Analyzing repetitiveness in big code to support software maintenance and evolution

Hoan Anh Nguyen
Iowa State University
Analyzing repetitiveness in big code
to support software maintenance and evolution

by

Hoan Anh Nguyen

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Tien N. Nguyen, Major Professor
Samik Basu
Manimaran Govindarasu
Suraj C. Kothari
Hridesh Rajan
Akhilesh Tyagi

Iowa State University
Ames, Iowa
2015
Copyright © Hoan Anh Nguyen, 2015. All rights reserved.
# TABLE OF CONTENTS

LIST OF TABLES ................................................................. vii
LIST OF FIGURES ............................................................... ix
ACKNOWLEDGMENTS ............................................................ xii
ABSTRACT ..................................................................... xiii

CHAPTER 1. INTRODUCTION ...................................................... 1

CHAPTER 2. CODE REPETITIVENESS DETECTION ............................... 5
  2.1 Code and Code Clones Representation ..................................... 6
  2.1.1 Code Fragment and Clone ............................................... 6
  2.2 Structural Similarity Measurement ........................................ 7
  2.2.1 Structure-oriented Representation ..................................... 7
  2.2.2 Structural Feature Selection .......................................... 8
  2.2.3 Characteristic Vectors ................................................. 8
  2.2.4 Vector Computing Algorithm ........................................ 12
  2.3 Code Clone Detection .................................................... 15
  2.3.1 Locality-sensitive Hashing .......................................... 15
  2.3.2 Code Clone Detection Algorithm ................................... 16
  2.3.3 Code Clone Detection Algorithm Complexity ....................... 17
  2.3.4 Clone Reporting ...................................................... 18
  2.4 Empirical Evaluation ................................................... 18
  2.4.1 Correctness .......................................................... 20
  2.4.2 Time Efficiency ..................................................... 21
  2.4.3 Scalability .......................................................... 21
  2.4.4 Clone Consistency Management ................................... 22
  2.4.5 Threats to Validity ............................................... 26
  2.5 Related Work .......................................................... 26
CHAPTER 3. TEMPORAL SPECIFICATION MINING

3.1 Introduction

3.2 Mining Multiple Object Usage Patterns

3.2.1 Formulation

3.2.2 Algorithm Design Strategy

3.2.3 Detailed Algorithm

3.2.4 Anomaly Detection

3.3 Empirical Evaluation

3.3.1 Pattern Mining Evaluation

3.3.2 Anomaly Detection Evaluation

3.4 Related Work

3.5 Conclusions

CHAPTER 4. API PRECONDITION MINING

4.1 Introduction

4.2 Motivating Example

4.3 Mining with Large Code Corpus

4.3.1 Control Dependence and Preconditions

4.3.2 Precondition Normalization

4.3.3 Precondition Inference

4.3.4 Precondition Filtering

4.3.5 Precondition Ranking

4.4 Empirical Evaluation

4.4.1 Data Collection

4.4.2 Ground-truth: Java Modeling Language (JML) Preconditions

4.4.3 RQ1: Accuracy

4.4.4 RQ2: Usefulness

4.4.5 Threats to Validity

4.5 Related Work

4.6 Conclusions
CHAPTER 5. CHANGE REPETITIVENESS ANALYSIS

5.1 Introduction ................................. 67
5.2 Research Question and Methodology .................... 69
  5.2.1 Research Question ......................... 69
  5.2.2 Data Collection ............................ 70
  5.2.3 Experimental Methodology .................. 71
5.3 Code Change Representation .......................... 71
  5.3.1 Illustration Example ........................ 71
  5.3.2 Representation .............................. 72
5.4 Code Change Detection ............................ 74
  5.4.1 Coarse-grained Differencing .................. 74
  5.4.2 Fine-grained Differencing .................... 78
  5.4.3 Collecting Code Changes ..................... 80
5.5 Change Database Building and Change Repetitiveness Computing ............... 81
  5.5.1 Design Strategies ........................... 81
  5.5.2 Detailed Algorithm .......................... 82
5.6 Analysis Results ................................ 82
  5.6.1 Boxplot Representation of General Change and Fix Repetitiveness .......... 82
  5.6.2 Exponential Relationship of Repetitiveness and Size ..................... 84
  5.6.3 Within and Cross-project Repetitiveness Comparison ..................... 85
  5.6.4 Repetitiveness of Bug Fixes ........................ 85
  5.6.5 Repetitiveness on Individual Datasets in SourceForge and GitHub .......... 86
  5.6.6 Repetitiveness on Representative Projects .......................... 88
  5.6.7 Repetitiveness and Change Type .......................... 88
  5.6.8 Threats to Validity .......................... 90
5.7 Related Work .................................. 90
  5.7.1 Large-scale Studies on Uniqueness and Repetitiveness of Source Code .......... 90
  5.7.2 Studies on Code Changes ........................ 90
  5.7.3 Code Clones .................................. 91
  5.7.4 Applications of Repetitiveness of Code Changes .......................... 91
5.8 Conclusions .................................. 92
CHAPTER 6. CODE CHANGE SUGGESTION ........................................ 93

6.1 Introduction ........................................................................ 93
6.2 Problem Formulation .......................................................... 95
   6.2.1 Transactions and Tasks ................................................. 96
   6.2.2 Fine-grained Code Change Representation ...................... 96
   6.2.3 Fine-grained Code Change Extraction .............................. 99
6.3 Modeling Task Context with LDA ........................................ 99
   6.3.1 Key design strategies .................................................. 99
   6.3.2 Details on Modeling Task Context ................................. 101
6.4 Change Suggestion Algorithm ............................................ 102
   6.4.1 Task-based Similarity Measurement for Code Fragments .... 102
   6.4.2 Detailed Algorithm .................................................... 103
6.5 Empirical Evaluation .......................................................... 103
   6.5.1 Data Collection .......................................................... 104
   6.5.2 Evaluation Setup and Metric ....................................... 105
   6.5.3 Parameter Sensitivity Analysis ..................................... 105
   6.5.4 Locality of Task Context ............................................. 107
   6.5.5 Change Suggestion Accuracy Comparison with Base Models .. 108
   6.5.6 Fix Suggestion Using Task Context ............................... 109
   6.5.7 Task Context versus Structural and Co-Change Contexts .... 109
   6.5.8 Case Studies ............................................................. 111
   6.5.9 Threats to Validity ...................................................... 113
6.6 Related Work ................................................................. 113
6.7 Conclusion .......................................................................... 115

CHAPTER 7. GRAPH-BASED API ADAPTATION ......................... 116

7.1 Introduction ........................................................................ 116
7.2 Motivating Examples .......................................................... 119
   7.2.1 Examples of API usage via method invocations ............... 120
   7.2.2 Examples of API usage via inheritance ......................... 121
   7.2.3 Observations ............................................................. 122
7.3 Client API Usage Extractor (CUE) ...................................... 122
   7.3.1 API Usage via Invocation ............................................. 123
### Chapter 7: Adaptation of Behavioral Interface Specifications

#### 7.3.2 API Usage via Inheritance

#### 7.4 Usage Adaptation Miner (SAM)

- **7.4.1 i-Usage Change Detection**
- **7.4.2 x-Usage Change Detection**
- **7.4.3 Usage Adaptation Pattern Mining**

#### 7.5 Recommending Adaptations

- **7.5.1 API i-Usage Adaptation Recommendation**
- **7.5.2 API x-Usage Adaptation Recommendation**

#### 7.6 Evaluation

- **7.6.1 Adaptation of i-Usage**
- **7.6.2 Adaptation of x-Usage**

#### 7.7 Related Work

- **7.7.1 Library Evolution and Client Adaptation**
- **7.7.2 Program Differencing and Origin Analysis**
- **7.7.3 API Usage Specification Extraction**
- **7.7.4 Empirical Studies of API Evolution**

#### 7.8 Conclusions

### Chapter 8: Conclusions and Future Work

#### 8.1 Conclusions

#### 8.2 Future Work

- **8.2.1 Mining Postconditions of Behavioral Interface Specifications**
- **8.2.2 Inferring API Specifications from Similar APIs**
- **8.2.3 Validating Mined Specifications**
- **8.2.4 Mining Bug Fixing Change Patterns**
LIST OF TABLES

Table 2.1 Example of Patterns and Features: numbers are the indexes of nodes and texts are features. ................................................................. 9
Table 2.2 Example of Feature Indexing. Based on the occurrence counts of features in fragment A, the vector for A is (2,1,1,1,2,1,1,1,1,1,1). ...................... 10
Table 2.3 Comparison on Bellon’s Benchmark ........................................ 18
Table 2.4 Comparison on Time Efficiency ............................................. 21
Table 2.5 Time Efficiency and Scalability .............................................. 22
Table 2.6 Clone Change Inconsistency Detection and Synchronization .......... 23
Table 3.1 Pattern Mining Result (σ = 6, δ = 0.1) .................................... 38
Table 3.2 Anomaly Detection and Manual Checking ................................. 39
Table 4.1 Mined Preconditions for String.substring(int,int) .................. 47
Table 4.2 Extracting preconditions for String.substring(int, int) from the usages in Figures 4.1 and 4.2 .................................................. 50
Table 4.3 Collected projects and API usages. ........................................ 54
Table 4.4 Specifications for JDK classes from JML Website ..................... 56
Table 4.5 JML preconditions of JDK methods in classes with full specifications ................................. 56
Table 4.6 Mining accuracy over preconditions ...................................... 57
Table 4.7 Mining accuracy over methods .............................................. 57
Table 4.8 Newly found preconditions in JML specifications ...................... 58
Table 4.9 Different types of incorrectly-mined preconditions ...................... 59
Table 4.10 Four Types of Missing Preconditions ..................................... 59
Table 4.11 Suggesting preconditions ................................................... 62
Table 5.1 Collected Projects and Code Changes .................................... 70
Table 6.1 Collected Projects and Code Changes .................................... 104
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 6.2</td>
<td>Suggestion accuracy comparison between the model using task context and base models.</td>
<td>108</td>
</tr>
<tr>
<td>Table 6.3</td>
<td>Change suggestion accuracy comparison between using task context and using other contexts</td>
<td>112</td>
</tr>
<tr>
<td>Table 6.4</td>
<td>Fix suggestion accuracy comparison between using task context and using other contexts</td>
<td>112</td>
</tr>
<tr>
<td>Table 7.1</td>
<td>Subject Systems</td>
<td>139</td>
</tr>
<tr>
<td>Table 7.2</td>
<td>Precision of API Usage Change Detection</td>
<td>139</td>
</tr>
<tr>
<td>Table 7.3</td>
<td>Accuracy of i-Usage Location Recommendation</td>
<td>141</td>
</tr>
<tr>
<td>Table 7.4</td>
<td>Accuracy of i-Usage Operations Recommendation</td>
<td>142</td>
</tr>
<tr>
<td>Table 7.5</td>
<td>Accuracy of x-Usage Recommendation</td>
<td>143</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2.1 An example of graph-based representation. Numbers are unique identities/indexes of nodes. Texts next to nodes are their labels. ......................... 9
Figure 2.2 Three cases of adding an edge to a fragment. ....................... 12
Figure 2.3 Code Clone Detection Algorithm and Example. ...................... 17
Figure 2.4 Imprecise Clones Reported by Deckard ................................ 20
Figure 2.5 Change Inconsistency in Name in ArgoUML at revision 2453 ....... 24
Figure 2.6 Change Inconsistency in Structure in ArgoUML at revision 2761 .... 24
Figure 2.7 Recommended Clone Synchronization for ZK at revision 9323 ....... 26
Figure 3.1 Pattern and Occurrences .............................................. 31
Figure 3.2 Pattern Mining Algorithm - PattExplorer ............................ 34
Figure 3.3 A Violation Example. $G_1$ and $H_1$ are occurrences of $P_1$. $H_1$ violates $P$ while $G_1$ does not. ............................... 37
Figure 3.4 A Common Usage Pattern Mined from AspectJ ................. 38
Figure 4.1 Client code of API String.substring(int, int) in project SeMoA at revision 1929. ... 46
Figure 4.2 Client code of API String.substring(int, int) in project Jmol at revision 18626. ... 47
Figure 4.3 Approach Overview: We first find all methods that call each API and compute the control-flow graph for each. Then, we generate the control dependence graph to identify conditions leading to an API call. From that, we create an inverted index, then normalize each condition. We infer and merge conditions and then filter some out. The final list is ranked, giving us our result. ................. 48
Figure 4.4 Inferring non-strict inequality preconditions. ....................... 51
Figure 4.5 Merging preconditions with implication. ............................ 52
Figure 4.6 JML Specification for String.substring(int, int) .................... 55
Figure 4.7 Mining accuracy as the size of dataset varies ...................... 60
Figure 4.8 Mining accuracy as technical components are successively added ........... 61
Figure 5.1  An Example of Code Change ................................................................. 72
Figure 5.2  Tree-based Representation of Code Changes ................................. 72
Figure 5.3  Greedy Matching Algorithm ............................................................... 76
Figure 5.4  Tree-based Origin Analysis Algorithm .............................................. 77
Figure 5.5  Initial Mapping and Top-Down Mapping .......................................... 79
Figure 5.6  Extracted Code Changes for Different Heights for the Example in Figure 5.2 . 81
Figure 5.7  Algorithm for Extracting and Computing Repetitiveness ................... 82
Figure 5.8  The Repetitiveness of Code Changes and Fixes over Change Size for all 5,682 Projects in the Corpus ................................................................. 83
Figure 5.9  $R^2$ of Fitted Exponential Curve to Repetitiveness over Change Size .... 84
Figure 5.10  Repetitiveness of Code Changes and Fixes over Change Size for Individual Datasets. (The horizontal and vertical axes are for the sizes of changes and the corresponding median values of repetitiveness over all projects in the dataset, respectively) ...................................................... 86
Figure 5.11  Cross-project Repetitiveness of General Changes and Fixing Changes. (The horizontal and vertical axes are for the sizes of changes and the corresponding repetitiveness values in each project, respectively) ...................................................... 87
Figure 5.12  Within-project Repetitiveness of General Changes and Fixing Changes. (The horizontal and vertical axes are for the sizes of changes and the corresponding repetitiveness values in each project, respectively) ...................................................... 87
Figure 5.13  Repetitiveness of Code Changes and Fixes over Change Type .......... 89
Figure 6.1  An Example of Code Change ................................................................. 97
Figure 6.2  Tree-based Representation for the Code Change in Figure 6.1 ............. 98
Figure 6.3  Extracted Code Changes for the Example in Figure 6.2 ...................... 99
Figure 6.4  LDA-based Task Context Modeling .................................................... 100
Figure 6.5  Change Suggestion Algorithm ............................................................. 103
Figure 6.6  Parameters sensitivity analysis on the impact to the suggestion accuracy in project ONDEX ................................................................. 106
Figure 6.7  Analysis on the locality of task context .............................................. 106
Figure 6.8  Suggestion accuracy comparison between fixing and general changes using task context ................................................................. 109
Figure 6.9  A case study in SWGAide .................................................................. 113
Figure 6.10  A case study in ONDEX ................................. 113

