoutSeeds Iteration 1" to Mode "WithoutSeeds Iteration 2". This difference is due to seed tests that help achieve higher coverage during Mode "WithSeeds Iteration 1", leaving more harder blocks to be covered in Mode "WithSeeds Iteration 2". In summary, the results show that further iterations can help generate new regression tests that can achieve more coverage.

5.6 Seeker: Demand-Driven Method Sequence Generation

5.6.1 Motivation

Although both MSeqGen and DyGen are shown effective in achieving high coverage of the code under test, a major issue with both these approaches is that they are not effective when code bases that use classes required for generating target sequences are not available or code bases include sequences that are different from target sequences. For example, if a class c is newly introduced, it is not possible to find code bases using the class c. Furthermore, mined sequences may not include all necessary method calls required for producing desired object states.

To address these issues, we propose a novel approach, called *Seeker*, that uses a combination of static and dynamic analyses. Seeker synthesizes skeletons statically and leverages a technique

```
00:class AdjacencyGraph : IVEListGraph {
01:   private VEDictionary m_VOutEdges;
02:   private ECollection m_Edges;
03:   private ArrayList m_Vertices; ...
04:   public void AddVertex(IVertex v) {
05:      m_VOutEdges.Add(v, new ECollection());
06:      m_Vertices.Add(v); } ...
07:   public Edge AddEdge(IVertex v1, IVertex v2) {
08:      if (!m_Vertices.Contains(v1)) //(B1)
09:         throw new VNotFoundException("");
10:      if (!m_Vertices.Contains(v2)) //(B2)
11:         throw new VNotFoundException(""); ...
12:      // create edge
13:      IEdge e = Provider.ProvideEdge(v1, v2);
14:      VOutEdges[v1].Add(e);
15:      m_Edges.Add(e); }
16:} //End of class AdjacencyGraph
```

Figure 5.18: A class under test from C# QuickGraph library [100].

called Dynamic Symbolic Execution (DSE)[7] [49, 70, 126] for dynamic analysis. DSE explores an MUT and generates inputs that can achieve high structural coverage of the MUT. Although DSE can effectively handle generation of primitive data, DSE alone cannot generate target sequences. The primary reason is that DSE, being a dynamic-analysis technique, does not have the knowledge of methods that are not yet explored. Therefore, DSE cannot generate skeletons in target sequences, since it is not possible to explore all methods especially in real-world applications. The reason is that often real-world applications reuse API classes and methods from system libraries such as .NET libraries.

On the other hand, static analysis alone cannot generate target sequences, since the inherent imprecision of static analysis causes combinatorial explosion when generating skeletons [49]. Furthermore, static analysis faces challenges in generating primitive data. For example, consider the `UndirectedDFS` example shown Figures 5.18 and 5.19 (the same example presented in Section 5.1). Consider B1 in Statement 8 of Figures 5.18 as a target branch. The following method sequence (*S1*) produces the preceding desired object state, thereby covering B3.

```
00: AdjacencyGraph ag = new AdjacencyGraph();
01: Vertex v1 = new Vertex(0);
02: ag.AddVertex(v1);
03: ag.AddEdge(v1, v1);
```

In this sequence, `AddVertex` should precede `AddEdge` to satisfy the requirement (Statements 8 and 10) that the vertices passed as arguments should already exist in the graph object. We refer to method sequences that produce desired object states as *target sequences*. Target

[7]DSE-based approaches are also referred to as approaches that use mixed concrete and symbolic execution.

```
17:class UDFSAlgorithm {//UDFS:UndirectedDepthFirstSearch
18:      private IVEListGraph m_Graph;
19:      private VColorDictionary m_Colors;
20:      private EColorDictionary m_EdgeColors;
21:      private bool isComputed;
22:      public bool IsComputed {
23:         get { return isComputed; } }
24:      public UDFSAlgorithm(IVEListGraph g) {
25:         ... }
25:      public void Compute(IVertex s) { ...
27:         bHasEdges = false;
28:         foreach(IEdge e in m_Graph.Edges) { //(B3)
29:         bHasEdges = true;
30:         m_EdgeColors[e]=GraphColor.White; } ...
31:         if (bHasEdges) { //(B4)
32:            isComputed = true;
33:            foreach (IEdge e in m_Graph.Edges) { //(B5)
34:               if (m_EdgeColors[e] == GraphColor.White) {
35:                  isComputed = false; break; }
36:            }
37:      } }
38:} //End of class UDFSAlgorithm

Method Under Test (MUT):
39:public void foo(UDFSAlgorithm udfs) {
40:   if(udfs.IsComputed) } //(B6)
41:      ...  }
42:}
```

Figure 5.19: Another class under test and an MUT from C# QuickGraph library [100].

sequences include two major parts: skeletons (method sequences without primitive data) and primitive data passed as arguments to those method calls. Both skeletons and data are required for producing target sequences. Consider generating the desired object state produced by Sequence S1 using static analysis alone. Static analysis, being conservative, identifies three methods (`AddVertex`, `RemoveVertex`, and `ClearVertex` of the `AdjacencyGraph` class) that modify the field `m_Vertices` as candidates for Statement 2 in S1. Similarly, static analysis identifies six candidates for Statement 3, resulting in the total number of candidate sequences as 18 (3 * 6) and thereby causing combinatorial explosion.

Seeker addresses these preceding challenges by using a feedback loop between static analysis and dynamic analysis (e.g., DSE). In particular, Seeker statically synthesizes sequences in stages and uses dynamic analysis to filter out candidate methods that are not required at each stage. For example, in Statement 2 of S1, Seeker statically identifies three methods (`AddVertex`, `RemoveVertex`, and `ClearVertex`) as candidates. Seeker next uses DSE to filter out `RemoveVertex` and `ClearVertex` methods that do not help in producing the desired object state. In essence,

Seeker explores nine candidate sequences (3 + 6) to produce the desired object state of S1. More specifically, Seeker reduces the number of candidate sequences from $(a_1 * a_2 * .... * a_n)$ to $(a_1 + a_2 + ... + a_n)$. This feedback loop between static and dynamic analyses is the key essence of Seeker and helps systematically explore a potentially large space of possible sequences, thereby scaling Seeker to large real-world applications. To address technical challenges such as nested classes, Seeker includes a novel technique based on method-call graphs. A method-call graph is a directed graph that includes caller-callee relations among methods. This technique helps synthesize skeletons that generate desired values for member fields including private fields.

Seeker makes the following major contributions:

- A novel approach, called Seeker, that addresses the challenging problem of generating target sequences without requiring any additional information. Seeker forms a feedback loop between static and dynamic analyses for incrementally generating target sequences.

- A technique based on method-call graphs to analyze the branches that are not yet covered by DSE and synthesize skeletons that can help cover those not-yet-covered branches. Our technique automatically handles nested classes.

- Three evaluations with four popular applications (including 28 KLOC) to show the effectiveness of our Seeker approach. In our evaluations, we show that Seeker performs better than two state-of-the-art test generation approaches: Pex [126] and Randoop [97] that are representative of a DSE-based and a random approach, respectively. The results show that Seeker achieves 12% (653 new branches) and 26% (1571 new branches) higher branch coverage than Pex and Randoop, respectively. Achieving such higher coverage compared to Pex and Randoop is significant, since the branches that are not covered by these approaches are generally quite hard to cover. Seeker also detects 34 new defects including an infinite loop defect in QuickGraph [100].

5.6.2 Example

We next explain our approach using the same illustrative examples shown in Figures 5.18 and 5.19. The figure shows two classes under test `AdjacencyGraph` and `UDFSAlgorithm` from the QuickGraph library [100]. `AdjacencyGraph` represents a graph structure including vertices and edges, which are added using `AddVertex` and `AddEdge`, respectively. `UDFSAlgorithm` preforms an undirected depth first search on the graph structure. We added an additional method `IsComputed` for illustrative purposes. Consider the `foo` method as the MUT and Branch B6 in Statement 40 as a target branch to cover. We first present the branch coverage achieved by Pex and Randoop on these two classes and next describe our Seeker approach.

The test inputs generated by Randoop and Pex achieved branch coverage of 36.8% (21 out of 57) and 35.1% (20 out of 57), respectively. The reason for low coverage is that neither Randoop

```
00: VEProvider s0 = new VEProvider();
01: Vertex s1 = new Vertex(0);
02: AdjacencyGraph ag = new AdjacencyGraph(s0, false);
03: ag.AddVertex(s1);
04: ag.AddEdge((IVertex)s1, (IVertex)s1);
05: UDFSAlgorithm ud = new UDFSAlgorithm(ag);
06: ud.Compute((IVertex)null);
```

Figure 5.20: A test input generated by Seeker.

nor Pex could satisfy the requirement of `AddEdge` to successfully add an edge to the graph object (Branch B1 in Statement 8). Furthermore, neither Pex nor Randoop could generate a sequence that helps cover Branch B6 in Statement 40. As shown through this example, it is quite challenging to achieve high branch coverage of these classes under test due to the requirement of complex sequences. Such requirement is often encountered when testing object-oriented code.

We next present how our Seeker approach achieves high branch coverage by generating sequences incrementally based on the branches that are not yet covered by a DSE-based approach such as Pex. Although we describe our approach in the context of Pex, our approach is independent of Pex and can be used to assist any other DSE-based approach [9]. Initially, Seeker applies DSE to explore the `foo` method. DSE generates a `null` value for `udfs`, resulting in a `NullReferenceException`. Seeker suggests the following skeleton (to DSE) that helps create an object of `UDFSAlgorithm`.

```
00: bool arg1 = <sym>;
01: VEProvider s0 = new VEProvider();
02: AdjacencyGraph ag = new AdjacencyGraph (s0, arg1);
03: UDFSAlgorithm ud = new UDFSAlgorithm(ag);
```

In the preceding skeleton, `<sym>` represents a symbolic variable, whose value is generated by Pex while exploring `foo`. DSE again explores `foo` with the new skeleton and identifies that Branch B6 is not covered. Seeker analyzes Branch B6 statically and identifies that the `Compute` method can help cover this branch, since `Compute` changes the value of member field `isComputed` (Statement 32). Seeker may suggest more than one method due to imprecision of static analysis. The new skeleton suggested by Seeker is as follows:

```
00: bool arg1 = <sym>;
01: Vertex v = new Vertex(<sym>);
02: VEProvider s0 = new VEProvider();
03: AdjacencyGraph ag = new AdjacencyGraph (s0, arg1);
04: UDFSAlgorithm ud = new UDFSAlgorithm(ag);
05: ud.Compute(v);
```

After exploring `foo` with the preceding skeleton, DSE identifies Branches B3 and B4 as not-covered branches. Seeker further generates new skeletons and applies DSE to cover those not-yet-covered branches. Figure 5.20 shows the final target sequence (generated by Seeker)

that covers Branch B6. The test input includes four classes and six method calls. Using the skeleton generated by Seeker, DSE generates required values for symbolic variables to produce desired object states. For example, DSE identifies that the arguments passed to `AddEdge` should be the same as the argument passed to `AddVertex` to successfully add an edge. The test inputs generated by Seeker achieved 84.2% (48 out of 57) branch coverage. The remaining not-covered branches are related to the event handling mechanism, which is currently not handled by our implemented prototype. It is quite challenging to generate such sequences either randomly or using heuristics, since these four classes include 39 methods. However, the feedback loop between Seeker and DSE incrementally generates target sequences, thereby achieving high structural coverage of the code under test.