Figure 7.1  API usage adaptation in jBoss caused by the evolution of jFreeChart .......... 120
Figure 7.2  API usage adaptation in jBoss caused by the evolution of OpenNMS .......... 121
Figure 7.3  API usage adaptation in jBoss caused by the evolution of Axis ............... 121
Figure 7.4  API usage adaptation in jBoss caused by the evolution of Axis ............... 122
Figure 7.5  API i-Usage models in jBoss before and after migration to a new jFreechart
                   library version ........................................... 124
Figure 7.6  API x-Usage models in jBoss before and after migration to a new Axis library
                   version ..................................................... 126
Figure 7.7  API Usage Graph Alignment Algorithm ........................................... 128
Figure 7.8  API Usage Changes in JasperReport .............................................. 130
Figure 7.9  Adaptation Pattern Mining Algorithm ............................................ 131
Figure 7.10 API Usage Change Patterns and Reference Models ............................... 133
Figure 7.11 Usage Adaptation Recommending Algorithm .................................... 135
Figure 7.12 Create NumberAxis in jFreeChart ................................................. 140
Figure 7.13 API usage changes in Spring with respect to the evolution of Ruby .......... 141
ACKNOWLEDGMENTS

First and foremost, I would like to thank my family who have always been on my side in this eight-year long journey. Dad, you taught me how to live my life. Mom, you showed me how to care for the ones I love. And brother, I am so proud of you and you are an inspiration for me.

I also would like to thank my friends who have always had faith in me and encouraged me even when I lost faith in myself. I would like to thank my friends in Ames and at Iowa State University. They brought me a new home with more brothers and sisters than I have ever had.

I would like to thank my advisor, Dr. Tien Nguyen. When I first came to this whole new country, he was the first one who welcomed me at the airport. He opened the door to my research career, and has continuously motivated me and nurtured me. He gave me advice on all aspects of conducting research and inspired me to follow research in my career.

I would like to thank Dr. Hridesh Rajan for his support and supervision. He has been like my second advisor for the last three years. His advice as a programming language expert was very valuable to the work in this dissertation.

I would like to thank my teammates and colleagues: Tung Thanh Nguyen, Nam Hoai Pham, Jafar Al-Kofahi, Anh Tuan Nguyen, Hung Viet Nguyen and Robert Dyer. I have learned so much from them through our group discussions, brainstorming and presentations. Their feedback was essential in forming and improving my research ideas. I will never forget the nights without sleeping before the paper deadlines and the moments of receiving notifications together. I am truly grateful for having the opportunity to work and co-author papers with them.

Finally, I would like to thank the committee members for evaluating my work and giving advice on my future research.
ABSTRACT

Software systems inevitably contain a large amount of repeated artifacts at different level of abstraction—from ideas, requirements, designs, algorithms to implementation. This dissertation focuses on analyzing software repetitiveness at implementation code level and leveraging the derived knowledge for easing tasks in software maintenance and evolution such as program comprehension, API use, change understanding, API adaptation and bug fixing. The guiding philosophy of this work is that, in a large corpus, code that conforms to specifications appears more frequently than code that does not, and similar code is changed similarly and similar code could have similar bugs that can be fixed similarly.

We have developed different representations for software artifacts at source code level, and the corresponding algorithms for measuring code similarity and mining repeated code. Our mining techniques bases on the key insight that code that conforms to programming patterns and specifications appears more frequently than code that does not. Thus, correct patterns and specifications can be mined from large code corpus. We also have built program differencing techniques for analyzing changes in software evolution. Our key insight is that similar code is likely changed in similar ways and similar code likely has similar bug(s) which can be fixed similarly. Therefore, learning changes and fixes from the past can help automatically detect and suggest changes/fixes to the repeated code in software development.

Our empirical evaluation shows that our techniques can accurately and efficiently detect repeated code, mine useful programming patterns and API specifications, and recommend changes. It can also detect bugs and suggest fixes, and provide actionable insights to ease maintenance tasks. Specifically, our code clone detection tool detects more meaningful clones than other tools. Our mining tools recover high quality programming patterns and API preconditions. The mined results have been used to successfully detect many bugs violating patterns and specifications in mature open-source systems. The mined API preconditions are shown to help API specification writer identify missing preconditions in already-specified APIs and start building preconditions for the not-yet-specified ones. The tools are scalable which analyze large systems in reasonable times. Our study on repeated changes give useful insights for program auto-repair tools. Our automated change suggestion approach achieves top-1 accuracy of 45%–51% which relatively improves more than 200% over the base approach. For a special type of change suggestion, API adaptation, our tool is highly correct and useful.
CHAPTER 1. INTRODUCTION

Software nowadays is usually not developed from scratch. Programmers tend to reuse artifacts from other components or systems in every step of software development, from ideas, requirements, designs, algorithms to source code. This practice happens in both open-source and closed-source environments. It is due to the fact that multiple software programs could provide common functionalities or share common specifications, designs and/or algorithms, or developers might use the same libraries/frameworks leading to the same code usages or common programming idioms in their source code. This practice has the benefits of reducing development time and cost, shortening length of code and improving the quality of the products since the reused artifacts are usually well-studied, well-designed, well-tested, and compact.

It also introduces repeated software artifacts both within a software project and across multiple software projects. When these artifacts evolves, for supporting new features, improving the performance, fixing bugs, adapting changes in the libraries/frameworks or improving the readability, etc., they might also evolve in the similar ways leading to repeated changes.

This dissertation focuses on the repetitiveness of software artifacts at the source code level. Specifically, this work studies the repetitiveness of code and code changes in a large corpus. That knowledge is leveraged to recover the code specifications and programming patterns, and code change patterns which can be used to prevent bugs, suggest changes and recommend fixes. The key ideas behind this work are: (1) in a large corpus, code that conforms to specifications and programming patterns appears more frequently than code that does not, and (2) similar code is changed similarly and similar code could have similar bugs that can be fixed similarly.

This dissertation is organized into chapters, each of which contains a set of distinct contributions. The material in Chapters 2, 3, 4, 5 and 7 has been peer-reviewed and published in ACM- or IEEE-sponsored conferences and journals. The material in Chapter 6 is under review and has not been published yet.

Chapter 2: Code Repetitiveness Detection. ¹ This chapter presents a syntax-based code clone detection approach which is one component of our clone-aware configuration system—JSync. The core of this component is an efficient technique for extracting structural characteristic features from

¹The material in this chapter has been peer-reviewed and published in our conference papers Nguyen et al. (2009a,b) and journal article Nguyen et al. (2012).
graph-based code artifacts, called Exas. In Exas, a feature is a node with its numbers of incoming and out-going edges or a sequence of nodes along a path (of limited length). Then a graph is approximated by a vector counting all those features in the graph and the graph similarity is approximated via the vector distance. Using Exas vectors not only heuristically reduces the complexity of the graph comparison problem but also enables hashing in mining frequent sub-graphs which drastically improves the running time while still maintains high accuracy and make detecting clones possible in large code corpus.

Syntax-based code clone detection in JSync is one application of Exas where software artifacts have tree representations. Each source file is represented by its abstract syntax tree (AST) and each code fragment is represented as a sub-AST or a forest of consecutive sibling sub-ASTs. JSync first divides all fragments into small buckets by hashing their corresponding Exas vectors. Then, it compares all pairs in each bucket to find clone pairs. Finally, it links clone pairs to build clone groups and reports them.

The combination of vectorizing code fragments using Exas features and hashing makes JSync achieve both correctness and time efficiency in detecting clones. In the evaluation on Bellon’s benchmarks, it outperforms two popular clone detection approaches CCFinderX and Deckard. Our analysis on several open source systems shows that when software evolves, code clones also evolve—clone pairs are changed similarly. However, they are not always changed consistently causing delayed update in the later revisions. JSync raises the awareness of clone relation in software evolution. It detects change inconsistency to clone pairs and suggests synchronization.

Chapter 3 and Chapter 4: Mining Code Specifications. Specifications help developers and automated tools understand intended behavior of software, thus, enable developing high quality and reliable software systems. Unfortunately, specifications are often missing or incomplete. In our study, only 7% of APIs in the Java Development Kit (JDK) have been specified so far. Missing specifications could lead to wrong usages of software which would cause bugs or hinder the use of automated behavior verifiers to detect bugs early. The next two chapters introduce novel approaches for mining two kinds of code specifications. Chapter 3 presents a graph-based approach, GrouMiner, for mining temporal specifications in object-oriented programs called object usage patterns. The temporal specifications specify which and in what order method calls, field accesses and the control structures can interact with each other to complete a task. GrouMiner uses graph to represent programs and uses Exas to efficiently mine the specifications. The idea behind GrouMiner is that frequent usages (or usage patterns) are considered as correct usages, thus, candidates for specifications. GrouMiner found high quality patterns in real-world open source projects which can be useful for program comprehension and documentation.
Chapter 4 presents a consensus-based approach for mining preconditions of APIs. Preconditions are the conditions that must hold before calling an API to guarantee its expected behavior. Preconditions are part of behavioral interface specifications. Our approach mines those conditions by analyzing the guard conditions before the call sites of the API in the client code. Our approach is the first one that analyzes conditions across large number of projects. If a guard condition is checked frequently in majority of the API’s call sites, it will be considered as a correct precondition. Project-specific conditions will be eliminated because they are only checked in one or a small number of projects.

Using mined specifications we developed techniques for detecting buggy code that violates those specifications in mature open source projects. Some bugs had not been identified by the developers. A research group led by Sunghun Kim at The Hong Kong University of Science and Technology used GrouMiner to analyze bug fixing changes and came up with a set of bug fixing templates for object-orient programs.

Chapter 5 and Chapter 6: Code Change Repetitiveness. These two chapters aim to answer two research questions: (1) how repetitive code changes and bug fixes are in software evolution and (2) how useful repeated and previously-seen changes and bug fixes are in suggesting future changes and fixes. Our study is the first one that looks at the change repetitiveness at syntax level and the first one that was carried out in a large-scale code corpus of 5,682 open-source Java projects from two largest hosting services SourceForge and GitHub.

In Chapter 5, we have developed a framework for differencing Java programs between revisions at both coarse-grained (i.e. packages, classes/interfaces and methods) and fine-grained (i.e. statements and expressions within methods) level. Our findings provides useful insights and actionable implications for other research in fix recommendation and automatic program repair.

In Chapter 6, we try several models using repeated and previously-seen changes and fixes to suggest future changes/fixes. This work is the first one applying topic modeling on code changes and using topics to measure code change similarity and suggest changes/fixes. This model significantly improves the suggestion quality over other base models. It achieves accuracy of 45%–51% for top-1 suggestions.

Chapter 7: LibSync–Graph-based API Adaptation. This chapter deals with a special case of software changes in which the code using APIs (client code) needs to be changed accordingly to the change(s) in the code of the APIs’ library. There are many reasons for changes in the libraries such as adding new features, fixing bugs, improving performance, refactoring, etc. The result could be adding new APIs, removing old APIs, changing APIs’ signatures, deprecating APIs, etc., which could lead to

---

3The material in this chapter has been peer-reviewed and published in our conference paper Nguyen et al. (2013b).

4The material in this chapter has been peer-reviewed and published in our conference paper Nguyen et al. (2010a).
changes in the usages of certain APIs. If a user updates the library, his/her existing code could be broken. Sometimes, the code is still compiled but does not behave as expected.

We approach this problem using the same principle stated in the previous chapter: learning from already-seen adaptations and similar usages are adapted similarly. Different from existing approaches, LibSync represents usages as graphs and adaptation as graph transformation operations. This representation enables LibSync to capture not only 1-to-1 adaptations but also many-to-many adaptations. Using graph, LibSync can also include the surrounding code of the usage adaptation as the context of adaptation to improve the quality of recommendations. The evaluation of LibSync on real-world software systems shows that it is highly correct and useful with precision of 100% and recall of 91%.
CHAPTER 2. CODE REPETITIVENESS DETECTION

Recent research has pointed out that software systems inevitably contain a large amount of repeated code, with up to 30% of the total amount of code Kamiya et al. (2002); Kim et al. (2005a), mostly due to the copy-and-paste programming practice, the framework-based development, or design patterns. These repeated code fragments, called code clones, create several difficulties in software maintenance and affect software quality. For example, many bugs occur due to inconsistent modifications made to cloned code Jiang et al. (2007b). These bugs could go unnoticed for a long time, reducing the integrity and quality of the software.

With such a large percentage in the software systems, code clones are of significant concern in software development and maintenance. Conventional approaches assume clones to be harmful, thus, emphasize on the clone detection and removal by unifying and refactoring them. However, recent research results have shown that unifying cloned code might be inefficient and inconvenient Rajapakse and Jarzabek (2007). For example, unifying cloned blocks to a function and replace such cloned code by the function calls could increase running time Aversano et al. (2007). Additionally, in some situations, developers maintain code clones and evolve them into distinct code over time Kim et al. (2005a).

In this chapter, we introduce JSync, a tool for Java code clone detection. JSync models source code in a software system in terms of tree-based, logical entities, such as classes, methods, and statements. A source code file is represented via an abstract syntax tree (AST) and a code fragment is represented as a sub-AST. Two code fragments are considered as code clones if they are sufficiently similar. Their similarity is measured based on the distance of the characteristic vectors extracted from their corresponding tree structures. JSync maintains the clone relations among clone fragments via a clone graph, in which each code clone fragment is represented as a node, and two nodes for two fragments that are clones of each other are connected by an edge. A clone group is a connected component in the clone graph.

We had conducted empirical experiments on Bellon’s clone benchmark Bellon et al. (2007) and several other open-source software systems to evaluate the correctness and performance of our prototype tool. Running on Bellon’s benchmark Bellon et al. (2007), JSync achieves higher recall than CCFinderX CCFinderX and Deckard Jiang et al. (2007a), two popular clone detection tools. The tool is also
scalable and efficient. For example, it is able to detect cloned code in the systems of hundreds thousands LOCs in less than one minute. We used JSync to track clone relation in software evolution. We found that code clones are changed similarly. However, they are not always changed consistently which caused delayed update in later revisions. JSync raises the awareness of clone relation in software evolution. It detects inconsistent changes to clone pairs and suggests synchronization.

Section 2.1 describes the representation of code and clones. Section 2.2 presents our technique for extracting characteristic features that capture the structures of code fragments and measuring the similarity between code fragments. Section 2.3 describes the algorithm for detecting clones. Section 2.4 presents our empirical evaluation. Related work is in Section 2.5. Conclusions appear last.

## 2.1 Code and Code Clones Representation

In JSync, a software system is considered as a collection of source files. Each source file corresponds to a logical entity called *compilation unit*. Each compilation unit is represented as an Abstract Syntax Tree (AST), in which each node represents a *program entity* such as class, method, statement, and expression. The parent-child relation of the nodes represents the containment relation between them. The attributes of those nodes represent the properties of the corresponding entities, such as Name and Type. JSync models a system as a forest of the ASTs of all the source files.

### 2.1.1 Code Fragment and Clone

In JSync, a *fragment* corresponds to one or a collection of program entities (e.g. statement, method, class) that is of user interest in clone management. Since JSync views code as ASTs, it considers a fragment corresponding to either a subtree of an AST or an ordered sequence of the subtrees under the same parent node with the type *block* in the AST.

```java
for (int i = 0; i < n; i++) {
    total.nBytes += a[i].nBytes;
    total.more = a[i].more;
}
```

For example, the above code contains four fragments: two fragments for single assignments inside the for loop (one subtree each), one fragment for the two consecutive assignments (an ordered sequence of two subtrees each), and one fragment for the entire for statement (including its body). JSync does not merge the subtrees from different blocks of code to form a fragment because they belong to different scopes and might not form a meaningful fragment. For example, JSync does not build a fragment which starts with the last statement inside a for loop and ends with the statement right after that for.
Users can define the criteria of the fragments of interest. For example, users are generally not interested in very small cloned fragments. Thus, they can define the fragments to have the sizes larger than a chosen threshold with the size of a fragment is defined as the number(s) of AST nodes in the corresponding subtree(s) of the fragment. Users are able to exclude the generated source files or annotate the portions of code that are generated or boilerplate code (e.g. getter/setter). JSync would totally ignore them in building fragments (i.e. similar handling for comments and Javadoc) or skip building the corresponding fragment(s) but still use the features extracted from them in building other fragments.

Fragments can be copied, pasted, and sometimes modified, thus, producing code clones. Since viewing code as subtrees in ASTs, JSync considers cloned code to have similar structures. It defines a pair of two fragments as a clone pair if their structural similarity, measured by a similarity measurement, exceeds a pre-defined threshold. Those fragments are called cloned fragments (or clones for short).

Since a fragment might be cloned several times, the management and reporting of related clone fragments in groups will be more beneficial than that of individual clone pairs. For example, if a clone is modified, it will be better for a developer to check all other clones in its groups for consistent modifications. To support grouping cloned fragments, and modeling the changes to the clone relation between code fragments, JSync represents the clones and clone pairs in a software system as a clone graph, in which each node represents a cloned fragment and each edge represents a clone pair. Then, JSync considers a connected component of that graph as a clone group. Thus, clone graph and clone groups represent the clone relation among the code fragments.

### 2.2 Structural Similarity Measurement

The similarity between two trees (or graph) is commonly measured by their edit distance. Then, two fragments whose edit distance is small enough (i.e. smaller than a chosen threshold) can be considered as clones. However, calculating tree/graph edit distance for all pairs of code fragments to detect clones is computationally expensive making clone detection not scalable in large systems. This section presents Exas, an efficient structural characteristic feature extraction technique that approximately captures the structures within the graph-based representation of code artifacts.

#### 2.2.1 Structure-oriented Representation

In our structure-oriented representation approach, a software artifact is modeled as a labeled, directed graph (tree is a special case of graph), denoted as \( G = (V, E, L) \). \( V \) is the set of nodes in which a node represents an element within an artifact. \( E \) is the set of edges in which each edge between two nodes
models their relationship. $L$ is a function that maps each node/edge to a label that describes its attributes. For example, for ASTs, node types could be used as nodes’ labels. For program dependence graphs (PDG), program statements and control points could be used for labeling. Other attributes could also be encoded within labels. In existing clone detection approaches, labels for edges are rarely explored. However, for general applicability, Exas supports the labels for both nodes and edges.

The purpose of clone detection is to find cloned parts in software artifacts. Potential cloned parts in a software artifact are called fragments. In our approach, a fragment within a tree-based software artifact is considered as a subtree of the representation tree. For a graph-based software artifact, a fragment is considered as a weakly connected sub-graph in the corresponding representation graph.

Figure 2.1 shows a graph-based representation example containing two clone fragments A and B.

### 2.2.2 Structural Feature Selection

Exas focuses on two kinds of patterns capturing the structure of graphs, called $(p, q)$-node and $n$-path.

A $(p, q)$-node is a node having $p$ incoming and $q$ outgoing edges. Given a node, the values of $p$ and $q$ might be different in different examined fragments. For example, node 9 in Figure 2.1 is a $(3,1)$-node if the entire graph is currently considered as a fragment, but is a $(2,0)$-node if fragment A is examined.

An $n$-path is a directed path of $n$ nodes, i.e. a sequence of $n$ nodes in which any two consecutive nodes are connected by a directed edge in the graph. A 1-path contains only one node.

Structural feature of a $(p, q)$-node is the label of the node along with two numbers $p$ and $q$. For example, node 6 in fragment A is $(2,1)$-node and gives the feature $M-2-1$. Structural feature of an $n$-path is a sequence of labels of nodes and edges in the path. For example, the 3-path 1-5-9 gives the feature $I-G-S$. Table 2.1 lists all patterns and features extracted from A and B. It shows that both fragments have the same feature set and the same number of each feature. Later, we will show that it holds for all isomorphic fragments.

### 2.2.3 Characteristic Vectors

Intuitively, same or similar fragments will have same or similar feature sets, respectively. An efficient way to express the property having the same or similar features is the use of vectors. The characteristic vector of a fragment is the occurrence-count vector of its features. That is, each position in the vector is indexed for a feature and the value at that position is the number of occurrences of that feature in the fragment. Table 2.2 shows the indexes of the features, which are global across all vectors, and their occurrence counts in fragment A.
Figure 2.1: An example of graph-based representation. Numbers are unique identities/indexes of nodes. Texts next to nodes are their labels.

Table 2.1: Example of Patterns and Features: numbers are the indexes of nodes and texts are features.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Features of fragment A</th>
<th>Features of fragment B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-path</td>
<td>I 2 5 6 9</td>
<td>I 4 3 8 7 11</td>
</tr>
<tr>
<td></td>
<td>I G M S</td>
<td>I G M S</td>
</tr>
<tr>
<td>2-path</td>
<td>1-5 1-6 2-6 6-9 5-9</td>
<td>4-8 4-7 3-7 7-11 8-11</td>
</tr>
<tr>
<td></td>
<td>I-G I-M M-S G-S</td>
<td>I-G I-M I-M M-S G-S</td>
</tr>
<tr>
<td>3-path</td>
<td>1-5-9 1-6-9 2-6-9</td>
<td>4-8-11 4-7-11 3-7-11</td>
</tr>
<tr>
<td></td>
<td>I-G-S I-M-S I-M-S</td>
<td>I-G-S I-M-S I-M-S</td>
</tr>
<tr>
<td>(p,q)-node</td>
<td>1 2 5</td>
<td>4 3 8</td>
</tr>
<tr>
<td></td>
<td>I-0-2 I-0-1 G-1-1</td>
<td>I-0-2 I-0-1 G-1-1</td>
</tr>
<tr>
<td>(p,q)-node</td>
<td>(continued)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 9 7 11</td>
<td>M-2-1 S-2-0</td>
</tr>
<tr>
<td></td>
<td>M-2-1 S-2-0</td>
<td></td>
</tr>
</tbody>
</table>

Two fragments having the same feature sets and occurrence counts will have the same vectors and vice versa. The vector similarity can be measured via a chosen vector distance such as 1-norm distance.

\[ sim(u, v) = 1 - \frac{\|u - v\|_1}{(\|u\|_1 + \|v\|_1)/2} \]

The normalization (division) is used to tolerate the differences between fragments’ sizes.

Note that 1-paths are equivalent to 1-level binary subtrees used in Deckard Jiang et al. (2007a). Therefore, a Deckard vector of a fragment is a part of the Exas vector for that fragment. For example, Deckard vector for fragment A would be (2,1,1,1). In other words, Exas uses more features. It implies that Exas vector distance has better discriminative characteristic, i.e. is more accurate in measuring the fragments’ similarity. These are also true when applying for tree structures of source code ASTs. For example, Exas can better approximate the nesting and sequential structures of program elements.

### 2.2.3.1 Analytical Study

Given two (sub)graphs $G$ and $G'$. Let $V$ and $V'$ be their vectors, respectively, $d$ be the maximum degree of all nodes of all (sub)graphs, and $N$ be the maximum size of all $n$-paths. It is easy to verify the following Lemma 2.1.
Table 2.2: Example of Feature Indexing. Based on the occurrence counts of features in fragment A, the vector for A is (2,1,1,1,2,1,1,1,2,1,1,1,1,1).

<table>
<thead>
<tr>
<th>Feature</th>
<th>Index</th>
<th>Counts</th>
<th>Feature</th>
<th>Index</th>
<th>Counts</th>
<th>Feature</th>
<th>Index</th>
<th>Counts</th>
<th>Feature</th>
<th>Index</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
<td>2</td>
<td>I-G</td>
<td>5</td>
<td>1</td>
<td>I-G-S</td>
<td>9</td>
<td>1</td>
<td>G-I-1</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td>1</td>
<td>I-M</td>
<td>6</td>
<td>2</td>
<td>I-M-S</td>
<td>10</td>
<td>2</td>
<td>M-2-1</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>M</td>
<td>3</td>
<td>1</td>
<td>G-S</td>
<td>7</td>
<td>1</td>
<td>I-0-1</td>
<td>11</td>
<td>1</td>
<td>S-2-0</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td>4</td>
<td>1</td>
<td>M-S</td>
<td>8</td>
<td>1</td>
<td>I-0-2</td>
<td>12</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lemma 2.1 The number of \( n \)-paths containing a node is at most \( P = \sum_{n=1}^{N} n.d^{n-1} \) and that of \( n \)-paths containing an edge is at most \( Q = \sum_{n=2}^{N} n.d^{n-2} \).

For brevity, let us call \( n \)-paths and \((p,q)\)-nodes instances. Let \( S \) and \( S' \) be the sets of instances of \( G \) and \( G' \), respectively. If \( G \) is edited to be \( G' \), \( S' \) is updated accordingly from \( S \) by removing and/or inserting some instances. Let us call those removed and inserted instances as “affected instances”.

**Lemma 2.2** \( k \) graph editing operations affect at most \((2P + 4)k\) instances.

**Proof.** We consider four types of graph editing operations: removing an edge, inserting an edge, relabeling a node, and relabeling an edge. Removing (inserting) an edge removes (inserts) all \( n \)-paths containing it and replaces two \((p,q)\)-nodes at its two ends with two new \((p,q)\)-nodes. This replacement affects four instances. Thus, the total number of affected instances is at most \( Q + 4 \). Relabeling a node replaces its corresponding \((p,q)\)-node and all \( n \)-paths containing it with new instances, thus, affects at most \( 2 + 2P \) instances. Similarly, relabeling an edge affects at most \( 2Q \) instances since no \((p,q)\)-node is affected. In all cases, the total number of affected instances is at most \( 2P + 4 \). Therefore, \( k \) editing operations affect at most \((2P + 4)k\) instances.

**Lemma 2.3** If there are \( M \) affected instances, \( \|V - V'\|_1 \leq M \).

**Proof:** If an instance is removed (inserted), the occurrence counts of its feature reduce(increase) by one. Since there are \( M \) affected instances, \( V' \) is obtained from \( V \) by the total of \( M \) units of such increment and/or decrement. Since \( \|V - V'\|_1 \) is the total differences of occurrence counts between \( V \) and \( V' \), it is at most \( M \).

Lemmas 2.2 and 2.3 imply the following theorem.