5.6.3 Approach

Algorithms 7 and 8 show the two key algorithms `SeekerMain` and `AnalyzeTB`, respectively, of our Seeker approach. Seeker leverages DSE to generate sequences **incrementally** for covering the target branches that are not yet covered by DSE.

Algorithm 7 $SeekerMain(M_u)$

Require: A method under test M_u.
Ensure: Set<OSS> of M_u

```
1: Set<OSS> GlobOSS=(); Set<SKT> TempSKT=()
2: Set<TB> GlobCBSet=(); Set<M> AllExploredM=()
3: while true do
4:     DSE(M_u, TempSKT, out CovB, out UnCovB) //Processing all covered branches
5:     for all B tb ∈ CovB do
6:         GlobOSS += GetOSS(tb)
7:     end for
8:     if UnCovB.Count == 0 then
9:         break //Checking for exit condition
10:    end if
       //Processing all target (not covered) branches
11:    TempSKT = ()
12:    for all B tb ∈ UnCovB do
13:        TempSKT += AnalyzeTB(tb)
14:    end for
15: end while
16: AllExploredM += M_u
17: return GlobOSS
```

SeekerMain Algorithm

Given a method under test M_u, `SeekerMain` applies DSE (referred to with a function call `DSE` in Line 4 of Algorithm 7) to explore M_u. For each M_u, `SeekerMain` maintains a set of variables declared in Lines 1 to 2. `GlobOSS` includes all test inputs that are gathered via multiple iterations of Loop 3-15 (i.e., Lines 3 to 15), which is the *feedback* loop between Seeker and DSE. `TempSKT` includes temporary skeletons passed as inputs to `DSE`. Finally, the variable `AllExploredM` stores the methods under test that are already explored by `SeekerMain`.

Initially, `SeekerMain` applies `DSE` on M_u with an empty `TempSKT` and gathers the covered branches `CovB` and not-covered (target) branches `UnCovB`. `SeekerMain` terminates if there are no target branches. Otherwise, `SeekerMain` analyzes each target branch using `AnalyzeTB` and produces new skeletons that are stored in `TempSKT` (Lines 12 to 14), which is passed as input to DSE in the next iteration. Whenever a new branch is covered, `SeekerMain` updates `GlobOSS` with the skeleton that helped in covering that branch.

AnalyzeTB Algorithm

We next describe the second algorithm `AnalyzeTB`, which analyzes a target branch *tb* and generates skeletons that help cover *tb*. First, we explain the three major functions `DetectFields` (Line 4), `SuggestMethods` (Line 5), and `DetectPreqB` (Line 16) used in the algorithm and next explain the `AnalyzeTB` algorithm. The variables `FailedB` and `SuspB` represent the branches that Seeker fails to cover and that Seeker suspends, respectively.

DetectFields. Given *tb*, `DetectFields` statically identifies `tfields` (target fields that are of primitive types) that need to be modified to produce a desired object state for covering the target branch *tb*. It is trivial to identify `tfields` for target branches such as `if(a == 10)`, where `tfields` are directly included in the *tb*. However, in object-oriented code, target branches often involve method calls such as `if(!m_Vertices.Contains(v1))` in Statement 8 (Figure 5.18) rather than fields. It is challenging to identify target fields in the presence of method calls, since these method calls can further include nested method calls.

To address this issue, `DetectFields` uses an inter-procedural execution trace (hereby referred to as *trace*), which includes the statements executed in each method, gathered during the exploration of M_u with DSE. `DetectFields` performs backward analysis of the trace starting from the method call involved in *tb*. If the executed `return` statement of the method call in *tb* includes another nested method call, `DetectFields` analyzes the execution trace of the nested method call. `DetectFields` repeats the preceding step until a `return` statement without including method calls is found. The member fields involved in that `return` statement are identified as `tfields`. Along with identifying `tfields`, `DetectFields` identifies current values and desired values for `tfields` by applying a constraint solver on the *tb*.

Algorithm 8 *AnalyzeTB*(tb)

Require: A target branch tb, Current Method $currM$
Ensure: SKT of tb

```
1: Set<SKT> TbSKT = ()
2: Set<B> FailedB, SuspB
3:
4: Set<Field> tfields = DetectFields(tb)
5: Set<M> tmethods = SuggestMethods(tfields)
6: for all M m ∈ tmethods do
7:     if m ∈ currApplication then
8:         if m ≠ currM then
9:             //STEP 1
10:            if m ∉ AllExploredM then
11:                SeekerMain(m)
12:            end if
13:            TbSKT += (GetSKT(m) + m)
14:        else
15:            //STEP 2
16:            B PreqB = DetectPreqB(tb, m)
17:            if PreqB ∈ FailedB then
18:                continue;
19:            else if PreqB ∈ SuspB then
20:                SuspB += PreqB
21:            else
22:                TbSKT += (GetSKT(m) + m)
23:            end if
24:        end if
25:    else
26:        //STEP 3
27:        TbSKT += m
28:    end if
29: end for
30: return TbSKT
```

For example, consider the `true` branch of Statement 28 (Branch B3 in Figure 5.19) as tb. Given this tb, `DetectFields` first analyzes the trace of `Edges`, which is a getter method for the field `m_Edges`. Since the return statement includes a method call `MoveNext` on a non-primitive object of type `EdgeCollection`, `DetectFields` next analyzes the trace of the `MoveNext` method. Here, `MoveNext` (originally declared in the `System.Collections.ArrayList+ArrayListEnumeratorSimple` class) is the method that gets invoked in a `Foreach` statement. Finally, `DetectFields` identifies `tfield` as `ArrayList._size` and condition that is not satisfied as "`ArrayList._size + -1` >

-1" for not covering this branch. DetectFields also identifies the hierarchy of fields, referred to as field hierarchy, as "FH: udfs → UDFSAlgorithm.m_Graph → AdjacencyGraph.m_Edges → CollectionBase.list → ArrayList._size". This field hierarchy describes that _size is contained in the object type of list (ArrayList), which is in turn contained in the object type of m_Edges (CollectionBase) and so on. This field hierarchy is used by SuggestMethods discussed next. Using a constraint solver on tb, DetectFields identifies that the current and desired values for _size are zero and one, respectively.

SuggestMethods. Given a target field such as _size and its current and desired values, SuggestMethods identifies the target methods that can help achieve those desired values. Initially, SuggestMethods statically analyzes all public methods of the declaring class of the target field to identify the target methods that modify the target field. In particular, SuggestMethods identifies assignment statements, where the target field is on the left hand side. For example, SuggestMethods identifies the methods such as Add, Insert, and Reset of ArrayList as target methods.

The preceding step is sufficient when a class under test does not include non-primitive member fields $npmf_i$. Otherwise, if a class under test includes $npmf_i$, often it is not possible to call these identified methods directly. The reason is that these $npmf_i$ are often not visible outside the class under test. For example, identified target methods of the ArrayList class cannot be called, since the list member field is not visible outside the CollectionBase class.

To address this issue, SuggestMethods constructs a *method-call graph*, which is a directed graph that includes caller-callee relations among methods. SuggestMethods constructs the method-call graph on demand based on the field hierarchy identified by DetectFields. Figure 5.21 shows a sample method-call graph constructed for the field hierarchy FH. The root node of the graph includes the target field in tfield. The first level of the graph includes the methods (in the declaring class) that modify the target field. From the second level, the graph includes the methods from the declaring classes of fields in the field hierarchy. The graph includes an edge from a method M_i in one level to a method M_j in the next level, if M_i is called by M_j. For example, AdjacencyGraph.AddEdge invokes the CollectionBase.Add method and corresponding edge is shown from Levels L2 to L3.

SuggestMethods next traverses the method-call graph from the top to bottom and identifies the methods that can be called on the first visible field or the last field in the field hierarchy. SuggestMethods recommends these methods as target methods. For example, by traversing the method-call graph in Figure 5.21, SuggestMethods identifies the target method as AddEdge method (Level 3), since the field m_Graph is visible outside the UDFSAlgorithm (set through its constructor).

DetectPreqB. Given a target branch tb and a method m, the DetectPreqB function identifies prerequisite branches, referred to as PreqB, that need to be covered before covering tb

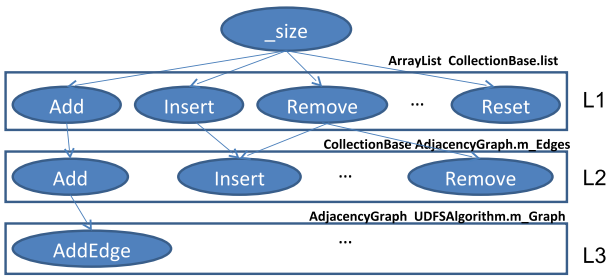


Figure 5.21: A sample method-call graph.

within m. Using a similar technique as `SuggestMethods`, `DetectPreqB` first identifies the assignment statements, where the target field is on the left hand side. Next, `DetectPreqB` constructs a control-flow graph of m and performs backward traversal from the identified assignment statements to the beginning of m to capture the prerequisite branches.

For example, consider a target branch as Branch B4 (Statement 31 in Figure 5.19). To cover B4, `DetectPreqB` identifies `PreqB` as Branch B3 (Statement 28), since covering the `true` branch of B3 helps modify the `bHasEdges` variable to achieve the desired value.

AnalyzeTB. We next describe the complete `AnalyzeTB` algorithm. Given a target branch tb, `AnalyzeTB` identifies skeletons that help cover the target branch. `AnalyzeTB` uses three steps for analyzing each target method returned by the `SuggestMethods` function.

Step 1. `AnalyzeTB` executes Step 1, if the target method `m` is in the code under test and is not the same method as `currM`, which represents the current method under exploration. If m is not yet explored, `AnalyzeTB` calls `SeekerMain` on m to explore m and gathers SKT of m, since `m` belongs to the code under test and has to be explored in future exploration. `AnalyzeTB` uses SKT of m to construct SKT desired for covering tb.

Step 2. `AnalyzeTB` executes Step 2, if the target method `m` is in the code under test and is the same as `currM`. This scenario can happen during the exploration of methods such as `Compute` (Figure 5.19), where the same method is suggested as the target method for covering the `true` branch B4 (Figure 5.19). `AnalyzeTB` constructs desired SKT for covering tb based on $PreqB$ identified using `DetectPreqB`.

Step 3. `AnalyzeTB` executes Step 3, if the target method `m` is not in the code under test. Since m is not in the code under test, `AnalyzeTB` simply uses the target method to construct the desired SKT for covering tb.

Based on the preceding steps, `AnalyzeTB` either returns SKT that can help cover tb or suspends tb based on its prerequisite branches. The branches suspended by `AnalyzeTB` are resumed later after their prerequisite branches are covered.

Implementation

We implemented a prototype for our Seeker approach as an external plugin for Pex 0.92 [126]. Pex provides extensible Application Programming Interfaces (APIs) that can be overridden to add new functionality. Given an M_u, our prototype launches Pex multiple times to generate test inputs. Our prototype uses a file-based repository to cache information across multiple launches of Pex. After each launch, Pex returns target branches that are not yet covered due to lack of desired method sequences. Our prototype analyzes those target branches and generates skeletons. Our prototype persists these generated skeletons in the repository and reuses them during the next launch of Pex. Our prototype relaunches Pex with new skeletons to generate primitive data for the method calls in skeletons. We expect that the runtime performance of our prototype can be significantly improved by implementing our approach within Pex or by adopting a memory-based repository, left as our immediate future work. Our current open-source prototype can be downloaded from `http://pexase.codeplex.com/releases/view/50822`.

5.6.4 Evaluation

To show the effectiveness of our Seeker approach, we compared our approach with two categories of approaches: *random* and *DSE-based approaches*. We used two state-of-the-art tools Randoop [97] and Pex [126] as representative tools for random and DSE-based approaches, respectively. We applied all three approaches on four popular real-world applications. We also compared our results with the results of existing manual tests available with the subject applications. More details of subjects and results of our evaluations are available at `http://research.csc.ncsu.edu/ase/projects/seeker/`. All experiments were conducted on a machine with 3.33GHz Intel Core 2 Duo processor with 4 GB RAM.

Research Questions

In our evaluations, we addressed the following research questions.

- **RQ1:** How much higher percentage of branch coverage is achieved by Seeker compared to Randoop and Pex, respectively? This research question helps show that Seeker performs better than Randoop and Pex in achieving high structural coverage such as branch coverage of the code under test.
- **RQ2:** How much higher percentage of def-use coverage is achieved by Seeker compared to Randoop and Pex, respectively? This research question helps show that Seeker performs better than Randoop and Pex in achieving high data-flow coverage [42] such as def-use coverage of the code under test.

Table 5.5: Subjects and their characteristics.

Subject	Version	# Classes	# Methods	KLOC
QuickGraph	1.0	88	634	5.1
Dsa	0.6	27	308	3.3
XUnit	1.6.1	151	1267	11.9
NUnit	2.5.7	225	2344	8.1
TOTAL		**491**	**4553**	**28.4**

- **RQ3:** How many new defects are detected by Seeker compared to Randoop and Pex, respectively? This research question helps address whether Seeker has higher defect-detection capabilities compared to Randoop and Pex, respectively.

- **RQ4:** How high percentage of branch coverage is achieved by Seeker, MSeqGen [125], and their combination? Since Seeker and MSeqGen complement each other, this research question helps show the benefits and limitations of Seeker, MSeqGen, and their combined approach.

Subjects

We used four popular applications as subjects in our evaluations. Table 5.5 shows the characteristics such as the number of classes, methods, and KLOC of our subjects. QuickGraph [100] is a popular C# graph library that provides various graph data structures and algorithms such as depth-first search. Data structures and algorithms (Dsa)[8] provides various data structures, complementing those from the .NET libraries. xUnit[9] and NUnit[10] are widely used open source unit testing libraries for all .NET languages. For NUnit, we focused on applying all three approaches on its core component, the `util` namespace (including 8.1 KLOC). We used these applications as subject applications, since these applications are popularly used and also by previous work [125]. All subjects include a total of 28 KLOC.

Evaluation Setup

We next describe our evaluation setup for addressing the preceding three research questions. Seeker and Pex accept Parameterized Unit Test (PUT) [127] as input and generate conventional tests. Unlike conventional tests, PUTs accept parameters. Since PUTs are not available with our subjects, we automatically generated PUTs for each public method by using the *PexWizard* tool, which is provided with Pex. We first applied Seeker on PUTs generated for each subject application. We measured four metrics for generated tests: branch coverage, def-use coverage,

[8] `http://dsa.codeplex.com/`
[9] `http://xunit.codeplex.com/`
[10] `http://www.nunit.org/`

number of distinct defects detected, and the time taken. We used a coverage measurement tool, called NCover[11], to measure the branch coverage achieved by generated tests. For measuring def-use coverage, we developed a tool, called DUCover, for C# based on the techniques described in [54, 83, 96], since there exist no def-use coverage measurement tool for C#. In particular, in object-oriented code, definitions and uses for instance variables can occur in different member methods of classes under analysis. DUCover automatically measures coverage of such def-use pairs based on method sequences among generated tests. We measured distinct defects, since multiple tests could detect the same defect.

As mentioned in Section 5.6.3, our implementation has runtime performance overhead, since we launch Pex multiple times. To ensure that our results are not biased by the limitations of our implementation, we used customized settings for Pex and Randoop rather than using their default settings, respectively. These customized settings allow Pex and Randoop to run for the same or higher amount of time compared to Seeker. In essence, our settings favor Pex and Randoop compared to Seeker. We used the following customized settings for Pex and measured the three metrics for the tests generated by Pex.

```
Timeout = 500 sec. (default: 120)
MaxConstraintSolverTime = 10 sec. (default: 2)
MaxRunsWithoutNewTests = 2147483647 (default: 100)
MaxRuns = 2147483647 (default: 100)
```

The values in brackets represent the default values. For example, the default value of the timeout parameter is 120 seconds. Instead, we used 500 seconds for the timeout parameter. Similarly, for Randoop, we set the timeout value (default: 120 seconds) for each subject to the same amount of time taken by Seeker for that subject. However, we observed that Randoop may generate thousands of tests that are too many to be compiled within visual studio for measuring metrics. Therefore, we limited the number of generated tests to $10,000$.

To compare Seeker with MSeqGen (RQ4), we first applied MSeqGen alone on QuickGraph, which is the only subject used for MSeqGen (integrated with Pex), and measured the three metrics. For the combined approach, we used sequences extracted by MSeqGen as input to Seeker. In this setting, Seeker enhances the sequences extracted by MSeqGen to generate more sequences that could help produce desired object states.

RQ1: Branch Coverage

We next address the first research question. Table 5.6 shows our results for all subject applications. For each subject, due to space constraint, we show results for a few selected namespaces (Column "Namespace") that help provide insights in subsequent sections, instead of all namespaces. Column "Branches" shows the number of branches in each application. Among the

[11] http://www.ncover.com/

Table 5.6: Branch coverage achieved by Randoop, Pex, Seeker, and manually written tests.

Subject	**Namespace**	**Branches**	**Randoop**			**Pex**			**Seeker**			**Manual**	
		#	Tests	Cov	Time	Tests	Cov	Time	Tests	Cov	Time	Tests	Cov
QuickGraph	OVERALL	1119	10140	51.2	0.2	334	31.6	4.4	1923	**68.2**	3.2	21	26
	Algorithms	572	-	38.1	-	-	24.8	-	-	52.1	-	-	24.8
	Collections	269	-	87.7	-	-	17.8	-	-	94.0	-	-	11.2
	... (5 more)												
Dsa	OVERALL	665	10493	14.9	1.0	552	83.8	3.7	961	**90**	0.9	298	93.2
	Algorithms	198	-	41.9	-	-	100	-	-	100	-	-	88.3
	DataStruc.	433	-	0	-	-	76.7	-	-	86.4	-	-	90.8
	... (2 more)												
xUnit	OVERALL	2379	10148	24.9	6.1	1265	38.6	4.5	1360	41.1	2.0	282	62.7
	Gui	432	-	34.3	-	-	40.8	-	-	**46.1**	-	-	17.8
	Sdk	706	-	25.1	-	-	35.6	-	-	**40.2**	-	-	86.3
	... (6 more)												
NUnit	Util	1810	10129	16.1	1.7	816	35.3	7.5	1804	**43.5**	3.7	319	63.9
TOTAL		**5973**	**40910**	**26**	**9.0**	**2967**	**41.3**	**20.1**	**6048**	**52.3**	**9.8**	**920**	**59.2**

remaining columns, subcolumns "# Tests", "Cov", and "Time" show the number of tests generated by each approach (Randoop, Pex, Seeker, and Manually written tests), branch coverage achieved, and time taken in hours, respectively. Table 5.7 shows further details regarding the sequences generated by each approach. Columns "Avg.", "SD", and "Max" show the average lengths, standard deviation, and maximum lengths of sequences generated by each approach, respectively. We next summarize our results for each approach and next present lessons learned through our evaluations.

Results Summary

Randoop. Our results show that Randoop achieved the lowest coverage among all approaches for all applications, except for Quickgraph. For QuickGraph, Randoop achieved higher coverage than Pex. Randoop could not achieve any coverage for the `DataStructures` namespace of Dsa, since Randoop cannot handle generics. Furthermore, Table 5.7 shows that sequences generated by Randoop are often longer than the sequences generated by other approaches. In summary, our results show that target sequences cannot be generated by combining method calls randomly to form longer sequences.

Pex and Seeker. Our results show that Pex, which is a DSE-based approach, can effectively handle generation of primitive data, but cannot generate target sequences. For example, Pex achieved 100% coverage for the `algorithms` namespace of Dsa. This namespace does not require sequences and includes implementations of various sorting algorithms such as mergesort. On the other hand, Pex achieved only 31.6% for QuickGraph, which requires complex sequences

for achieving high coverage. In our evaluations, we used customized settings for Pex instead of default values, thereby favoring Pex compared to Seeker. For example, our settings help Pex run for 20 hours (for all subjects) compared to 9.8 hours for Seeker. Still, Seeker achieved 12% (653 new branches) higher branch coverage than Pex. Indeed, allowing Seeker run for more time could help achieve more coverage. Therefore, our results show that it is difficult to achieve high coverage by letting Pex run for more time, showing the significance of our Seeker approach.

Lessons Learned. Although Seeker achieved higher coverage than Randoop and Pex, the coverage achieved is still not close to 100%. Moreover, coverage achieved by Seeker is lower than manually written tests for all subjects, except for QuickGraph, which is the most complex subject. For QuickGraph, Seeker achieved 42.2% (472 new branches) higher branch coverage than manually written tests. This result shows that Seeker can generate complex sequences that are hard to be written manually. We next summarize lessons learned through our evaluations.