**Theorem 2.4** If graph edit distance of \( G \) and \( G' \) is \( k \), \( \|V - V'\|_1 \leq (2P + 4)k \).

We could consider two isomorphic graphs as having the editing distance of zero. Therefore, applying Theorem 2.4, we have the following corollary.
Corollary 2.5. If $G$ and $G'$ are isomorphic, they have the same vector, i.e., $V = V'$.

The results can be applied directly to (sub)trees. However, tree editing distance can be defined in a different set of operations. The following results are for the case in which $G$ and $G'$ are (sub)trees and tree editing operations include relabeling, inserting, and deleting a node.

Lemma 2.6 The number of $n$-paths containing a node in a (sub)tree is at most $R = \sum_{n=1}^{N} \sum_{i=1}^{n} d^{i-1}$.

Lemma 2.6 can be verified similarly to Lemma 2.1 with a note that a node in a tree has exactly one incoming edge except for the root node which has no incoming edges.

Lemma 2.7 $k$ tree editing operations affect at most $(2R + 3)k$ instances.

Proof: Relabeling a node affects at most $2R + 2$ instances (see the proof of Lemma 2.2). Removing a node $u$, i.e. connecting all its children to its parent $v$, removes all $n$-paths containing $u$, inserts some $n$-paths containing $v$, replaces $(p, q)$-node at $v$ with a new one, and removes $(p, q)$-node at $u$. Thus, the total number of affected instances is $2R + 3$. Similar argument is applied for the case of inserting a node $u$. Therefore, a single tree editing operation affects at most $(2R + 3)$ instances, thus, $k$ operations affect at most $(2R + 3)k$ instances.

From Lemmas 2.6 and 2.7, we have the following theorem.

Theorem 2.8. If tree editing distance of $G$ and $G'$ is $k$, $\|V - V'\|_1 \leq (2R + 3)k$.

2.2.3.2 Implications in Clone Detection

The aforementioned important properties of Exas characteristic vectors imply that they are very useful in the problems involving graph isomorphism or tree/graph similarity, especially in structure-oriented clone detection.

State-of-the-art graph-based clone detection approaches Deissenboeck et al. (2008); Komondoor and Horwitz (2001) require graph-based cloned fragments to be isomorphic. With Corollary 2.5, instead of checking isomorphism of two (sub)graphs, we could compare their Exas characteristic vectors to find cloned (sub)graphs. That corollary guarantees that all clone pairs will be detected. However, it is not a sufficient condition for absolute precision, i.e. two (sub)graphs with the same vectors might be non-isomorphic since nodes and edges which cannot be mapped between two (sub)graphs can make up $n$-paths or $(p, q)$-nodes with the same feature. Other criteria should be used along with Exas to increase the precision of detected results.
Theorem 2.4 shows that our approach is also useful for problems involving graph editing distance such as similarly-matched clone detection in graph-based representation or graph similarity measurement.

Theorem 2.8 is useful for clone detection approaches based on tree editing distance, i.e. two tree-based fragments are considered clones if their editing distance is smaller than a chosen threshold $k$. For a set of fragments, we can always find a common value $R$ for any two fragments. Then, the vector distance of any two cloned fragments will be less than $(2R + 3)k$. In other words, the 1-norm distance of Exas characteristic vectors could be used as a necessary condition: to be a cloned pair, two fragments must have the distance of their vectors smaller than a chosen threshold $\delta = (2R + 3)k$. Of course, a small vector distance does not imply a small tree editing distance, i.e. this condition is not sufficient.

### 2.2.4 Vector Computing Algorithm

In this section, we describe an efficient algorithm to calculate Exas vectors from the structure-oriented representation. The key idea is that the characteristic vector of a fragment is calculated from the vectors of its sub-fragments. A node is the smallest fragment and its vector is calculated directly.

#### 2.2.4.1 Key Computation Operation: `incrVector`

The key operation in our algorithm, `incrVector`, is the computation of the vector for a fragment $g = f + e$ (i.e. $g$ is built from $f$ by extending $f$ with an edge $e$), given $e$ and $f$ (along with its vector) as inputs. In brief, the vector of $g$ is derived from that of $f$ by updating it with the occurrences of all new features of $g$ created by the addition of the edge $e$ into $f$.

Since we consider only weakly connected components as fragments, at least a node of $e$ must belong to $f$. Let $e = (u, v)$. There are three following cases:

**Case 1: incoming-edge** (Figure 2.2a), i.e. $u \notin f$ and $v \in f$. In this case, $u$ is a newly added node. New features are created from the 1-path $u$, the 2-path $u - v$, the new $(0, 1)$-node at $u$. The $(x,y)$-node at $v$ is replaced by the new $(x + 1, y)$-node because of the new incoming edge. All new $n$-paths of $f$ will
have the first node of $u$ and the second node of $v$. Therefore, they are generated by adding $u$ to the first of all $(n-1)$-paths starting from $v$. These $(n-1)$-paths can be achieved by a depth-first search (DFS) within fragment $g$ from node $v$ to the depth of $n-2$.

Case 2: outgoing-edge (Figure 2.2b), i.e. $u \in f$ and $v \notin f$. The situation is similar. However, new $n$-paths are generated from $(n-1)$-path ending at $u$, i.e. DFS needs to expand in backward direction. Furthermore, $(x,y)$-node at $v$ is replaced by a new $(x,y+1)$-node.

Case 3: connecting-edge (Figure 2.2c), i.e. both $u$ and $v$ were already in $f$. In this case, new $n$-paths are generated by the combination of any $i$-path ending at $u$ (DFS in backward direction) and an $j$-path starting from $v$ (DFS in forward direction), for all $i + j = n$. Both $(x,y)$-nodes at $u$ and $v$ are replaced by new $(x',y')$-nodes.

2.2.4.2 Time Complexity and Improvement

Assume that $d$ is the maximum degree of the nodes and $N$ is the maximum length of $n$-paths of interest. The number of $n$-paths searched by DFS is $O(d^{N-2})$ in all three cases (in the 3rd case, two DFSs from $u$ and $v$ to level $n-2$ are sufficient to find all those $x$-paths and $y$-paths). This seems to be exponential. However, instead of extracting features from $n$-paths of all sizes, we just extract features from short $n$-paths, i.e. $n$-paths having at most $N$ nodes. This gains much efficiency and reduces little precision. In our experiments in Nguyen et al. (2009a), in most subject systems, $N = 4$ gives the precision of almost 100%. Moreover, in practice, representation graphs are generally not very dense, i.e. $d$ is small. Thus, $O(N.d^{N-2})$ is indeed not very time-consuming.

2.2.4.3 Vector Computation for All Fragments in a Graph

Using incrVector operation, Exas calculates the vector of any individual fragment by starting from one of its nodes, adding one of its edges, then computing the vector, and so on. Thus, time complexity of computing vector for a fragment is $O(m.N.d^{N-2})$, with $m$ as the fragment’s size, i.e. the number of edges.

For the clone detection problem, the goal is to calculate the vectors for all potential cloned fragments in a graph. Generating all its sub-graphs and then calculating their vectors as for individual fragments will not take advantage of incrVector operation. A more efficient approach is to generate the fragments with the increase in size by adding edge-by-edge and then to calculate the vector for the larger fragment from the vectors of the smaller ones.

However, the number of sub-graphs of a graph is exponential to its size. To increase efficiency, if graph isomorphism is used as a clone condition, we can take advantage of the following fact: to be a
clone, a fragment should contain a smaller cloned fragment, i.e. two large isomorphic graphs should contain two smaller isomorphic sub-graphs.

Let $C_k$ be the set of all cloned fragments of size $k$ (i.e. with $k$ edges). Observe that: (1) every fragment of size $k$ can be generated from a fragment of size $k - 1$ by adding a relevant edge; and (2) if two fragments of size $k$ are a clone pair (isomorphic), there exists two cloned fragments of size $k - 1$ within them, i.e. every clone pair of $C_k$ can be generated from a clone pair of $C_{k-1}$.

Those facts imply that $C_k$ can be generated from $C_{k-1}$ by following steps: (1) extending all cloned fragments in $C_{k-1}$ by one edge to have a candidate set $D_k$, (2) calculating vectors for all fragments in $D_k$ by the incrVector operation, (3) grouping $D_k$ into clone groups by characteristic vectors (i.e. all fragments in a group must have the same characteristic vectors), and (4) adding only the cloned fragments in $D_k$ into $C_k$. By gradually generating $C_0, C_1, C_2, ..., C_k, ...$, we can find all cloned fragments precisely and completely. Note that, this strategy reduces significantly time complexity for fragment generation and vector computation for sparse graphs. In worst case (such as for a complete graph), time complexity is still exponential. More details can be found in Pham et al. (2009).

### 2.2.4.4 Vector Calculation for All Fragments in a Tree

For tree-based representations, a fragment is represented by a subtree. Since each node is the root of a subtree, i.e. each fragment corresponds to a node, the generation process is not needed. To compute the vectors for all subtrees in a tree, Exas traverses it in post-order. When a root $p$ of a subtree $T(p)$ is traversed, the vector for $T(p)$ will be calculated as follows. Assume that $c_1, c_2, ..., c_k$ are the children of $p$, connecting from $p$ by edges $e_1, e_2, ..., e_k$. Because of the post-order traversal, the vectors of the sub-trees $T(c_1), T(c_2), ..., T(c_k)$ have been already calculated. Adding edge $e_i$ to subtree $T(c_i)$ using incrVector operation gives the vector $V_i$ for each branch. Then, the vector of $T(p)$ is derived from all vectors $V_1, V_2, ..., V_k$ and the $(p,q)$-node at $p$. By this strategy, time for computing vectors for all fragments of a tree is just $O(m.N.d^{N-2})$.

### 2.2.4.5 Vector Indexing and Storing

The potential number of features is huge. For example, if the number of different labels of nodes is $L_v$ and that of edges is $L_e$, the total number of potential features generated from all $n$-paths of size no longer than $N$ is $\sum_{n=1}^{N} L_v^n L_e^{n-1}$. However, in practice, the actual features encountering in certain graph modeling for a software artifact is much smaller because there are not all combinations of nodes and edges that make sense with respect to the semantics of elements in artifacts.
Our experiment confirmed this fact. We conducted an experiment with WCD-MALIB, a real-world model-based system with 388 nodes and $L_v = 107$ labels ($L_e = 0$). With $N = 4$, the actual features encountered in the whole model is only 381, although the number of all possible features is more than $10^7$.

More importantly, most fragments do not contain all features, especially small fragments. Thus, the characteristic vectors are often sparse. To efficiently process sparse characteristic vectors, a hashmap $H$ is used to map between features and their indexes (i.e. their positions in vectors for storing their occurrence counts). $H$ is global and used for all vectors. During vector calculation, if a feature has never been encountered before, it will be added into $H$ with a next (increasing) available index. The vector of a fragment is also stored as a hashmap that maps the index of each feature into its corresponding counting value in the vector.

### 2.3 Code Clone Detection

To find the clone pairs and construct the clone graph, one could perform pairwise comparison on all fragments in the software system. Then, those pairs are used to form the clone graph and its clone groups. However, because a system usually has a large number of fragments, pairwise comparison is not efficient. Thus, JSync compares only the fragments that are likely to be clones, i.e having similar vectors. To find those fragments, it uses locality-sensitive hash functions Andoni and PiotrIndyk.

#### 2.3.1 Locality-sensitive Hashing

A locality-sensitive hashing (LSH) function is a hash function for vectors such that the probability that two vectors having the same hashcode is a strictly decreasing function of their corresponding distance Andoni and PiotrIndyk. In other words, vectors having a smaller distance will have a higher probability $p$ of having the same hashcode, and vice versa. Then, if we use locality-sensitive hash functions to hash the fragments into buckets based on the hashcodes of their vectors, fragments having similar vectors tend to be hashed into the same buckets, and the other ones are less likely to be so.

To increase the probability that two similar vectors $u$ and $v$ be hashed to the same buckets, JSync uses $N$ independent hash functions, and hashes each vector to $N$ corresponding buckets. Then, if two vectors are missed by a hash function, they still have chances to be so from the others. Indeed, the probability that $u$ and $v$ are missed by all $N$ functions, i.e. having all different hashcodes is $(1 - p)^N$. If $N$ is sufficiently large, this probability approaches to zero. That is, there is a high probability that $u$ and $v$ will be hashed into the same bucket.
2.3.2 Code Clone Detection Algorithm

The pseudo code of the detection algorithm is presented in Figure 2.3a. B and G are two maps to store the buckets and the clone graph. B[i] denotes the bucket corresponding to the hashcode i. G(u) denotes the clone group that contains fragment u. The clone graph is stored as adjacent lists: G(u) contains the adjacent nodes of u, i.e. the clones of u, and u itself.

First, each fragment u in the software system is hashed into N buckets indexed by its hash codes produced from N independent hash functions (lines 2-4). (Note that hash codes produced from different hash functions are distinguished by encoding the index of each function as a part of the hash code). Then, the fragments of each bucket are compared pairwise to detect the clone pairs (lines 5-8). Since a fragment can be enclosed in another one, a clone pair might have both fragments being enclosed by two fragments of some other pair, thus becomes redundant. To filter those redundant pairs (u, v) (lines 11-16), JSync checks if any fragment enclosing u is cloned with a fragment containing v. If this is the case, (u, v) will be removed from the clone relation. The most costly part of this task is to find the set E(u) of all fragments enclosing u. JSync handles this efficiently based on two observations. First, a fragment containing u must be in the same source file with u. Second, due to the bottom-up process in generating the fragments, i.e. building the larger one from the small ones, the fragment enclosing u must be either u or be generated after u, hence its index must not be less than u’s. Thus, JSync searches for E(u) only within the fragments from u to the end of the list of all fragments in its file. After filtering redundant clone pairs, JSync collects all fragments in each connected component of the clone graph G into a clone group (lines 17-20).