Path explosion. Although Seeker suggests shorter skeletons (as shown in Table 5.7), we identify that skeletons suggested by Seeker increase the number of paths to be explored by Pex. For example, for the `algorithms` namespace of QuickGraph, Seeker achieved 52.1%. Although Seeker suggested desired skeletons to Pex, Pex could not generate tests using those skeletons for this namespace. The primary reason is that Pex, by default, attempts to cover all feasible paths among method calls within the suggested sequences. In future work, we plan to address this issue by developing a search strategy that can guide Pex. The insight for our future work is that not all paths in the method calls of suggested skeletons need to be explored for producing desired object states.

Environment dependency. A primary reason for the low coverage achieved by Seeker for xUnit and NUnit is their dependency on environment, which is currently beyond the scope of Seeker. For example, in xUnit, majority of the classes requires assembly files that include tests or project files in XML formats. However, Seeker achieved 28.3% (121 new branches) higher coverage than manually written tests for the `Gui` namespace, which includes some classes that require sequences and do not depend on the environment. In future work, we plan to address this issue by combining Seeker with other approaches [82] that mock environments, thereby isolating the environment dependency.

Abstract classes, interfaces, and callback methods. All our subjects are libraries that include elements such as abstract classes or interfaces, whose implementations are often not available within those libraries. These libraries expect client applications to provide such implementations. For example, Dsa provides three abstract class such as `CommonBinaryTree`.

Table 5.7: Statistics of generated sequences.

Subject	**Randoop**			**Pex**			**Seeker**		
	Avg.	SD	Max	Avg.	SD	Max	Avg.	SD	Max
QuickGraph	21.6	21.6	191	4.4	3.0	14	5.6	2.9	17
Dsa	3.0	2.5	20	2.7	2.0	12	3.2	1.9	12
xUnit	6.1	5.8	65	3.3	4.9	58	2.4	2.0	37
NUnit	4.7	5.0	121	4.1	3.0	20	4.3	2.9	19

Table 5.8: Def-Use coverage achieved by Randoop, Pex, Seeker, and manually written tests.

Subject	**# Def-Use pairs**	**Randoop**		**Pex**		**Seeker**		**Manual**	
		# Covered	%	# Covered	%	# Covered	%	# Covered	%
QuickGraph	892	402	45.1	198	22.2	447	50.1	152	17.0
Dsa	583	0	0	96	16.5	222	38.1	185	31.7
xUnit	1256	196	15.6	316	25.2	357	28.4	24	1.9
TOTAL	**2731**	**598**	**21.9**	**610**	**22.3**	**1026**	**37.6**	**361**	**13.2**

Without these abstract classes, Seeker achieved 94.3% coverage (higher than manually written tests) for the `DataStructures` namespace of Dsa. Similarly, xUnit includes methods such as `ExecutorCallback.Wrap` that requires a callback method. We identify that manually written tests achieved higher coverage than Seeker, since those tests include necessary implementations. In future work, we plan to address this issue by developing a technique similar to mocking the environment.

RQ2: Def-Use Coverage

We next address the second research question on whether Seeker achieves higher def-use coverage compared to Pex and Randoop. Table 5.8 shows the def-use coverage achieved by Randoop, Pex, Seeker, and manually written tests, respectively. We could not apply our DUCover tool on tests generated for NUnit, due to a technical limitation of executing nunit tests using NUnit. Along with def-use coverage, we also measure all-defs coverage to provide more insights. All-defs criteria described that for each definition in the code under test, some use of the this definition is being exercised by a test input. Table 5.9 shows the all-defs coverage achieved by all approaches for each subject.

Our results show that Seeker achieved higher def-use and all-defs coverage compared to both Pex and Randoop, respectively, for all subjects. The results also show that Seeker achieved higher def-use coverage than manually written tests. A primary reason could be that the programmers may not write tests to achieve high def-use coverage. Although Seeker achieved higher def-use coverage than Pex and Randoop, the coverage achieved by Seeker is not close to 100%. There are two major reasons. First, some of the def-use pairs are infeasible. For example, consider the `Deque` class shown in Figure 5.22. In this class, the `m_deque` field is

```
00:public class Deque<T> {
01:   private DoublyLinkedList<T> m_deque;
02:   ...
03:   public override void Add(T item) {
04:      EnqueueBack(item);
05:   }
06:   public override void Clear() {
07:      m_deque.Clear();
08:      Count = 0;
09:   }
10:   public override T DequeFront() {
11:      Guard.InvalidOperation(Count == 0, Resources.DequeDequeueEmpty);
12:      T item = m_deque.Head.Value;
13:      m_deque.RemoveFirst();
14:      Count--;
15:      return item;
16:   }
17:   ...
18:}
```

Figure 5.22: The `Deque` class from Dsa.

Table 5.9: All defs coverage achieved by Randoop, Pex, Seeker, and manually written tests.

Subject	**# All Defs**	**Randoop**		**Pex**		**Seeker**		**Manual**	
		# Covered	%	# Covered	%	# Covered	%	# Covered	%
QuickGraph	136	97	71.3	65	47.8	109	80.1	31	22.8
Dsa	112	0	0	34	30.3	59	52.7	59	52.7
xUnit	922	97	10.5	144	15.6	156	16.9	13	1.4
TOTAL	**1170**	**194**	**16.6**	**243**	**20.8**	**324**	**27.7**	**103**	**8.8**

defined in Statement 7 in the `Clear` method. On the other hand, the `m_deque` field is accessed in Statement 12 in the `DequeFront` method, forming a def-use pair. However, this def-use pair is an infeasible pair, since the `Clear` method sets the value of the `Count` field to zero (Statement 8) and the `DequeFront` method includes an additional condition check (in Statement 11) that throws exception if the value of the `Count` field is zero. In future work, we plan to identify such infeasible pairs by constructing inter-procedural control-flow graphs and by using constraint solving to detect infeasible paths. Detecting such inter-procedural infeasible paths helps detect infeasible def-use pairs. Second, Seeker, which is developed around Pex, is primarily intended for achieving higher branch coverage rather than def-use coverage. In future work, we plan to develop a new search strategy for Seeker that guides Pex to achieve higher def-use coverage along with the branch coverage.

RQ3: Defects

We next address the second research question regarding comparing defect-detection capabilities of Randoop, Pex, and Seeker. Table 5.10 shows our results. Subcolumns "AT", "FT", and "D"

Table 5.10: Defects detected by all approaches.

Subject	Randoop			Pex			Seeker		
	All Tests	Failing Tests	Defects	All Tests	Failing Tests	Defects	All Tests	Failing Tests	Defects
QuickGraph	6956	456	10	334	14	11	1923	117	34
Dsa	687	17	3	552	34	15	961	61	20
xUnit	112	0	0	1265	12	5	1360	12	5
NUnit	528	76	3	816	10	7	1804	16	13
Total	**8283**	**549**	**11**	**2967**	**70**	**38**	**6048**	**206**	**72**

show the total number of generated tests, number of failing tests, and number of distinct defects detected, respectively, by each approach. For Randoop, due to the large number of failing tests, we regenerated tests with its default parameters, instead of analyzing all tests generated in Section 5.6.4. Furthermore, all our tests are automatically generated and do not include test oracles. Therefore, we used uncaught exceptions as test oracles with focus on robustness issues. In particular, we considered the tests that throw exceptions as failing tests. However, we considered the failing tests that throw *expected* exceptions as passing tests. Furthermore, we ignored the defects related to `NullReferenceExceptions` that are thrown by passing `null` values to arguments of public methods. The primary reason is that often open source applications do not check `null` values for the arguments of public methods, and can also be fixed by automatically adding a `null` check on arguments of all public methods. Since manually written tests of these subjects do not include any failing tests, we consider all defects detected by Randoop, Pex, and Seeker as new defects.

Our results show that Randoop, Pex, and Seeker detected 11, 38, and 72 defects, respectively. We reported detected defects on hosting websites of our subject applications. In all subjects, defects detected by Randoop are related to `NullReferenceExceptions`. Similarly, except for Dsa, all defects detected by Pex are also related to `NullReferenceExceptions`. In Dsa, Pex detected two and five defects related to `OverflowException` and `IndexOutOfRange` exceptions, respectively. Seeker detected all defects detected by Randoop and Pex, and also detected new defects related to `InvalidOperationException` in QuickGraph. This exception is thrown when an attempt to modify a collection is made after an enumerator is created on that collection. It requires specific method sequences to cause this exception. Furthermore, Seeker detected a defect related to an infinite loop in QuickGraph. Figure 5.23 shows the test that detected the infinite loop. The test includes five classes and six method calls. Along with the skeleton generated by Seeker, the values "0" and "1" generated by Pex in Statement 7 helped trigger the infinite loop in the `RandomGraph.Graph` method. In summary, our results show that Seeker has higher defect-detection capabilities compared to Randoop and Pex.

```
00: BidirectionalGraph bidGraph;
01: Random random;
02: VertexAndEdgeProvider s0 = new VertexAndEdgeProvider();
03: Vertex s1 = new Vertex();
04: bidGraph = new BidirectionalGraph((IVEProvider)s0,
             PexSafeHelpers.ByteToBoolean((byte)16));
05: bidirectionalGraph.AddVertex((IVertex)s1);
06: random = new Random();
07: RandomGraph.Graph((IEdgeMutableGraph)bidGraph, 0, 1, random, false);
```

Figure 5.23: A test (generated by Seeker) that detected infinite loop in QuickGraph.

Table 5.11: Branch coverage achieved by MSeqGen (M) and Seeker (S) for QuickGraph.

Namespace	**# Branches**	**Pex**	**M**	**S**	**M+S**
Algorithms	572	24.8	27.4	**52.1**	44.2
Collections	269	17.8	63.2	**94.0**	95.6
Concepts	51	39.2	74.5	74.5	74.5
Exceptions	5	80.0	80.0	80.0	80.0
Predicates	58	93.1	93.1	100	98.3
Providers	5	60.0	80.0	80.0	80.0
Representations	159	52.2	64.8	67.9	67.3
TOTAL	**1119**	**31.6**	**47.3**	**68.2**	**64.3**

RQ4: MSeqGen Comparison

We next address the third research question regarding comparing branch coverage achieved by Seeker with MSeqGen. MSeqGen took 1.3 hours to generate tests for QuickGraph. Table 5.11 shows our results. Columns "Pex", "M", and "S" show branch coverage achieved by Pex, MSeqGen, and Seeker, respectively. Column "M + S" shows branch coverage achieved by combining MSeqGen and Seeker. Although MSeqGen achieved higher coverage than Pex, our results show that Seeker achieved much higher coverage than MSeqGen, especially for complex namespaces such as `Algorithms` and `Collections`. There are two major reasons for the lower coverage of MSeqGen compared to Seeker. First, sequences extracted by MSeqGen from the existing code bases do not include sequences for many classes under test. For example, although we used 3.85MB of .NET assembly code for extracting sequences, none of these code bases include sequences for the `EdgeDoubleDictionary` or `EdgeStringDictionary` classes. Therefore, MSeqGen could not achieve any coverage for these classes. On the other hand, Seeker achieved 100% coverage for these two classes. Second, MSeqGen-extracted sequences are different from desired sequences required for producing desired object states. However, the amount of time taken by MSeqGen (1.3 hours) is quite lower than the amount of time taken by Seeker (3.2 hours) for generating tests.

In contrast to our original expectation, "M + S" achieved lower coverage than Seeker alone, except for the `Collections` namespace. In our inspection, we found that "M + S" often results in more sequences, thereby increasing the exploration space for Pex. Although we can address

this issue by using customized settings for Pex (similar to those used for RQ1), the current limitations of the Seeker prototype prevents from using such customized settings. In future work, we plan to combine both these approaches by improving the performance of Seeker. In summary, Seeker achieved higher branch coverage than MSeqGen, and unlike MSeqGen, Seeker does not require any additional information such as usage information.

5.7 Related Work

Our three approaches (MSeqGen, DyGen, and Seeker) are closely related to two major research areas: method sequence generation and regression testing. Existing approaches for method sequence generation in object-oriented testing can be further broadly classified into two major categories: *implementation-based* and *usage-based* approaches.

Implementation-based approaches. These approaches use the implementation information of classes under test for generating tests. These approaches can further be classified into two sub-categories: *direct construction* [89] and *sequence generation* [29, 59, 63, 86, 97, 128, 138].

The direct construction approaches such as Korat [89] construct desired object states by directly assigning values to member fields of classes under test. However, these approaches require specifications such as class invariants [76], which are rarely documented by developers. In contrast, Seeker is a sequence-generation approach and does not require class invariants.

Among sequence-generation approaches, Buy et al. [23] proposed an approach that generates sequences for exercising the def-use pairs associated with member fields of classes under test. Their approach can be used for testing classes in isolation to achieve def-use coverage. Similarly, *bounded-exhaustive* approaches [138] or evolutionary approaches [59, 128] can be used to test individual classes only and cannot generate target sequences that involve methods from multiple classes (as shown in their evaluations). In contrast to these approaches, Seeker can be used to test multiple classes together and can handle large real-world applications.

Randoop [97] is a random approach that generates sequences by randomly combining method calls. Zheng et al. [143] proposed a heuristic approach that assists a random approach with sequences that mutate object fields accessed by a MUT. However, due to the large search space of possible sequences, there is often a low probability for randomly generating target sequences. In contrast to these approaches, Seeker is a systematic approach that generates sequences incrementally based on the branches that are not yet covered, thereby significantly reducing the number of candidate target sequences.

Seeker is also related to the extended chaining approach proposed by McMinn and Holcombe [86]. Their approach identifies a sequence of methods that need to be executed to cover a target branch. Seeker significantly differs from their approach in two major aspects. First, their approach can handle only procedural-oriented code such as C and cannot handle object-oriented code that include additional challenges such as inheritance and nested classes. Second,

their approach requires users to provide a bound on the length of the desired sequence and method calls that can be included in that sequence. In contrast, Seeker does not require any manual effort and automatically infers required information from the branches that are not yet covered by DSE.

Usage-based approaches. Jaygarl et al. proposed OCAT [61] that captures object states dynamically during program executions and reuses captured object states to assist a random approach. A major issue with OCAT is that these approaches require system tests for capturing object states and sequences, respectively. Furthermore, captured object states or sequences can be different from the desired ones. Although OCAT includes a mutation technique, the mutation technique requires class invariants to effectively mutate private fields. Seeker complements this approach and does not require any additional information. Furthermore, Seeker can also handle private fields effectively through method-call graphs.

An approach, called UnitPlus [114], captures sequences in existing test code and suggests those sequences to developers in reducing the effort of writing new unit tests. In contrast to UnitPlus, MSeqGen captures sequences from existing code bases but uses those sequences for assisting test-generation approaches. Unlike existing approaches that replay exactly the same captured behavior, MSeqGen replays beyond the captured behavior using techniques such as sequence generalization or generating new sequences by combining extracted sequences.

MSeqGen is also related to another category of approaches based on mining source code [4, 38, 131]. These approaches mine code bases statically and extracts frequent patterns as implicit programming rules. These approaches use mining algorithms such as frequent itemset mining [130] or association rule mining [6] for extracting frequent patterns. These mined programming rules are used for assisting programmers in writing code or detecting violations in an application under analysis. MSeqGen also uses static analysis for extracting patterns as sequences that can produce objects of receiver or arguments types of a MUT. Unlike these existing approaches, MSeqGen uses extracted sequences in a novel way for assisting test-generation approaches in achieving high structural coverage such as branch coverage.

Regression testing. There exist approaches [37, 96, 106] that use a capture-and-replay strategy for generating regression tests. In the capture phase, these approaches monitor the methods called during program execution and use these method calls in the replay phase to generate unit tests. DyGen also uses a strategy similar to the capture-and-replay strategy, where DyGen captures dynamic traces during program execution and use those traces for generating regression tests. However, unlike existing approaches that replay exactly the same captured behavior, DyGen replays beyond the captured behavior by using DSE in generating new regression tests.

Another existing approach, called Orstra [136], augments an existing test suite with additional assertions to detect regression faults. To add these additional assertions, Orstra executes

a given test suite and collects the return values and receiver object states after the execution of methods under test. Orstra generates additional assertions based on the collected return values or receiver object states. DyGen also uses a similar strategy for generating assertions in the regression tests. Another category of existing approaches [31, 40, 117] in regression testing primarily target at using regression tests for effectively exposing the behavioral differences between two versions of a software. For example, these approaches target at selecting those regression tests that are relevant to portions of the code changed between the two versions of software. However, all these approaches require an existing regression test suite, which is the primary focus of our DyGen approach.

5.8 Chapter Summary

Generation of desirable method sequences for achieving high structural coverage of the code under test is a known challenging problem in unit testing of object-oriented code. In this chapter, we proposed three approaches, MSeqGen, DyGen, and Seeker, for addressing the preceding problem. MSeqGen addresses this problem from a novel perspective of incorporating other sources of information such as how method calls are used in practice. MSeqGen extracts method sequences based on how method calls are used in practice by mining code bases that use receiver or argument object types of a method under test. MSeqGen uses extracted sequences for assisting a random and a DSE-based approach. On the other hand, DyGen mines dynamic traces recorded during program executions and uses those traces to generate regression tests. Both MSeqGen and DyGen represent a step towards a new direction of leveraging research in the field of mining software engineering data to assist test generation, serving as a synergy between these two major research areas. In contrast to MSeqGen and DyGen, Seeker generates method sequences based on the implementation information of methods under test, thereby complementing MSeqGen and DyGen. In our evaluations, we showed that our approaches perform better than existing state-of-the-art approaches by achieving higher coverage than the state-of-the-art approaches. Such an improvement is significant since the branches that are not covered by these state-of-the-art approaches are generally quite difficult to cover.

Chapter 6

Future Work

In this dissertation, we presented a framework, called *WebMiner*, that integrates code searching and mining to achieve SE tasks such as detecting defects in applications under analysis. This research has demonstrated the effectiveness of leveraging information available on the web for improving both software productivity and quality. Inspired by our existing results, in future work, we plan to extend this research in four major directions. We next discuss our future work.

6.1 Mining Unstructured Software Engineering Data

In this research, we primarily focused on mining source code, which can be considered as a form of structured SE data. However, on the web, there exist other SE data, especially in the form of natural language, that includes valuable information useful to assist programmers. For example, developer forums such as Eclipse [36] and BCEL [16], or various technical blogs [98] include valuable information for assisting programmers in achieving different SE tasks such as learning about APIs or debugging an SE problem at hand [74]. This unstructured data could include additional information that helps address different SE tasks that cannot be effectively addressed using the source code alone. For example, an existing approach, called iComment [116], shows the significance of using code comments for detecting defects in applications under analysis. In particular, their approach extracts implicit programming rules by analyzing code comments and uses those rules to detect inconsistencies (between code comments and source code) that represent defects or bad comments in applications under analysis.

To the best of our knowledge, there exist no approach that effectively leverages the unstructured SE data available on the web. Therefore, to address this issue, in future work, we plan to develop a new general framework, called *WebMiner++*, that can leverage both structured and unstructured information available on the web. In particular, WebMiner++ includes additional

techniques based on Natural Language Processing (NLP) for analyzing the SE data available in the form of natural language on the web. We expect that this new synergy of NLP, searching, and mining could help address various new SE tasks that cannot be handled by our existing WebMiner framework. We next explain more details about our proposed approach.

Initially, we plan to conduct an empirical study on identifying what kinds of unstructured SE data are available on the web and what kinds of SE tasks can be effectively addressed by leveraging the available unstructured SE data. We expect that this empirical study can help address a research question on whether it is worthwhile to invest efforts in developing our proposed WebMiner++ framework. Apart from the empirical study, our WebMiner++ framework will help address the following fundamental research questions in leveraging unstructured data available on the web.

- **How to analyze and understand unstructured SE data?** A major challenge in dealing with unstructured SE data is to analyze and understand the data, since natural language can often be ambiguous [81]. To address this issue, in our future work, we plan to adapt existing state-of-the-art techniques such as word tagging, which targets at identifying the part-of-speech of each word in a sentence [51] or develop new techniques. We also plan to develop specific techniques based on the SE data under analysis. For example, in developer forums, the first message could be a question posted by a programmer and further messages within the same thread could be considered as responses for that question [74].

- **How to index analyzed data and search for relevant data?** Recently, code search engines (CSE) are introduced to effectively search code in the open source code available on the web. The primary reason for the introduction of CSEs is that normal search engines such as Yahoo (`www.yahoo.com`) and Google (`www.google.com`) mainly search based on the textual content of a file. However, a file including source code is more than just a textual file. For example, each word in the source code file has a different meaning, depending on several factors such as the programming language or the location of that word in the source code file. Therefore, to effectively search in available open source code, CSEs index source code based on the semantics of the corresponding programming language of the source code file. However, to the best of our knowledge, there exists no search engines that index unstructured SE data. Therefore, to address this issue, we plan to enhance CSEs by developing new techniques for indexing unstructured data along with the structured data.

- **How to accept inputs from programmers for achieving the SE task at hand?** This research question targets at developing new techniques on how to effectively capture programmer's requirements such as in a simple form of textual descriptions or more

advanced form of test cases. There exist approaches [72, 102] that accept a programmer's requirements in the form of test cases. These approaches further check whether the searched code examples meet the programmer's requirements by checking whether the test cases are passed. However, these approaches are limited to source code and cannot handle unstructured SE data. In future work, we plan to develop new techniques that can handle both structured and unstructured SE data.

- **How to mine searched data?** Often, there can be more than one candidate solution for achieving the SE task at hand. Among those candidate solutions, a few solutions are more frequently used compared to the others. Programmers can have higher confidence on such frequently used solutions than the other candidate solutions. To identify the frequently used solutions among candidate solutions, we plan to develop new techniques to transform unstructured searched data into an intermediate form suitable for applying data mining techniques. We also plan to adapt existing mining techniques such as association rule mining [6] or develop new techniques for identifying frequently used solutions among candidate solutions.