Figure 2.3b shows an illustrated example for this algorithm. The left part of Figure 2.3b shows the fragment set F containing six fragments from a to f which form two clone groups \{a, b\} and \{d, e, f\}. They are hashed by two hash functions h1 and h2 into the buckets B[i], which are displayed in the middle of Figure 2.3b (Buckets of h1 have odd indexes and those of h2 have even indexes). Then, by pairwise comparing the fragments in each bucket, the clone pairs are detected and added to the clone graph represented in the right part of Figure 2.3b. For example, the clone pair (a, b) is hashed to and detected from buckets B[1] and B[2], (d, e) is from B[3], and (d, f) is from B[4].

After all clone pairs are detected and filtered, clones in each connected component are unioned to form a clone group. For example, (d, e) and (d, f) form the clone group of \{d, e, f\}.

It should be noted that, due to the nature of locality-sensitive hashing, non-cloned fragments such as c and f might be hashed to the same bucket (B[5]) while cloned fragments such as e and f might not be hashed to the same bucket. Thus, such pair might not be represented as a clone pair in the clone graph.
If more hash functions are used, there is higher probability that they will be hashed to the same bucket and be detected. In this example, the clone pair \((e,f)\) is still detected in the clone group \(\{d,e,f\}\) due to the connectivity of \((d,e)\) and \((d,f)\). This result implies a benefit of considering a connected component in the clone graph as a clone group. That is, some cloned fragments might not be detected from the buckets, but are still detected by their clone relation to other fragments.

### 2.3.3 Code Clone Detection Algorithm Complexity

The first part of our clone detection algorithm at lines 2-8 has the complexity of \(O(nNq)\), where \(n\) is the number of fragments, \(N\) is the number of hash functions, and \(q\) is the maximal size of a bucket. In practice, especially in large systems, with \(q\) is usually very small in comparison to \(n\). Due to the nature of locality-sensitive hashing, a bucket generally contains only the vectors of very similar fragments. Thus, this part could be considered as linear to \(n\). The function \texttt{Filter} basically checks each clone pair against all other pairs, so its complexity is \(O(p^2)\), where \(p\) is the number of clone pairs. The complexity of the function \texttt{BuildCloneGroup} is subsumed by that of function \texttt{Filter}. Overall, our clone detection has the complexity of \(O(nNq + p^2)\).
Table 2.3: Comparison on Bellon’s Benchmark

<table>
<thead>
<tr>
<th>System</th>
<th>Files</th>
<th>KLOC</th>
<th>Reported (KLOC)</th>
<th>Filtered</th>
<th>Matched pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>JS 0.9</td>
<td>JS 0.95</td>
<td>JS 1.0</td>
</tr>
<tr>
<td>netbean-javadoc</td>
<td>101</td>
<td>19</td>
<td>3.59</td>
<td>3.32</td>
<td>2.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.80</td>
<td>1.7</td>
<td>1.34</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8.36</td>
<td>1.37</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.32</td>
<td>3.73</td>
<td>2.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.32</td>
<td>3.73</td>
<td>2.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.32</td>
<td>3.73</td>
<td>2.94</td>
</tr>
<tr>
<td>eclipshell-ant</td>
<td>178</td>
<td>35</td>
<td>2.33</td>
<td>1.65</td>
<td>1.34</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2.82</td>
<td>8.36</td>
<td>1.37</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.34</td>
<td>3.73</td>
<td>2.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.34</td>
<td>3.73</td>
<td>2.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.34</td>
<td>3.73</td>
<td>2.94</td>
</tr>
<tr>
<td>eclipshell-jdtcore</td>
<td>741</td>
<td>148</td>
<td>37.92</td>
<td>32.57</td>
<td>28.56</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40.50</td>
<td>62.04</td>
<td>24.57</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>36.83</td>
<td>1345</td>
<td>468</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1345</td>
<td>468</td>
<td>451</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>451</td>
<td>423</td>
<td>450</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>423</td>
<td>450</td>
<td>450</td>
</tr>
<tr>
<td>j2sdk-javax-swing</td>
<td>538</td>
<td>204</td>
<td>27.35</td>
<td>21.54</td>
<td>19.48</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32.91</td>
<td>80.93</td>
<td>23.73</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>37.12</td>
<td>777</td>
<td>427</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>37.12</td>
<td>777</td>
<td>427</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>37.12</td>
<td>777</td>
<td>427</td>
</tr>
</tbody>
</table>

2.3.4 Clone Reporting

A clone might be a part of another clone. For example, if a class is cloned from another class, its
methods will also be clones of the methods of that class. Therefore, any clone group whose all members
belong to the members of another clone group will be considered redundant, and will not be reported.

2.4 Empirical Evaluation

Let us present our empirical evaluation on JSync’s clone detection accuracy and efficiency. All
experiments were run on a computer with both Windows Server 2008 and Linux OS, and with AMD
Phenom II X4 3.41GHz CPU and 12GB RAM. JSync was configured to process code fragments with
the minimum size of 50 nodes, 16 independent hash functions, and the similarity threshold \( \sigma \) of 0.95.

We evaluated the performance of JSync on clone detection in comparison with Deckard Jiang et al.
(2007a), a state-of-the-art tree-based clone detector and with CCFinder Kamiya et al. (2002), a popular
token-based clone detection tool. CCFinder was run with its default running setting. Deckard were ran
with the similarity threshold of 0.95, which is also the default value of Deckard and the one that gave
higher recall than the compared tool as reported in their paper Jiang et al. (2007a). JSync was run
with three different threshold values of 0.9, 0.95 and 1.0.

We used Bellon’s benchmark Bellon et al. (2007) for our evaluation. This benchmark contains manually
verified clone pairs, called reference pairs, from four Java projects and four C projects. As reported in
Bellon et al. (2007), those pairs have been manually verified from 2 percent of all 325,935 candidate
clone pairs submitted by several well-established clone detection tools including CCFinder CCFinderX,
CPMiner Li et al. (2006), CloneDR Baxter et al. (1998), with several clone detection approaches ranging
from text-, token-, tree-, graph- to metric-based techniques. Because JSync focuses on Java code, we
took only the four Java projects in the benchmark as the subject systems. Because the benchmark con-
tains only 2 percent of all submitted clone pairs, (which already took the author 77 hours to manually
verify,) we used it for only comparing the recall between tools but not measuring the absolute recall of
each individual tool.
Detailed information of four subject systems is described in Table 2.3. The largest subject is Swing, a GUI API of Java SDK 2, with around 200KLOCs and 777 reference clone pairs in the benchmark. Eclipse JDTcore has 1,345 pairs in the benchmark, although its size is a bit smaller, at around 150KLOCs. In total, four subject systems contain around 400KLOCs and more than 2,200 reference clone pairs.

We ran three tools in those subject systems. The total cloned LOCs reported by each tool are shown in Table 2.3. As seen, for Swing (204KLOCs), JSync (JS) and CCFinder (CF) report around 20-30KLOCs, while Deckard (DK) reports 81KLOCs of clones. Similarly for jdt-core, JSync and CCFinder report around 30-40KLOCs, while Deckard reports more than 60KLOCs of clones. In order to understand those large differences, we analyzed a small random sample of 100 clone pairs reported from Deckard and discovered several imprecise clone pairs. Here are some typical patterns of those clones:

W1. Many reported clones from Deckard contain all or mostly import statements. Even though having similar structures, those fragments are of little interest for clone management (e.g. many program editors generate such code automatically). Deckard reports a large amount of such “import” code.

W2. Deckard reports many clones that are actually getters/setters of individual fields. Similar to W1, those fragments are often produced by modern program editors and do not need clone management. In many cases, they are identical in structure due to simple logic such as assignment and return statements. However, they have the accesses to two unrelated fields, thus, have no semantic relation, and would not require any clone consistency management. Thus, we also consider them as imprecise clones.

W3. There are some cases in which a cloned fragment reported by Deckard consists of a part of a method and another part of the next method, however, it does not contain the entirety of any method. Figure 2.4 shows an example of such clone pairs reported by Deckard. As seen, the return statement of the first fragment belongs to a method and the remaining part belongs to a different method. In addition, these fragments are unlikely to be copy-and-pasted, thus, unlikely to be clones.

We had reported this result to the corresponding author of Deckard. The author confirmed this imprecision and explained the existence of such clones due to the following reasons. First, Deckard represents source code via parse trees and has no semantic information of the fragments corresponding to each sub-tree or sub-forest of a parse tree. Therefore, it considers the fragments containing import statements or getters/setters as clones, because they have similar structures as represented in the parse tree. Second, it uses a token-based sliding window to merge small cloned fragments into a larger one. Thus, it could accidently merge the fragments of two different methods.

CCFinder, due to its token-based approach, could not recognize the methods’ boundaries and the getters/setters. Therefore, like Deckard, it also reports code clones as described in W2 and W3. However, it recognizes import statements, thus, does not report them.
Since JSync uses more sophisticated source code analysis, it does not report such kinds of imprecise clones. JSync represents source code as ASTs, thus, it can recognize import, package and field declaration statements, and getters/setters. Currently, JSync disregards such kind of code and considers the fragments inside methods’ boundaries. In addition, JSync builds a larger fragment from the smaller ones via merging AST’s sub-trees. Thus, the fragments will always be inside methods’ boundaries or contain well-formed code structures.

To be more precise in our evaluation, we built a post-processing tool to filter those three kinds of imprecise clones from the reported results of CCFinder and Deckard. As seen in Column Filtered of Table 2.3, after such processing, for Swing, Deckard detected 37KLOCs of clones, while JSync detected 22KLOCs and CCFinder detected 24KLOCs.

To evaluate the correctness and efficiency of JSync in comparison with the state-of-the-art tools, we checked their reported clone pairs against the Bellon’s benchmark. We use the good-value matching criterion as described in Bellon et al. (2007): a reported pair $v - v'$ is considered to match a reference pair $u - u'$ in the benchmark if their overlapping level in terms of lines of code exceeds 70%.

### 2.4.1 Correctness

Column Matched Pairs in Table 2.3 shows the matching result, i.e. the number of the reported clone pairs of each tool that match the reference clone pairs in the benchmark. Sub-columns JS 0.9, JS 0.95 and JS 1.0 show the corresponding numbers of matched pairs for JSync with different similarity thresholds, respectively. As seen, CCFinder has the lowest number of matches. JSync has the highest, except the result with threshold of 1.0 on jdtcore, while Deckard has slightly smaller numbers of matches than JSync. For example, on Swing with the largest number of reference clone pairs (1,345), CCFinder
matches only 264, while JSync matches 468, 451, and 423 for thresholds of 0.9, 0.95, and 1.0, respectively, and Deckard matches 450. Both JSync and Deckard are tree-based approaches that use vector-based similarity to detect clones. However, Deckard counts only AST’s node types as features, while JSync uses additional features, e.g. vertical and horizontal sequences extracted from the paths and sibling nodes in an AST (Section 3.2.1). Thus, at the same level of similarity, Deckard is simpler on its clone criteria, thus, would consider more fragments as cloned code, i.e. it reports more imprecise clones. That is the reason why although reporting more clones, Deckard has slightly smaller numbers of matches to reference pairs in the benchmark than JSync.

2.4.2 Time Efficiency

Table 2.4 shows the running time of three tools. Since CCFinder runs on Windows while Deckard runs on Linux, we ran JSync in both environments for comparison. As seen, JSync is more time-efficient than both CCFinder on Windows and Deckard on Linux. For example, for jdt-core (140KLOCs) and Swing (200KLOCs), JSync took about 1-2 minutes while CCFinder took more than 2 minutes on Windows. For those two systems, it took about 20 minutes for Deckard and 3-4 minutes for JSync on Linux.

2.4.3 Scalability

To evaluate the scalability of JSync, we ran JSync on several subject systems. As shown in Table 2.5, for medium systems of some hundreds thousands LOCs, most of the time it took JSync only less than one minute, except for ZK where the running time was almost three minutes. For the large systems of more than 4 MLOCs (Eclipse and JDK 6), it took JSync less than half an hour. From this table, we also see that the running time is sensitive to the number of clone pairs. That is why ZK took more time than its similar medium-sized systems and so did JDK 6 and Eclipse.

We attribute JSync’s high level of efficiency to the following design and implementation features. First, it has an incremental extraction process of code fragments and characteristic vectors, which computes the vector of a fragment from those of its sub-fragments. Second, JSync uses the vector distance for code similarity instead of the tree editing distance, thus, reduces the code similarity measurement.
Table 2.5: Time Efficiency and Scalability

<table>
<thead>
<tr>
<th>System</th>
<th>GEclipse</th>
<th>jEdit</th>
<th>ZK</th>
<th>Columbia</th>
<th>ArgoUML</th>
<th>Eclipse</th>
<th>JDK 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>157K</td>
<td>176K</td>
<td>182K</td>
<td>193K</td>
<td>474K</td>
<td>4,119K</td>
<td>4,288K</td>
</tr>
<tr>
<td>Fragments</td>
<td>43K</td>
<td>60K</td>
<td>32K</td>
<td>66K</td>
<td>73K</td>
<td>1,066K</td>
<td>672K</td>
</tr>
<tr>
<td>Clones</td>
<td>12K</td>
<td>4K</td>
<td>4K</td>
<td>8K</td>
<td>10K</td>
<td>129K</td>
<td>74K</td>
</tr>
<tr>
<td>Pairs</td>
<td>35K</td>
<td>7K</td>
<td>72K</td>
<td>12K</td>
<td>26K</td>
<td>520K</td>
<td>1,077K</td>
</tr>
<tr>
<td>Time (mm:ss)</td>
<td>0:40</td>
<td>0:31</td>
<td>2:50</td>
<td>0:51</td>
<td>0:50</td>
<td>10:56</td>
<td>27:54</td>
</tr>
</tbody>
</table>

computation by an order of magnitude. And importantly, with the use of locality sensitive hashing, JSync does not require the pair-wise comparison on all code fragments. Since cloned code fragments are mostly hashed to the same buckets, and a bucket tends to contain a small number of fragments, the number of comparisons are mostly proportional to the number of clone pairs in the system.