Apart from the preceding four fundamental research questions, inspired by existing approaches [12, 18], we also plan to explore another direction of exploiting multiple sources of information compared to using a single source of information for achieving SE tasks at hand.

6.2 Moving from Syntactic to Semantic Analysis

The current research in the area of mining SE data focuses primarily on identifying and clustering syntactically similar candidate patterns. For example, consider our PARSEWeb approach presented in Section 3.2. *PARSEWeb* accepts queries of the form "*Source* $\rightarrow$ *Destination*" and mines frequent method sequences that accept an object of the *Source* type and produces an object of the *Destination* type. To reduce manual effort, PARSEWeb clusters method sequences that are structurally similar by using heuristics. However, as shown in Rostra [137], two different method sequences that are structurally different can result in the same object state and can be considered similar semantically. For example, consider the two method sequences shown in Figure 6.1. The method `push` pushes an element on to the stack, whereas the method `pop` removes the top element from the stack. Although both these sequences are structurally different, the object states of `IntStack` produced by these two sequences are the same and can be considered as similar semantically.

To the best of our knowledge, there exists no mining technique that clusters method sequences that are semantically similar. Recent work by Jiang and Su [45] showed encouraging results in this direction. Inspired by Schwartz's randomized polynomial identity testing [110],

```
Sequence 1:
   IntStack s1 = new IntStack();
   s1.push(3);
   s1.push(2);
   s1.pop();
   s1.push(5);

Sequence 2:
   IntStack s2 = new IntStack();
   s2.push(3);
   s2.push(5);
```

Figure 6.1: Two method sequences that produce the same object state [137].

their approach uses random testing to identify code snippets that exhibit the same input and output behaviors, in contrast to the previous approaches [15,45] that primarily focus on detecting code snippets that are syntactically similar. Inspired by their initial results, in our future work, we plan to integrate dynamic analysis into data mining algorithms to group candidates that are similar both syntactically and semantically. For example, two method sequences including different method calls resulting in the same end object state can be considered as similar sequences. We plan to use existing state-of-the-art testing techniques [117,126,138] to identify similar sequences based on their resulting end object states. We expect that the synergy of static and dynamic analyses can open a new research direction and help overcome some of the existing limitations in mining SE data.

6.3 Combining Static Verification and Dynamic Testing

The long-term goal of this research direction is to improve software quality by statically detecting specification violations and dynamically confirming those violations as real defects, primarily targeting at combining our work presented in Chapters 4 and 5. We expect that such an integrated approach can address the weaknesses of existing state of the art that primarily uses either static or dynamic verification techniques individually. For example, dynamic verification techniques suffer from scalability issues when applied to large applications. On the other hand, static verification techniques, despite scalable, suffer from a large number of false warnings due to their conservative nature. Therefore, statically detecting specification violations and dynamically confirming those violations can address both false-warning and scalability issues. Detection of specification violations using static or dynamic verification techniques requires specifications that describe programming rules to be followed while writing source code. Without specifications, these techniques can detect only robustness-related defects such as division by zero or dereferencing a null pointer. The goal of this research direction is to mine specifications automatically from existing code bases and use those specifications to statically detect violations and dynamically confirm those violations.

```
00:String removeDoubleEntries(Matcher matcher) {
01:   ...
02:   ArrayList entries = new ArrayList();
03:   while (matcher.find()) {
04:      entries.add(matcher.group()); }
05:   Iterator it = entries.iterator(); ...
06:   String last = (String) it.next(); ...
07:}
```

Figure 6.2: A code example from the Columba application with a potential defect.

```
00:String removeDoubleEntries(Matcher matcher) {
01:   ...
02:   ArrayList entries = new ArrayList();
03:   while (matcher.find()) {
04:      entries.add(matcher.group()); }
05:   Iterator it = entries.iterator(); ...
06:   if(it.hasNext()) {
07:      String last = (String) it.next(); ...
08:   }
09:}
```

Figure 6.3: Modified code example with additional condition check.

We next use a real code example shown in Figure 6.2 from an open source e-mail client application, called Columba[1], as an illustrative code example. The method `removeDoubleEntries` accepts an object of the `Matcher` class, which matches a string against a pattern. The method `Matcher.find` (Statement 3) scans the input string and identifies substrings that match the pattern. This code example includes a potential defect, where the `Iterator.next` method (Statement 6) throws `NoSuchElementException` when there are no entries in the `ArrayList` object. Figure 6.3 shows the modified code example that includes an additional condition check in Statement 6 that helps prevent the exception being thrown. We next explain how we plan to statically detect violations in code examples such as the code example shown in Figure 6.2 and dynamically confirm those violations.

Mining specifications. Detecting specification violations using static or dynamic verification techniques requires specifications that describe programming rules to be followed while writing source code. Since these specifications are often not well documented in practice [73], we plan to use our Alattin [121] (Section 4.3) approach to automatically mine specifications from existing code bases. Alattin primarily applies data mining techniques on relevant code examples of API methods such as `Iterator.next` to mine high-level specifications in the form of common patterns (e.g., frequent occurrences of pairs or sequences of API method calls). A specification for the `Iterator.next` method that can be mined by Alattin is "P_1: `boolean-check` on return

[1] `http://sourceforge.net/projects/columba/`

```
00:String removeDoubleEntries(Matcher matcher) {
01:   ...
02:   ArrayList entries = new ArrayList();
03:   while (matcher.find()) {
04:      entries.add(matcher.group()); }
05:   Iterator it = entries.iterator(); ...
06:   if(entries.size() > 0) {
07:      String last = (String) it.next(); ...
08:   }
09:}
```

Figure 6.4: Modified code example with a new condition check.

```
00:String removeDoubleEntries(Matcher matcher) {
01:   ...
02:   ArrayList entries = new ArrayList();
03:   while (matcher.find()) {
04:      entries.add(matcher.group()); }
05:   Iterator it = entries.iterator(); ...
06:   if(entries.size() > 0) {
07:      if (it.hasNext() == false)
08:         Assert.fail();
09:      String last = (String) it.next(); ...
10:   }
11:}
```

Figure 6.5: The code example with a test target.

of `Iterator.hasNext` before `Iterator.next`". The preceding specification describes that there should be a boolean condition check on the `Iterator.hasNext` method before the `Iterator.next` method (as shown in Figure 6.3).

Detecting violations. Given specifications mined by approaches such as Alattin, we plan to use existing static verification tools such as Findbugs [58] to statically detect violations in applications under analysis. For example, given the specification P_1 and the code example shown in Figure 6.2, static-verification tools can detect a violation in Statement 6, since there is no condition check before the `Iterator.next` method. However, static-verification tools often produce false warnings due to their conservative nature. For example, consider another modified code example shown in Figure 6.4. The static-verification tool detects a violation in this code example, since the code example does not include a condition check on the `Iterator.hasNext` method. However, due to the additional condition check `if(entries.size() > 0)` in Statement 6, the violation detected by the static-verification tool turns out to be a false warning. There can be several other reasons for false warnings such as infeasible paths. We next describe how we address the issue of false warnings by dynamically generating test inputs to confirm statically detected violations.

Table 6.1: Different pattern formats and approaches using those pattern formats.

Pattern Format / SE Task	PARSEWeb	CAR-Miner	Alattin	MSeqGen	DyGen
Sequences	X			X	X
Sequence Association Rules		X			
Alternative Patterns			X		

Setting up test targets. To avoid false warnings, we plan to add additional branches in the code that negates the necessary condition check and define those new branches as test targets. Figure 6.5 shows the new branch added in Statements 7 and 8. Achieving this test target can help confirm the violation, since the test target confirms that the `entries` object does not include any elements and the next method call `Iterator.next` in Statement 9 throws `NoSuchElementException`. We next describe how we plan to generate test inputs dynamically for achieving these test targets, thereby confirming violations.

Confirming violations. To generate test inputs automatically for achieving test target, method sequences are required to produce desired object state for the `Matcher` object. In this code example, the desired object state is one where the input string does not contain any substrings that match the pattern. To address this issue of producing desired object states, we plan to use our MSeqGen (Section 5.4) and Seeker (Section 5.6) approaches.

6.4 Cross-Cutting Analysis of Patterns and Approaches

In this dissertation, we presented six approaches based on our WebMiner framework. Among these six approaches, five approaches (PARSEWeb [119], CAR-Miner [122], Alattin [121], MSeqGen [125], and DyGen [118]) mine patterns in different pattern formats and use those patterns for achieving corresponding SE tasks under analyses, respectively. However, a pattern format such as the sequence pattern format used in one approach can also be applicable to another approach and can help improve the effectiveness of that approach. The goal of this research direction is to explore how different pattern formats proposed/used in this dissertation for one approach can help improve other approaches.

Table 6.1 shows pattern formats used in this dissertation and approaches where those pattern formats are currently used. For example, the sequence pattern format is used in PARSEWeb, MSeqGen, and DyGen approaches. We next explain how the pattern formats that are already used by one approach can help improve other approaches and also explain how other pattern formats that are not currently used in this dissertation can be further leveraged. Consider the sequence pattern format shown in Table 6.1, where patterns are represented in the form of sequences of method calls. Among our approaches, PARSEWeb, MSeqGen, and DyGen mine patterns in the form of sequences of method calls. However, the sequence patterns mined

by PARSEWeb and DyGen include multiple object types, whereas patterns mined by MSeqGen include only one object type. In future work, we plan to explore how sequence patterns with multiple object types can help improve MSeqGen. Currently, we rely on underlying test-generation approach such as Pex [126] to combine sequences of individual object types to form a sequence of multiple object types. We expect that the sequences with multiple object types can help reduce the exploration space of a test-generation tool. However, these sequences could also reduce the possible number of desired states that can be generated. We also plan to explore other pattern formats such as finite state automata [8]. For example, mining patterns in the form of finite state automata can help generate more desired states compared to sequences. However, in contrast to patterns in the form of sequences, patterns in the form of finite state automata may increase the exploration space of test-generation approaches, posing additional challenges.

Next, consider alternative patterns proposed in our Alattin approach. The primary advantage of alternative patterns is that these patterns help reduce false positives among detected violations. Therefore, in our future work, we plan to reuse these patterns for our CAR-Miner approach to reduce false positives among detected exception-handling violations. In particular, we plan to extend sequence association rules (proposed in our CAR-Miner approach) to include alternative patterns as follows. Currently, sequence association rules are of the form "$(FC_c^1 ... FC_c^n) \wedge FC_a \Rightarrow (FC_e^1 ... FC_e^m)$". This sequence association rule describes that function call FC_a should be followed by function-call sequence $FC_e^1 ... FC_e^m$ in exception paths only when preceded by function-call sequence $FC_c^1 ... FC_c^n$. In future work, we plan to extend these rules with alternative patterns for mining rules of the form "$(FC_c^1 ... FC_c^n) \wedge FC_a \Rightarrow (FC_{e1}^1 ... FC_{e1}^{m_1}) \vee (FC_{e2}^1 ... FC_{e2}^{m_2}) \vee \ ... \ \vee (FC_{ek}^1 ... FC_{ek}^{m_k})$". In the preceding rule, $(FC_{e1}^1 ... FC_{e1}^{m_1})$ represent an alternative sequence. We expect that these extended sequence association rules could help reduce false positives among violations detected by our CAR-Miner approach.