Although both Deckard and JSync rely on locality-sensitive hashing, JSync is more efficient due to its specialized implementation of hashing and other operations for sparse vectors. That is, the characteristic vectors of fragments generally contain only a small number of characteristic features, since a piece of code rarely contains all syntactic structures. Taking advantage of this, we implemented our vector operations (e.g. addition, similarity, locality sensitive hashing) based on a representation for sparse vectors, which further improved JSync’s performance. For example, if the total number of features is 1,000, but a fragment has only 100 features, the calculation of its hashcode would take only 100 operations on those features, i.e. 10 times faster than doing hashing on a full vector representation for all 1,000 features.

In brief, this experiment shows that, JSync can scale to large systems, and is more efficient than CCFinder and Deckard, and detects more correct clones of interest than both of them.

2.4.4 Clone Consistency Management

This experiment shows our analysis on the existence of inconsistent changes to clone pairs in software evolution and the usefulness of JSync in clone change inconsistency detection and synchronization.  

2.4.4.1 Clone Change Consistency Analysis

In this experiment, we performed a clone consistency analysis using JSync for a set of subject systems listed in Table 2.6. Since we were concerned with the clone changes related to bugs, we focused on analyzing only the bug fixing changes. We had collected this data from several open-source repositories through a semi-automatic process. First, all revisions having commit logs containing keywords of bug, fix and error were automatically extracted. Then, the authors manually examined to decide which ones

1Details on the techniques can be found in the journal article Nguyen et al. (2012).
## Table 2.6: Clone Change Inconsistency Detection and Synchronization

<table>
<thead>
<tr>
<th>System</th>
<th>Revisions</th>
<th>Clone Pairs</th>
<th>Inconsistencies</th>
<th>Sync Pairs</th>
<th>Sync Tokens</th>
<th>2-Sync Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argouml</td>
<td>2000-3000</td>
<td>58</td>
<td>14</td>
<td>15</td>
<td>34</td>
<td>9</td>
</tr>
<tr>
<td>Columba</td>
<td>100-350</td>
<td>94</td>
<td>10</td>
<td>11</td>
<td>32</td>
<td>3</td>
</tr>
<tr>
<td>GEclipse</td>
<td>3000-4000</td>
<td>33</td>
<td>12</td>
<td>14</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>jEdit</td>
<td>4000-5000</td>
<td>397</td>
<td>49</td>
<td>62</td>
<td>17</td>
<td>22</td>
</tr>
<tr>
<td>ZK</td>
<td>9000-10000</td>
<td>81</td>
<td>22</td>
<td>48</td>
<td>12</td>
<td>33</td>
</tr>
</tbody>
</table>

were really bug fixes. Columns Range and Fix show the range of analyzed revisions and the number of bug fixing revisions among them for each system, respectively.

We ran JSync on those fixing revisions and found 107 clone-related fixing revisions out of 663 total ones, i.e. those revisions contain the fixes to some clone pairs (Column Cloned). The detailed numbers of those cloned pairs are in Column Clone Pairs. Column All displays the total number of clone pairs in the fixing revisions. Among them, some clone pairs have 1-side changes, i.e. only one member of the pair was changed while the other was not changed even in the later revisions in the project’s history (Column 1). Most of those pairs have 2-side changes, i.e. both members of each pair were changed/updated at the same revision (Column 2).

Interestingly, there are 18 pairs that have late synchronizing fixes in which one member of a pair was changed at that fixing revision and the other one was fixed accordingly at a later revision (Column late). Among those pairs, the reported numbers of clone pairs having inconsistencies on structural changes, identifier renaming, and value changes are shown in Columns S, N, and L, respectively. If a clone pair has two or more kinds of inconsistencies, they are counted accordingly.

This result shows that there exists inconsistent changes to cloned code in regard to all kinds of structure, naming, and values. JSync is able to detect them using its tree-based alignment and differencing. Importantly, there are 18 cases that the developers did not fix clone-related bugs until a later revision. Thus, JSync would be useful in those cases in helping developers recognize and fix earlier those bugs caused by clone inconsistencies.

Figure 2.5 shows such a case in ArgoUML. Two methods getExtendingClasses (on the left) and getExtendedClasses (on the right) of class CoreHelper are clones. At revision 2453, only method getExtendingClasses was changed by a renaming of a class type from MClass to MClassifier in the if statement for checking the type of the object client. In this one-side change scenario, JSync recognized that the two clones had naming inconsistencies. Then, at revision 2474, a change was made to getExtendedClasses in the same way (not shown).

An example of structural inconsistency is shown in Figure 2.6. In ArgoUML, two methods getIndex-OfChildren of two classes GoCollaborationInteraction (on the left) and GoProjectMachine (on the right) are
24

```java
public class CoreHelper {
    public Collection getExtendingClasses(MClassifier clazz) {
        if (clazz == null) return new ArrayList();
        Iterator it = clazz.getSpecializations().iterator();
        List list = new ArrayList();
        while (it.hasNext()) {
            MGeneralization gen = (MGeneralization) it.next();
            MGeneralizableElement client = gen.getChild();
            if (client instanceof MClass MClassifier) {
                list.add(client);
            }
        }
        return list;
    }
}
```

Figure 2.5: Change Inconsistency in Name in ArgoUML at revision 2453

```java
public class GoCollaborationInteraction {
    public int getIndexOfChild(Object parent, Object child) {
        if (!(parent instanceof MCollaboration)) return -1;
        Vector children = new Vector(getChildren(parent));
        if (children != null && children.contains(child)) {
            return children.indexOf(child);
        }
        return -1;
    }
}
```

Figure 2.6: Change Inconsistency in Structure in ArgoUML at revision 2761

clones. At revision 2761, only the method in the former class was modified with an addition of the if statement for checking the type of the object parent. The method in the latter class was not changed accordingly until revision 2858. **JSync** could detect that case and correctly recommend the synchronization with an addition of the corresponding if statement. In this example, the late synchronization was applied on cloned code in two different source files. Generally, developers could easily forget to synchronize cloned code in multiple source files. If **JSync** were used during the development, those inconsistencies could have been fixed sooner, i.e. **JSync** could prevent those (late updating) bugs from occurring, thus, improve software quality. Next, we will discuss its usefulness in supporting clone synchronization.

### 2.4.4.2 Clone Synchronization

The evaluation on synchronization was carried out on the revisions of five subject systems in Table 2.6 as follows. First, for each pair of consecutive revisions where a fix occurred, we identified all clone pairs containing changed members. Second, from each pair, we picked one changed clone member and
synchronized the other member according to its changes. Third, we compared each synchronized version with the one that was actually changed by the developers at that or a later revision. In each comparison, we checked if that pair was synchronized correctly or not by counting the number of tokens related the change and the number of correct tokens during synchronization.

The last nine columns in Table 2.6 show the result for this experiment. A pair is considered to be incorrectly synchronized if the result contains incorrect token(s). Columns T and F under Clone Pairs show the numbers of correctly and incorrectly synchronized pairs. Column 1-F shows the number of incorrect pairs having 1-side changes, because JSync recommends the change to the other member of a clone pair while it did not change. As Column F covers the cases in Column 1-F, we can see that there are only two incorrect pairs having 2-side or late-synchronization changes in JEdit and ZK. Examining those incorrect ones, we found that they occurred in the pairs that contained code clones which have same or similar behaviors/functionality but with different concrete surrounding code. In those cases, JSync incorrectly mapped the locations of the changed program elements between clone members, leading to incorrect recommendations.

To measure the synchronization accuracy in term of tokens, we computed the precision as the ratio between the number of correctly synchronized tokens over the number of all recommended ones. Column Sync Tokens shows the precision for all synchronized tokens while Column 2-Sync Tokens shows the precision for only 2-side or late-synchronization clone pairs. For all tokens, the precision is between 70% and 90%. Again, most of the incorrect ones belong to 1-side changes. Thus, the precision for 2-Sync Tokens is very high of 95%-100%. This result implies that there are a small number of cases that the inconsistent changes to clones are 1-side and do not require the suggested synchronization, however, for most of other cases that require consistent fixing, JSync is able to recommend with very high accuracy.

Let us now examine a case among the correctly synchronized pairs. Figure 2.7 shows two cloned methods taken from ZK revision 9322 (in two classes LabelImageElement and Audio). Later at revision 9323, method setImage in class LabelImageElement was changed with two modifications to two if statements and one addition of an assignment one (A to A’ in Figure 2.7). We ran JSync to synchronize this change to method setSrc in class Audio (B to B* in Figure 2.7). First, it detected all the changes to setImage. Then based on the clone matching and differencing between A and B, it identified the corresponding locations of modifications and insertions in B, and recommended the changes to B. This synchronized version of setSrc was compared to its actual change in the ZK repository, and we found that they are the same. That is, JSync correctly recommended this synchronization.
2.4.5 Threats to Validity

In our clone detection experiment, we used Bellon’s benchmark. Thus, our result has inherent threats to validity as in their work. For example, only two percent of all submitted clones was checked to build the benchmark, thus might not be representative for all the clones in those systems. In addition, human errors can be a factor in manually checking of clone fragments. Even though we ran JSync with different thresholds when comparing with other approaches, we used the default running parameters for CCFinder and Deckard, thus, their results might not be as good as those with tuning for each system. In the clone change management experiment, the heuristic of searching for bug fixing revisions by certain keywords helps in semi-automating the process but might miss some fixing revisions. In addition, since we did not know all the actual clones, we could only evaluate precision but not recall of the synchronization. Finally, the experiments were conducted on only open-source systems.

2.5 Related Work

Several approaches for code clone detection have been proposed Bellon et al. (2007); Tairas; Roy et al. (2009). Generally, they can be classified based on their code representations. The typical categories are text-based Ducasse et al. (1999); Marcus and Maletic (2001), token-based Baker (1997); Kamiya et al. (2002); Li et al. (2006); Mende et al. (2009), tree-based Baxter et al. (1998); Jiang et al. (2007a), and graph-based Gabel et al. (2008); Komondoor and Horwitz (2001). The text-based and token-based are usually efficient but could not detect Type-3 clones. In contrast, graph-based approaches,
though providing clones of higher level of abstraction, are time-consuming in detecting similar subgraphs. Krinke’s tool detects code clones via a program dependence graph (PDG) Krinke (2001). It finds the maximal isomorphic subgraphs in a PDG by calculating the maximal $k$-limited path induced subgraphs. Such induced subgraphs are defined as the maximal similar subgraphs that are induced by $k$-limited paths starting at two vertices. Their approach is more heavy-weight than our vector-based calculation after structural feature extraction in Exas.

Deckard Jiang et al. (2007a) introduced the use of vectors in clone detection. Our vector representation for tree-based fragments is more generalized. Deckard tool counts only distinct AST node types in a subtree for a fragment, while JSync captures structural features via paths and sibling sets.

Recent approaches have been proposed to support code clone management. CloneTracker, a clone management tool Duala-Ekoko and Robillard (2008), is based on a clone tracking tool with the same name Duala-Ekoko and Robillard (2007). CloneTracker Duala-Ekoko and Robillard (2008) uses CRD, a light-weight clone region description scheme, to map clone groups from the previous version to the current groups. However, some detected clones could be missed due to the approximate nature of CRD mapping Duala-Ekoko and Robillard (2007). In contrast, JSync uses its tree mapping algorithm, that avoids losing detected clones in clone tracking. Moreover, from one version to another, CloneTracker re-runs the detection only on changed and currently tracked files. Thus, it could miss cross-revision clone pairs. Such a pair is the result of a copy of a fragment from an un-changed file into a changed one.

In contrast to off-line clone management strategy in JSync, CLONEBOARD de Wit et al. (2009) tracks live changes to cloned code within Eclipse editor, analyzes them and provides resolution strategies for inconsistent modifications to clones. CLONEBOARD dynamically infers the clone relations via monitoring clipboard activity from a developer during an editing session.

Other clone tracking tools include Bakota et al. (2007); Jablonski and Hou (2007); Mende et al. (2009). Clone Detection Toolbox Mitter (2006) uses Unix diff to get the changes to clones, tracks them in different versions, and updates its clone database. However, it requires the re-run of clone detection on the entire new version. Furthermore, its line-based tracking of clones does not adapt well with modifications Duala-Ekoko and Robillard (2007). Bakota et al. (2007) proposed the mapping of clones from one version to another based on a light-weight AST-based similarity measure. More specifically, to compare the syntactical structures of two ASTs in two versions, their tool unparses the code of those ASTs, and then compares the two resulting strings using textual similarity. In contrast, JSync uses a novel tree edit script algorithm (Section 5.4.2) which better captures the code structure. In brief, the aforementioned clone tracking approaches might result in incompleteness in tracking/managing clones. None of them supports clone-aware synchronizing.
DejaVu Gabel et al. (2010) is a tool to detect the **syntactic inconsistency** bugs for cloned code. Given a code base, a parallel inconsistent clone analysis first enumerates all groups of non-similar cloned code fragments. Then, it analyzes clone groups and separates each group of inconsistent fragments into a fine-grained set of inconsistent changes and classifying each as benign or buggy Gabel et al. (2010). The key difference with JSync is that DejaVu utilizes a sequence alignment algorithm, while a novel tree-based change recovery algorithm is developed in JSync. More importantly, DejaVu does not aim to support clone synchronization. Juergens et al. Juergens et al. (2009) conducted a large-scale case study on both commercial and open-source systems and found that inconsistent changes to code clones are very frequent and creates a significant number of faults.