Along with these preceding pattern formats, in our future work, we also plan to use advanced graph representations such as *multi-layered* graphs and develop/adapt graph mining algorithms. The primary advantage of the graph pattern format compared to the pattern formats used in our dissertation is that graphs help represent additional information such as control-flow and data-flow information [95]. We expect that leveraging such advanced graph representations can help address new SE tasks that cannot be addressed using the existing pattern formats used in our dissertation. For example, we can extend PARSEWeb to mine inter-framework patterns that describe interactions among classes that belong to different frameworks, instead of intra-framework patterns.

Chapter 7

Assessment and Conclusion

7.1 Conclusion

Mining software engineering data (MSED) is a new research area in software engineering introduced a decade ago. MSED primarily targets at applying data mining techniques on SE artifacts to achieve various SE tasks such as detecting defects in applications under analysis by automatically inferring API usage specifications. In this dissertation, we advanced this research area by expanding the date scope to the large amount of open source code available on the web. In particular, we proposed a general framework, called WebMiner, that leverages a code search engine (CSE) for collecting relevant code examples of an API method under analysis from the web. WebMiner uses collected code examples for achieving various SE tasks, thereby improving software productivity and quality. WebMiner includes a novel technique, called partial-program analysis, that does not require the code examples to be compilable, since often the code examples collected from CSEs cannot be compiled due to lack of other source files that the code examples are dependent upon. Our partial-program analysis further alleviates the requirements of existing static analyses techniques that often require the source code under analysis to be compilable. Being light-weight, our analysis is highly scalable and can handle a large number of code examples ($\approx$ 96 million LOC as shown in our CAR-Miner evaluation [122]).

In this dissertation, we showed the effectiveness of our WebMiner framework, by developing six approaches (based on our framework) that address different SE tasks. The evaluation results of our approaches showed the effectiveness of expanding data scope to the large amount of open source code via addressing new programmer queries or detecting new defects that were not detected by related existing approaches. In particular, we showed that our PARSEWeb approach [119] addresses queries posted in developer forums and performs better than two related approaches [56, 80], thereby helping improve productivity of programmers. On the other hand, we showed that our CAR-Miner approach [122] mines 294 real programming rules

in five real-world applications (including 285 KLOC) and also detects 160 defects, where 87 defects are new defects that are not found by a related approach [133]. In our approaches, we also showed the effectiveness of developing new mining algorithms rather than being constrained by the existing off-the-shelf mining algorithms. For example, in our Alattin approach [121,123], we developed new mining algorithms that mine patterns in new formats such as *Or* and *Xor* pattern formats, and their combinations. We also showed how these different pattern formats help reduce false positives and false negatives among violations detected in applications under analysis.

In this dissertation, we took a step forward in developing a new approach, called MSeqGen [125], that represents a new direction of leveraging research in the area of MSED to assist test generation, serving as a synergy between these two major research areas. In particular, in our MSeqGen approach [125], we showed that the method sequences mined from existing code bases help improve existing state-of-the-art test-generation approaches [97, 126] in achieving higher structural coverage than without using mined method sequences. Although we focus primarily on open source code in this dissertation, our techniques are general and can be applied on large proprietary code bases such as Microsoft code bases as shown in the evaluation results of our DyGen approach [118].

7.2 Risk Analysis

We next present a few risks involved in using the approaches presented in this dissertation. A major underlying observation of our dissertation is that majority of the programmers correctly adhere to API usage specifications. Based on this observation, our approaches presented in this dissertation apply data mining techniques to identify common patterns and use those common patterns as API usage specifications. However, in this dissertation, all our approaches, except DyGen, are applied on open source code available on the web. Since some of the code bases available on the web may not be of high quality, our approaches that primarily rely on such code bases for inferring API usage specifications may infer incorrect patterns. For example, consider that an API method is not popular and is used incorrectly by only a few code bases that are not of high quality. Our approaches may infer the incorrect usage as a common pattern and suggest the same for programmers, thereby propagating the incorrect usage further. This issue may not happen when our approaches are applied on proprietary code bases such as Microsoft code bases. Furthermore, in open source code bases, this issue can be alleviated by restricting the code bases that can be used by the search phase of our WebMiner framework. For example, Google code search provides a search option, called package, where the search can be restricted to a few code bases (such as Eclipse code bases) that are of high quality. We expect that these risks can further be alleviated by conducting more evaluations using other code search engines such as Koders [69] and with a wide variety of subject applications.

7.3 Lessons Learned

We next summarize some lessons learned through our research and hope that these lessons would help other researchers and also our future research.

Existing artifacts. A major lesson learned from our research is that reusing existing artifacts often helps address various problems in software engineering. In particular, we reused existing open source code available on the web to infer API usage specifications that can help improve both software productivity and quality. However, in a few scenarios, existing artifacts such as method sequences may not help as shown in the evaluation results of our MSeqGen approach [125]. In those scenarios, additional techniques such as our Seeker approach [124] are required to further address the SE task at hand.

Combinations of techniques. Our dissertation demonstrated the potential of combining techniques from different research areas. A primary advantage is that the techniques from one research area and their sources of information can help address limitations of the techniques in the other research area. Our WebMiner framework serves as an example for the effectiveness of combining techniques from research areas, where we combined techniques from code searching and data mining. Furthermore, in our MSeqGen approach, we combined techniques from MSED and software testing.

Scalability and precision. Improving scalability of a technique often requires compromising precision to a certain extent. Through our dissertation, we learned that sacrificing precision to a certain extent for improving the scalability to a large amount of code could still help achieve good results. For example, the heuristics proposed in our partial-program analysis technique are not complete (as described in Section 2.3). However, our evaluation results show that our approaches developed based on our WebMiner framework performed better than the related approaches.

Problem-driven methodology. Our research demonstrated the effectiveness of using a problem-driven methodology rather than a solution-driven methodology. For example, in our CAR-Miner and Alattin approaches, we empirically investigated problems in the SE domain and identified required types of patterns for addressing those problems. We further developed new mining algorithms for mining these required types of patterns, rather than being constrained by existing off-the-shelf mining algorithms from the data mining community.

Although our dissertation took the MSED research area a step forward by combining techniques from searching and mining to mine structured data such as source code, there exist a

large amount of knowledge on the web in the form of unstructured data. In future work, we plan to develop a new framework that leverages this unstructured SE data available on the web to address other SE tasks.

REFERENCES

[1] Mithun Acharya. *Mining API Specifications from Source Code for Software Reliability*. PhD thesis, North Carolina State University, 2009.

[2] Mithun Acharya and Tao Xie. Mining API error-handling specifications from source code. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 370–384, 2009.

[3] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 25–34, 2007.

[4] Mithun Acharya, Tao Xie, and Jun Xu. Mining interface specifications for generating checkable robustness properties. In *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE)*, pages 311–320, 2006.

[5] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. pages 307–328, 1996.

[6] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[7] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 98–109, 2005.

[8] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 4–16, 2002.

[9] Saswat Anand, Corina S. Pasareanu, and Willem Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 134–138, 2007.

[10] J.D. Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy. *The Java Developer's Guide to Eclipse*. Addison-Wesley Professional, 2004.

[11] The Apache Jakarta Project, 2007. `http://jakarta.apache.org/regexp/`.

[12] Alberto Bacchelli, Michele Lanza, and Romain Robbes. Linking e-mails and source code artifacts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 375–384, 2010.

[13] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 681–682, 2006.

[14] Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the shape of Java software. *SIGPLAN Not.*, 41(10):397–412, 2006.

[15] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the 14th International Conference on Software Maintenance (ICSM)*, pages 368–377, 1998.

[16] Jakarta BCEL user forum, 2001. `http://mail-archives.apache.org/mod_mbox/jakarta-bcel-user/200609.mbox/thread`.

[17] Dev2Dev Newsgroups by developers, for developers, 2006. `http://forums.bea.com/bea/message.jspa?messageID=202265042&tstart=0`.

[18] Andrew Begel, Khoo Yit Phang, and Thomas Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *Proceedings of the 32th International Conference on Software Engineering (ICSE)*, pages 125–134, 2010.

[19] Black duck's web page with koders usage information, 2010. `http://corp.koders.com/about/`.

[20] Barry Boehm. A view of 20th and 21st century software engineering. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 12–29, 2006.

[21] Jan Bosch, Peter Molin, Michael Mattsson, and PerOlof Bengtsson. Object-oriented framework-based software development: problems and experiences. *ACM Comput. Surv.*, 32(3):3–8, 2000.

[22] Douglas Burdick, Manuel Calimlim, and Johannes Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, pages 443–452, 2001.

[23] Ugo Buy, Alessandro Orso, and Mauro Pezze. Automated testing of classes. *SIGSOFT Softw. Eng. Notes*, 25(5):39–48, 2000.

[24] Ray-Yaung Chang, Andy Podgurski, and Jiong Yang. Finding what's not there: A new approach to revealing neglected conditions in software. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)*, pages 163–173, 2007.

[25] Lori Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[26] S. Clarke. Measuring API usability. *Dr. Dobbs Journal*, May 2004:S6–S9, 2004.

[27] Codease, 2005. `http://www.codase.com/`.

[28] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[29] Christoph Csallner and Yannis Smaragdakis. JCrasher: An automatic robustness tester for Java. *Softw. Pract. Exper.*, 34(11):1025–1050, 2004.

[30] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming Language Design and Implementation (PLDI)*, pages 59–69, 2001.

[31] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.

[32] Amit Deshpande and Dirk Riehle. The total growth of open source. In *Open Source Development, Communities and Quality*, volume 275, pages 197–209, 2008.

[33] Prem Devanbu, Sakke Karstu, Walcélio Melo, and William Thomas. Analytical and empirical evaluation of software reuse metrics. In *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, pages 189–199, 1996.

[34] J. Duran and M. Ntafos. An evaluation of random testing. *IEEE Trans. Softw. Eng.*, 10(4):438–444, 1984.

[35] Eclipse, 2010. `http://www.eclipse.org/`.

[36] Eclipse developer forum, 2010. `http://www.eclipse.org/forums/`.

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

[38] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of 18th Symposium on Operating System Principles (SOSP)*, pages 57–72, 2001.

[39] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

[40] Robert B. Evans and Alberto Savoia. Differential testing: A new approach to change detection. In *Proceedings of the 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 549–552, 2007.