### 2.6 Conclusions

In this chapter, we introduce JSync, a tool for Java code clone detection. JSync models source code in a software system in terms of tree-based, logical entities, such as classes, methods, statements and expressions. A source code file is represented via an abstract syntax tree (AST) and a code fragment is represented as a sub-AST or a forest of consecutive sibling sub-ASTs. Two code fragments are considered as code clones if they are sufficiently similar. Their similarity is measured based on the distance of the characteristic vectors extracted from their corresponding tree structures.

Our empirical experiments on Bellon’s clone benchmark Bellon et al. (2007) and several other open-source software systems show that JSync outperforms the state-of-the-art clone detection approaches in terms of both correctness and time efficiency. JSync also raises the awareness of clone relation in software evolution by detecting inconsistent changes to clone pairs and suggesting synchronization.
CHAPTER 3. TEMPORAL SPECIFICATION MINING

The usage of multiple objects in object-oriented programming often follows certain specifications, i.e. specific temporal orders of method calls and/or control structures to perform some intended programming task. Unfortunately, those specifications are not always documented. The missing of that information creates long learning curve and, more importantly, leads to code smells and software bugs due to the misuse of objects. This chapter presents GrouMiner, a novel graph-based approach for mining the usage patterns of one or multiple objects. GrouMiner approach includes a graph-based representation for multiple object usages, a pattern mining algorithm, and an anomaly detection technique. Our experiments on several real-world programs show that GrouMiner is able to find useful usage patterns with multiple objects and control structures, and detect the usage anomalies that caused yet undiscovered defects in those programs.

3.1 Introduction

In object-oriented programming, developers usually deal with multiple objects of the same or different classes. Objects interact with one another via their provided methods and fields. The interplay of several objects, which involves objects’ fields/method calls and the control flow among them, often follows certain orders or control structure constraints that are parts of the intended usages.

In team development, newly introduced program-specific APIs by one or more team members often lack of usage documentation due to busy schedules. Other developers have to look through new code to understand the programming usages. This is a very inefficient, confusing, and error-prone process. Developers often do not know where to start. Even worse, they sometimes do not properly use newly introduced classes, leading to errors. Moreover, specific orders and/or control flows of objects’ method calls cannot be checked at compile time. As a consequence, errors could not be caught until testing and even go unnoticed for a long time. These also occur often in the case of general API usages.

In this chapter, we propose GrouMiner, a new approach for mining the usage patterns of objects and classes using graph-based algorithms. In GrouMiner, the usage of a set of objects in a scenario is repre-
presented as a labeled, directed acyclic graph (DAG), of which nodes represent objects' constructor calls, method calls, field accesses, and branching points of control structures, and edges represent temporal usage orders and data dependencies among them. A usage pattern is considered as a subgraph that frequently “appears” in the object usage graphs extracted from all methods in the code base. Appearance means that it is label-isomorphic to an induced subgraph of each object usage graph, i.e. satisfying all temporal orders and data dependencies between the corresponding nodes in that graph.

GrouMiner detects those patterns using a novel graph-based algorithm for mining the frequent induced subgraphs in a graph dataset. The patterns are generated increasingly by their sizes (in terms of the number of nodes). Each pattern $Q$ of size $k+1$ is discovered from a pattern $P$ of size $k$ via extending the occurrence(s) of $P$ in every method’s graph $G$ in the dataset with relevant nodes of $G$. The generated subgraphs are then compared to find isomorphic ones. To avoid the computational cost of graph isomorphism problem, we use Exas Nguyen et al. (2009a) (Section 2.2), our efficient structural feature extraction method to extract a characteristic vector for each subgraph. An Exas vector is an occurrence count vector of sequences of the labels of nodes and edges. The generated subgraphs having the same vector are considered isomorphic and counted toward the frequency of the corresponding candidate. If it exceeds a threshold, the candidate is considered as a pattern and is used to discover the larger patterns.

The mined patterns could assist developers in object usages. Those that are confirmed by the developers as typical usages could also be used to automatically detect the locations in programs that deviate from them. A portion of code is considered a violation of a pattern $P$ if the corresponding object usage graph contains only an instance of a strict sub-pattern $P$, i.e., not all properties of $P$ are satisfied. These locations are often referred to as violations and rare violations are considered as usage anomalies.

The departure points of GrouMiner from existing mining approaches for temporal object usages include two aspects. First, the mined patterns provide more information to assist developers in the usage flows among objects including control structures (e.g. conditions, loops, etc). Existing object mining approaches are limited to the patterns in the form of either (1) a set of pairs of method calls and in each pair, one call occurs before another, or (2) a partial order among method calls. Their patterns do not contain control structures or conditions among them. In other words, their detected patterns correspond to the subset of edges in GrouMiner’s pattern graphs. Second, GrouMiner’s mined patterns are for both common and program-specific cases with multiple interplaying/interacting objects, without requiring external inputs except the program itself. Existing approaches discover patterns involving methods of a single object without control structures.
The main contributions of this chapter include

1. An efficient and scalable graph-based mining algorithm for object usage patterns,

2. An automated, graph-based technique for detecting and ranking the anomalies in object usages,

3. An empirical study on real-world systems shows the benefits of our approach. The evaluation shows that our tool could efficiently detect a number of high-quality object usage patterns in several open source projects. GrouMiner is able to detect yet undiscovered defects caused by the misuse of the objects even in mature software.

Section 3.2 presents the mining technique in details. Section 3.3 describes our empirical evaluation of GrouMiner. Related work is given in Section 3.4. Conclusions appear in Section 3.5.

### 3.2 Mining Multiple Object Usage Patterns

This section describes our novel graph-based pattern mining algorithm for multiple object usages. Intuitively, an object usage is considered as a pattern if it frequently appears in source code. GrouMiner is interested only in the intra-procedural level of source code, therefore the groums are extracted from all methods, i.e. an object usage in each method is represented by a groum. However, in many cases, the object usages involve only some, but not all action and control nodes of an extracted groum in a method. In addition, the usages must include all temporal and data properties of those nodes, i.e. all involving edges. Therefore, in a groum representing a method, an object usage is an induced subgraph of that groum, i.e. involving some nodes and all the edges of such nodes. Note that any induced subgraph of a groum is also a groum.
Figure 3.1 shows an example. The usage pattern ABC of size 3 is used in four methods. The groum representing the pattern “appears” in four corresponding methods’ groupms. In each of the last two methods, it appears twice. However, in the groum 4, GrouMiner could only consider that the pattern is used once because the two corresponding sub-graphs are overlapped.

3.2.1 Formulation

**Definition 3.1** A groum is a directed acyclic graph (DAG). A node is either an action node or a control node. An action node represents an invocation of a constructor or a method, or an access to a field of one object. Label of an action node is “C,m” with C is the class’ name of the corresponding object and m is the method’s (or field’s) name. A control node represents the branching point of a control structure. Label of a control node is the name of its corresponding control structure. An edge represents a usage order and a data dependency. An edge from node A to node B means that A is used before B, i.e. A is generated before B in executable code, and A and B have a data dependency. Edges have no label.

Details about how to extract groum from source code can be found in Nguyen et al. (2009c).

**Definition 3.2** A groum dataset is a set of all groums extracted from the code base, denoted by \( D = \{G_1, G_2, ..., G_n\} \).

**Definition 3.3** An induced subgraph X of a groum \( G_i \) is called an occurrence of a groum P if X is equivalent, i.e. label-isomorphic, to P.

In Figure 3.1, the dataset contains four groums and the pattern ABC appears in all of them, with the total of six occurrences. We use \( G_i(P) \) to denote the occurrence set of P in \( G_i \) and \( D(P) = G_1(P) \cup G_2(P) \cup ... \cup G_n(P) \) to denote the set of all occurrences of P in the entire groum dataset. \( G_i(P) \) is empty if P does not occur in \( G_i \).

**Definition 3.4** The frequency of P in \( G_i \), denoted by \( f_i(P) \), is the maximum number of independent (i.e. non-overlapping) occurrences of P in \( G_i \). The frequency of P in the entire dataset, \( f(P) \), is the sum of frequencies of P in all groums in the dataset.

This definition implies that, if P occurs many times, only the non-overlapping occurrences are considered as different or independent. For example, in groum 3 of Figure 3.1, the pattern has two non-overlapping occurrences. In groum 4, two occurrences share node C, thus, they are considered as overlapping occurrences. Thus, the frequency of that pattern in groum 3 is two, that in groum 4 is one, and that in the whole dataset is five.
Definition 3.5 (Pattern) A group $P$ is called a pattern if $f(P) \geq \sigma$, i.e. $P$ has independently occurred at least $\sigma$ times in the entire group dataset. $\sigma$ is a chosen threshold.

Definition 3.6 (Pattern Mining Problem) Given the group dataset $D$ and the chosen pattern threshold $\sigma$, find the list $L$ of all patterns.

3.2.2 Algorithm Design Strategy

There have been many algorithms developed for mining frequent subgraphs on a graph dataset (i.e. multi-settings) or on a single graph. However, they are not applicable for this mining problem because (1) the existing mining algorithms for multi-settings count only one occurrence in each graph (i.e. the frequency of a candidate pattern is the number of graphs it occurs, which is different from our problem); and (2) mining algorithms on a single graph setting are developed for edge-oriented subgraphs, i.e. a subgraph is defined as a set of edges that form a weakly connected component. They are efficient on sparse graphs while our patterns are induced subgraphs of dense graphs Read and Corneil (1977).

We developed a novel mining algorithm for our problem, named PattExplorer, which has three following key design strategies: (1) incremental generation of candidates, (2) vector-based, approximated graph isomorphism, and (3) approximated frequency counting.

3.2.2.1 Incremental generation of candidates

The first design strategy of this algorithm is based on the following observation: isomorphic graphs also contain isomorphic (sub)graphs. Thus, sub-graphs of frequent (sub)graphs (i.e. patterns) are also frequent. In other words, larger patterns must contain smaller patterns. Therefore, the large patterns could be discovered (i.e. generated) from the smaller patterns. Based on this insight, PattExplorer mines the patterns increasingly by size (i.e. the number of nodes): patterns of a larger size are recursively discovered by exploring the patterns of smaller sizes. During this process, the occurrences of candidate patterns of size $k + 1$ are first generated from the occurrences of discovered patterns of size $k$ and those of size one. Then, the generated occurrences are grouped into isomorphic groups, each of which represents a candidate pattern. The frequency of each candidate is evaluated and if it is larger than a threshold, the candidate is considered as a pattern and is used to recursively discover larger patterns.

3.2.2.2 Vector-based, approximated graph isomorphism

The second design strategy comes from the fact that exact-matched graph isomorphism is highly expensive for dense graphs Nguyen et al. (2009a). A state-of-the-art algorithm for checking graph
function PattExplorer (D) 
  L ← {all patterns of size 1} 
  for each P ∈ L do Explore(P,L,D) 
  return L 

function Explore(P,L,D) 
  for each pattern of size 1 U ∈ L do 
    C ← P ⊕ U 
    for each Q ∈ patterns(C) do 
      if f(Q) ≥ σ then 
        L ← L ∪ {Q} 
        Explore(Q,L,D) 

Figure 3.2: Pattern Mining Algorithm - PattExplorer

isomorphism is canonical labeling Read and Corneil (1977), which works well with sparse graphs, but not with dense graphs. Our previous experiment Nguyen et al. (2009a) also confirmed this: it took 3,151 seconds to produce a unique canonical label for a graph with 388 nodes and 410 edges. Therefore, our algorithm employs an approximate vector-based approach. For each (sub)graph, PattExplorer extracts an Exas characteristic vector Nguyen et al. (2009a), an occurrence-counting vector of sequences of nodes and edges’ labels. Graphs having the same vector are considered as isomorphic. Exas was shown to be highly accurate, efficient, and scalable. For example, it took about 1 second to produce the vector for the aforementioned graph. It is about 100% accurate for graphs with sizes less than 10, and 94% accurate for sizes in 10-30. In our evaluation of GrouMiner, most patterns are of size less than 10. Details on Exas are in Nguyen et al. (2009a).

3.2.2.3 Approximated frequency counting

The third design strategy is due to the expensive computation of finding the maximal set of non-overlapping sub-graphs in the calculation of frequencies of the pattern candidates. An example is in groum 4 of Figure 3.1. In fact, this is equivalent to the problem of maximum independent set on graphs, since the overlapping relation could be represented as a graph in which the sub-graphs are considered as “nodes”, and their overlapping relations are considered as “edges”. Therefore, instead of exactly finding the maximal independent (i.e. non-overlapping) set of sub-graphs, PattExplorer does this approximately. That is, when a sub-graph is chosen to the independent set, its overlapping sub-graphs will be removed from the remaining set, i.e. will not be chosen.
3.2.3 Detailed Algorithm

The pseudo-code of PattExplorer is in Figure 3.2. First, the smallest patterns (i.e. patterns of size one) are collected into the list of patterns $L$ (line 2). Then, each of such patterns is used as a starting point for PattExplorer to recursively discover larger patterns by function Explore (line 3). The main steps of exploring a pattern $P$ (lines 6-12) are: 1) generating from $P$ the occurrences of candidate patterns (line 8), 2) grouping those occurrences into isomorphic groups (i.e. function patterns) and considering each group to represent a candidate pattern (line 9); 3) evaluating the frequency of each candidate pattern to find the true patterns and recursively discovering larger patterns from them (lines 10-12).

3.2.3.1 Generating Occurrences of Candidate Patterns

In the algorithm, each pattern $P$ is represented by $D(P)$, the set of its occurrences in the whole graph dataset. Each of such occurrences $X$ is a subgraph that might be extended into a larger subgraph by adding a new node $Y$ and all edges connecting $Y$ and the nodes of $X$. Let us denote that graph $X + Y$. Since a large pattern must contain a smaller pattern, $Y$ must be a frequent subgraph, i.e. an occurrence of a pattern $U$ of size 1. This will help to avoid generating non-pattern subgraphs (i.e. cannot belong to any larger pattern).