[41] Facebook developer toolkit, 2008. `http://www.codeplex.com/FacebookToolkit`.

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

[43] Chen Fu and Barbara G. Ryder. Exception-chain analysis: Revealing exception handling architecture in Java server applications. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 230–239, 2007.

[44] Chen Fu, Barbara G. Ryder, Ana Milanova, and David Wonnacott. Testing of Java web services for robustness. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 23–33, 2004.

[45] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 321–330, 2008.

[46] Mark Gabel and Zhendong Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 339–349, 2008.

[47] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag New York, Inc., 1997.

[48] Google Code Search Engine, 2006. `http://www.google.com/codesearch`.

[49] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

[50] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS)*, 2008.

[51] Sharon Goldwater and Tom Griffiths. A fully bayesian approach to unsupervised part-of-speech tagging. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics (ACL)*, pages 744–751, 2007.

[52] J. Hammond. What developers think, 2010. `http://www.drdobbs.com/architect/222301141/`.

[53] J. Han and M. Kamber. *Data mining: concepts and techniques*. Morgan Kaufmann Publishers Inc., 2000.

[54] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of Software Engineering (FSE)*, pages 154–163, 1994.

[55] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive interfaces. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 31–40, 2005.

[56] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 117–125, 2005.

[57] Reid Holmes and Robert J. Walker. Informing Eclipse API production and consumption. In *Proceedings of the 2007 OOPSLA workshop on Eclipse Technology eXchange (ETX)*, pages 70–74, 2007.

[58] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[59] Kobi Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 297–306, 2008.

[60] Package java.util, 2010. `http://download.oracle.com/javase/1.4.2/docs/api/java/util/package-summary.html`.

[61] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. OCAT: object capture-based automated testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*, pages 159–170, 2010.

[62] Henkel Johannes and Diwan Amer. Discovering algebraic specifications from java classes. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 431–456, 2003.

[63] Parasoft. Jtest manuals version 5.1. Online manual, 2006. `http://www.parasoft.com`.

[64] Viljamaa Jukka. Reverse engineering framework reuse interfaces. In *Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 217–226, 2003.

[65] Jung the Java Universal Network/Graph Framework, 2005. `http://jung.sourceforge.net/`.

[66] JUnit, 2001. `http://www.junit.org`.

[67] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 553–568, 2003.

[68] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[69] The Koders source code search engine, 2005. `http://www.koders.com`.

[70] Sen Koushik, Marinov Darko, and Agha Gul. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005.

[71] The Krugle code search for developers, 2006. `http://www.krugle.com`.

[72] Otávio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Lopes. Applying test-driven code search to the reuse of auxiliary functionality. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*, pages 476–482, 2009.

[73] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. In *IEEE Software*, pages 35–39, 2003.

[74] Wei Li, Charles Zhang, and Songlin Hu. G-finder: routing programming questions closer to the experts. In *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 62–73, 2010.

[75] Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software codes. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 306–315, 2005.

[76] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[77] Xiyang Liu, Hehui Liu, Bin Wang, Ping Chen, and Xiyao Cai. A unified fitness function calculation rule for flag conditions to improve evolutionary testing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 337–341, 2005.

[78] V. Benjamin Livshits and Thomas Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 296–305, 2005.

[79] Logic Project based on Eclipse GEF, 2006. `http://www.eclipse.org/downloads/download.php?file=/tools/gef/downloads/drops/R-3.2.1-&200609211617/GEF-examples-3.2.1.zip`.

[80] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 48–61, 2005.

[81] Christopher D. Manning, Hinrich Schiitze, and Lillian Lee. Foundations of statistical natural language processing, 2001.

[82] Madhuri R Marri, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. An empirical study of testing file-system-dependent software with mock objects. In *Proceedings of Automation of Software Test (AST)*, pages 149–153, May 2009.

[83] Vincenzo Martena, Alessandro Orso, and Mauro Pezzè. Interclass testing of object oriented software. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 145–154, 2002.

[84] Robert C. Martin. The Open Closed Principle. *j-C-PLUS-PLUS-REPORT*, 8(1):37–43, 1996.

[85] Y. Matsumoto. *A Software Factory: An Overall Approach to Software Production.* In P. Freeman ed., Software Reusability. IEEE CS Press, 1987.

[86] Phil McMinn and Mike Holcombe. Evolutionary testing of state-based programs. In *Proceedings of the 2005 conference on Genetic and Evolutionary Computation (GECCO)*, pages 1013–1020, 2005.

[87] Marcilio Mendonca, Paulo Alencar, Toacy Oliveira, and Donald Cowan. Assisting aspect-oriented framework instantiation: Towards modeling, transformation and tool support. In *Companion to the 20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 94–95, 2005.

[88] Amir Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 167–176, 2000.

[89] Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. Korat: A tool for generating structurally complex test inputs. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 771–774, 2007.

[90] Audris Mockus. Large-scale code reuse in open source software. *International Workshop on Emerging Trends in FLOSS Research and Development*, 0:7, 2007.

[91] Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 282–292, 2004.

[92] Amit A. Nanavati, Krishna P. Chitrapura, Sachindra Joshi, and Raghu Krishnapuram. Mining generalised disjunctive association rules. In *Proceedings of the 10th International Conference on Information and Knowledge Management (CIKM)*, pages 482–489, 2001.

[93] Cornelius Ncube, Patricia Oberndorf, and Anatol W. Kark. Opportunistic software systems development: Making systems from what's available. *Software, IEEE*, 25(6):38–41, nov. 2008.

[94] .NET framework 4.0 statistics, 2006. `http://geekswithblogs.net/sdorman/archive/2010/07/10/interesting-.net-framework-4-statistics.aspx`.

[95] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*, pages 383–392, 2009.

[96] Alessandro Orso and Bryan Kennedy. Selective capture and replay of program executions. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[97] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[98] .NET technical blog, 2010. `http://blog.dotnetwiki.org/`.

[99] Wolfgang Pree. Meta patterns - a means for capturing the essentials of reusable object-oriented design. In *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP)*, pages 150–162, 1994.

[100] QuickGraph: A 100% C# graph library with Graphviz Support, Version 2.0, 2008. `http://www.codeproject.com/KB/miscctrl/quickgraph.aspx`.

[101] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Path-sensitive inference of function precedence protocols. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 240–250, 2007.

[102] Steven P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 243–253, 2009.

[103] Martin P. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):26–34, 2009.

[104] Martin P. Robillard and Gail C. Murphy. Analyzing exception flow in Java programs. In *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 322–337, 1999.

[105] David S. Rosenblum and Elaine J. Weyuker. Predicting the cost-effectiveness of regression testing strategies. *SIGSOFT Softw. Eng. Notes*, 21(6):118–126, 1996.

[106] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for Java. In *Proceedings of the 20th Annual International Conference on Automated Software Engineering (ASE)*, pages 114–123, 2005.

[107] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. Detecting similar Java classes using tree algorithms. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR)*, pages 65–71, 2006.

[108] Naiyana Sahavechaphan and Kajal Claypool. XSnippet: Mining for sample code. In *ACM SIGPLAN symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 413–430, 2006.

[109] Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 471–480, 2008.

[110] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.

[111] Kazuhiro Shimizu and Takao Miura. Disjunctive sequential patterns on single data sequence and its anti-monotonicity. In *Proceedings of the 4th International Conference on Machine Learning and Data Mining in Pattern Recognition (MLDM)*, pages 376–383, 2005.

[112] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)*, pages 174–184, 2007.

[113] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Trans. Softw. Eng.*, 26(9):849–871, 2000.

[114] Yoonki Song, Suresh Thummalapenta, and Tao Xie. UnitPlus: Assisting developer testing in eclipse. In *Proceedings of the 2007 OOPSLA workshop on Eclipse Technology eXchange (ETX)*, pages 26–30, 2007.

[115] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 529–539, 2007.

[116] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*iComment: bugs or bad comments?*/. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 145–158, 2007.

[117] Kunal Taneja and Tao Xie. DiffGen: Automated regression unit-test generation. In *Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 407–410, 2008.

[118] Suresh Thummalapenta, Jonathan de Halleux, Nikolai Tillmann, and Scott Wadsworth. Dygen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces. In *Proceedings of the 4th International Conference on Tests and Proofs (TAP)*, pages 77–93, 2010.

[119] Suresh Thummalapenta and Tao Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proceedings of 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 204–213, 2007.

[120] Suresh Thummalapenta and Tao Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008.

[121] Suresh Thummalapenta and Tao Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 283–294, 2009.

[122] Suresh Thummalapenta and Tao Xie. Mining exception-handling rules as sequence association rules. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 496–506, 2009.

[123] Suresh Thummalapenta and Tao Xie. Alattin: Mining alternative patterns for defect detection. Technical Report TR-2010-21, North Carolina State University Department of Computer Science, Raleigh, NC, October 2010.

[124] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, and Peli de Halleux. Seeker: Demand-driven method-sequence generation for object-oriented unit testing. In *Under submission*, 2011.

[125] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 193–202, 2009.

[126] Nikolai Tillmann and Jonathan de Halleux. Pex white box test generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs (TAP)*, pages 134–153, 2008.

[127] Nikolai Tillmann and Wolfram Schulte. Parameterized Unit Tests. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 253–262, 2005.

[128] Paolo Tonella. Evolutionary testing of classes. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.

[129] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, 2004.

[130] Jianyong Wang and Jiawei Han. BIDE: Efficient mining of frequent closed sequences. In *Proceedings of 20th International Conference on Data Engineering (ICDE)*, pages 79 – 88, 2004.

[131] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 35–44, 2007.

[132] Westley Weimer and George Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 419–431, 2004.

[133] Westley Weimer and George Necula. Mining temporal specifications for error detection. In *Proceedings of 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 461–476, 2005.

[134] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes*, 27(4):218–228, 2002.

[135] Chadd C. Williams and Jeffrey K. Hollingsworth. Recovering system specific rules from software repositories. In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR)*, pages 1–5, 2005.

[136] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proceedings of 20th European Conference on Object-Oriented Programming (ECOOP)*, pages 380–403, 2006.

[137] Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proceedings of the 19th IEEE international conference on Automated Software Engineering (ASE)*, pages 196–205, 2004.

[138] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381, 2005.

[139] Tao Xie and Jian Pei. MAPO: Mining API usages from open source repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR)*, pages 54–57, 2006.

[140] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 282–291, 2006.

[141] Yunwen Ye and Gerhard Fischer. Supporting reuse by delivering task relevant and personalized information. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 513–523, 2002.

[142] Lizhuang Zhao, Mohammed J. Zaki, and Naren Ramakrishnan. BLOSOM: A framework for mining arbitrary boolean expressions. In *Proceedings of the 12th International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 827–832, 2006.

[143] Wujie Zheng, Qirun Zhang, Michael Lyu, and Tao Xie. Random unit-test generation with MUT-aware sequence recommendation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 293–296, 2010.