The operation $\oplus$ is used to denote the process of extending and generating all occurrences of candidate patterns from all occurrences of such two patterns $P$ and $U$:

$$P \oplus U = \{ X + Y | X \in G_i(P), Y \in G_i(U), i = 1..n \}.$$

3.2.3.2 Finding Candidate Patterns

To find candidate patterns, function patterns is applied on $C$, the set of all generated occurrences. It groups them into the sets of isomorphic subgraphs. Grouping criteria is based on Exas vectors. All subgraphs having the same vector are considered as isomorphic. Thus, they are the occurrences of the same candidate pattern and are collected into the same set. Then, for each of such candidate $Q$, the corresponding subgraphs are grouped by the graph that they belong to, i.e. are grouped into $G_1(Q), G_2(Q), ... G_n(Q)$, to identify its occurrence set in the whole graph dataset $D(Q)$.

3.2.3.3 Evaluating the Frequency

Function $f_i(Q)$ is to evaluate the frequency of $Q$ in each graph $G_i$. In general, such evaluation is equivalent to the maximum independent set problem because it needs to identify the maximal set of non-overlapping subgraphs of $G_i(Q)$. However, for efficiency, we use a greedy technique to find a
non-overlapping subset for $G_i(Q)$ with a size as large as possible. PattExplorer sorts the occurrences in $G_i(Q)$ descendingly by their numbers of nodes that could be added to them. As an occurrence is chosen in that order, its overlapping occurrences are removed. Thus, the resulting set contains only non-overlapping occurrences. Its size is assigned to $f_i(Q)$.

After all $f_i(Q)$ values are computed, the frequency of $Q$ in the whole dataset is calculated: $f(Q) = f_1(Q) + f_2(Q) + \ldots + f_n(Q)$. If $f(Q) \geq \sigma$, $Q$ is considered as a pattern and is used to recursively extend to discover larger patterns.

### 3.2.3.4 Disregarding Occurrences of Discovered Patterns

Since the discovery process is recursive, the occurrences of a discovered pattern could be generated more than once. (In fact, a sub-graph of size $k + 1$ might be generated at most $k + 1$ times from the sub-graphs of size $k$ it contains.) To avoid this redundancy, when generating the occurrences of candidate patterns, Explore checks if a sub-graph is an occurrence of a discovered pattern. It does this by comparing Exas vector of the sub-graph to those of stored patterns in $L$. If the answer is true, the sub-graph is disregarded in $P \oplus U$.

### 3.2.4 Anomaly Detection

The usage patterns can be used to automatically find the anomaly usages, i.e. the locations in programs that deviate from the typical object usages. The definition of an anomaly usage is adapted from Wasylkowski et al. (2007) for our graph-based representation.

Figure 3.3 shows an example where a BufferedReader is used without close(). $P$ is a usage pattern with a BufferedReader. $P_1$ is a sub-pattern of $P$, containing only two action nodes $<\text{init}>$ and readLine. A groum $G$ contains an occurrence of $P$, thus contains also another occurrence $G_1$ of $P_1$ as a subgraph of that occurrence of $P$. Another groum $H$ contains an occurrence $H_1$ of $P_1$ but no occurrence of $P$. Since $P_1$ is a sub-pattern of $P$, $H_1$ is called an inextensible occurrence of $P_1$ (i.e. it could not extend to an occurrence of $P$), thus is considered to violate $P$. Because containing $H_1$, $H$ is also considered to violate $P$. In contrast, $G_1$ is extensible, thus, $G_1$ and $G$ do not violate $P$.

However, not all violations are considered as defects. For example, there might exist the occurrences of the usage $<\text{init}>$-close() (without readLine) that also violate $P$, but they are acceptable. A violation is considered as an anomaly when it is too rare. The rareness of the violations could be measured by the ratio $v(P_1, P)/f(P_1)$, with $v(P_1, P)$ is the number of inextensible occurrences of $P_1$ corresponding to $P$ in the whole dataset. If rareness is smaller than a threshold, corresponding occurrences are considered as anomalies. The lower a rareness value is, the higher the anomaly is ranked.
Figure 3.3: A Violation Example. $G_1$ and $H_1$ are occurrences of $P_1$. $H_1$ violates $P$ while $G_1$ does not.

**Definition 3.7** A groun $H$ is considered as a usage anomaly of a pattern $P$ if $H$ has an inextensible occurrence $H_1$ of a sub-pattern $P_1$ of $P$ and the ratio $v(P_1, P)/f(P_1) < \delta$, with $v(P_1, P)$ is the number of such inextensible occurrences in the whole groum dataset and $\delta$ is a chosen threshold.

GrouMiner provides anomaly detection in two cases: (1) Detecting anomalies in the currently mined project (by using mined groums) and (2) Detecting anomalies when the project changes, i.e., in the new revision.

In both cases, the main task of anomaly detection is to find the inextensible occurrences of all patterns $P_1$ corresponding to the detected patterns. In the first case, because storing the occurrence set $D(P_1)$, GrouMiner can check each occurrence of $P_1$ in $D(P_1)$: if it is inextensible to any occurrence of a detected pattern $P$ generated from $P_1$, then it is a violation. Those violations are counted toward $v(P_1, P)$. After checking all occurrences of $P_1$, the rareness value $v(P_1, P)/f(P_1)$ is computed. If it is smaller than the threshold $\delta$, such a violation is reported as an anomaly. In the second case, GrouMiner must update the occurrence sets of detected patterns before finding the anomalies in the new version.

### 3.3 Empirical Evaluation

We run GrouMiner on several Java projects (Table 3.1) to evaluate its performance and effectiveness in detecting patterns. The experiments were carried out on a computer with Windows XP, Intel Core 2 Duo 2Ghz, 3GB RAM.

#### 3.3.1 Pattern Mining Evaluation

Table 3.1 shows the results that GrouMiner ran on nine different open-source projects with the total of more than 3,840 patterns. The number of groums $Groum$ and the maximum groum sizes $G_{max}$ are
Table 3.1: Pattern Mining Result ($\sigma = 6$, $\delta = 0.1$)

<table>
<thead>
<tr>
<th>Project</th>
<th>Class</th>
<th>Method</th>
<th>Groum</th>
<th>Gmax</th>
<th>Pattern</th>
<th>Pmax</th>
<th>Pattern by size</th>
<th>Time h:mm:ss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant 1.7.1</td>
<td>1123</td>
<td>12409</td>
<td>9573</td>
<td>153</td>
<td>697</td>
<td>17</td>
<td>317</td>
<td>315</td>
</tr>
<tr>
<td>Log4J 1.2.15</td>
<td>292</td>
<td>2479</td>
<td>1763</td>
<td>99</td>
<td>141</td>
<td>10</td>
<td>79</td>
<td>60</td>
</tr>
<tr>
<td>AspectJ 1.5.3</td>
<td>1500</td>
<td>14716</td>
<td>9818</td>
<td>332</td>
<td>1055</td>
<td>15</td>
<td>429</td>
<td>413</td>
</tr>
<tr>
<td>Axis 1.1</td>
<td>1127</td>
<td>7834</td>
<td>5355</td>
<td>425</td>
<td>614</td>
<td>16</td>
<td>251</td>
<td>258</td>
</tr>
<tr>
<td>Columba 1.4</td>
<td>799</td>
<td>5083</td>
<td>3024</td>
<td>185</td>
<td>219</td>
<td>7</td>
<td>118</td>
<td>94</td>
</tr>
<tr>
<td>Fluid 12.0.5</td>
<td>229</td>
<td>3506</td>
<td>2477</td>
<td>115</td>
<td>236</td>
<td>14</td>
<td>92</td>
<td>94</td>
</tr>
<tr>
<td>jEdit 3.0</td>
<td>204</td>
<td>2274</td>
<td>1757</td>
<td>244</td>
<td>238</td>
<td>10</td>
<td>119</td>
<td>77</td>
</tr>
<tr>
<td>Jigsaw 2.0.5</td>
<td>701</td>
<td>6528</td>
<td>5073</td>
<td>152</td>
<td>443</td>
<td>11</td>
<td>197</td>
<td>204</td>
</tr>
<tr>
<td>Struts 1.2.6</td>
<td>365</td>
<td>3209</td>
<td>2412</td>
<td>107</td>
<td>198</td>
<td>8</td>
<td>62</td>
<td>114</td>
</tr>
</tbody>
</table>

A Usage Example

```java
StringBuffer sb = new StringBuffer();
sb.append("{");
for ( Iterator iter = supportedTargets.iterator(); iter.hasNext();)
    String eva = (String) iter.next();
    sb.append(eva);
    if ( iter.hasNext() ) sb.append(",");
sb.append("}");
return sb.toString();
```

Pattern

```java
StringBuffer aStringBuffer = new StringBuffer();
aStringBuffer.append(String);
for ( Iterator iterator = Set.iterator(); iterator.hasNext();)
    String aString = iterator.next();
    aStringBuffer.append(aString);
    if ( iterator.hasNext() ) aStringBuffer.append(String);
aStringBuffer.append(String);
return aStringBuffer.toString();
```

Figure 3.4: A Common Usage Pattern Mined from AspectJ

quite large. The number of method groums is smaller than that of methods because many methods are abstract methods or in interfaces, thus, have no bodies, or have bodies that do not involve objects.

Table 3.1 shows that GrouMiner is quite efficient and can scale up to large graphs. The total size of graphs for AspectJ system is about 70,000 nodes. However, the pattern detection time is very reasonable (a few minutes for small systems, to a half an hour and an hour for large systems). The time depends more on the distribution nature of patterns and the groums of each system, rather than its size. In Table 3.1, we counted the total number of distinct patterns and eliminated the patterns that are contained within others. The numbers of detected patterns with the sizes of 3 or more are about 44%-69% of the total numbers. This is also an advantage of GrouMiner over existing approaches, which focus on patterns of pairs or a set of pairs of method calls. Moreover, many GrouMiner’s patterns are program-specific. The maximum size of patterns Pmax varies in different projects, and ranges from 7-17.

Case Study. Figure 3.4 shows a pattern mined from AspectJ to illustrate a routine to convert a Set to a String using StringBuffer and Iterator. GrouMiner is able to detect this pattern with four interplaying objects and the control structures for, if among method calls. For object iter, JADE/Weasel et al. (2007), a well-known object usage miner, would produce a pattern $P = \{ \text{hasNext()} < \text{hasNext()}, \text{hasNext()} < \text{next()} \}$ ($<$ means “occurs before”), thus, providing less information.
Table 3.2: Anomaly Detection and Manual Checking

<table>
<thead>
<tr>
<th>Project</th>
<th>Anomaly</th>
<th>Check</th>
<th>Defect</th>
<th>False positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant 1.7.1</td>
<td>145</td>
<td>15</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>Log4J 1.2.15</td>
<td>32</td>
<td>15</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>AspectJ 1.5.3</td>
<td>244</td>
<td>15</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>Axis 1.1</td>
<td>145</td>
<td>15</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>Columba 1.4</td>
<td>40</td>
<td>15</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>jEdit 3.0</td>
<td>47</td>
<td>15</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>Jigsaw 2.0.5</td>
<td>115</td>
<td>15</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>Struts 1.2.6</td>
<td>33</td>
<td>15</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>Fluid 12.05</td>
<td>64</td>
<td>64</td>
<td>5</td>
<td>49</td>
</tr>
</tbody>
</table>

3.3.2 Anomaly Detection Evaluation

We ran anomaly detection on all nine systems (Table 3.2). We chose to examine all 64 reported anomalies for the Fluid project where we have the domain knowledge. For other systems, we check top 15 anomalies and manually classified them. In Fluid, there are 3 defects among top 10 anomalies and 5 defects among top 15 anomalies. In addition to 5 defects found in Fluid, GrouMiner can reveal 5 more new defects in other mature software such as Ant, AspectJ, Columba, jEdit, and Jigsaw. All defects are both common and program-specific. Carefully examining those additional ones, we found that they are in the form of missing necessary steps in using the objects and missing condition and control structures. For example, in PointcutRewriter.simplifyAnd() in AspectJ, the use of Iterator.next() was not preceded by an Iterator.hasNext().

Similarly, in the method MapEntry.parseRestNCSA() of Jigsaw 2.0.5, the call to StringTokenizer.nextToken() was not preceded by StringTokenizer.hasNext(). On the other hand, the usage of ICloseableIterator in the method AbstractMessageFolder.recreateMessageFolderInfo of Columba and that of BufferedReader in the method Registers.toString of jEdit missed a ICloseableIterator.close() and a BufferedReader.close(), respectively.

3.4 Related Work

There exist several methods for mining temporal program behaviors. The closest research to GrouMiner is JADET Wasylkowski et al. (2007). For each Java object in a method, JADET extracts a usage model in term of a finite state automaton (FSA) with anonymous states and transitions labeled with feasible method calls. The role of JADET’s object usage model is similar in spirit to our groum. However, its model is built for a single object and does not contain control structures. GrouMiner’s graphs represent the usage of multiple objects including their interactions, control flow and condition nodes among method calls. Another key difference is that GrouMiner performs frequent subgraph
mining on object usage graphs to find graph-based patterns. In contrast, JADET uses frequent itemset
mining to extracts a pattern in term of a set of pairs of method calls.

Dynamine Livshits and Zimmermann (2005) looks at the set of methods that were inserted between
versions of a software to mine usage patterns. Each pattern is a pair of method calls. Engler et al. Engler et al. (2001)’s approach is also limited to patterns of pairs of method calls. Thus, each
pattern corresponds to an edge in a GrouMiner’s pattern. Acharya et al. Acharya et al. (2007) mine
API call patterns using a frequent closed partial order mining algorithm and express them in term of
partial orders of API method calls. Their patterns do not have controls and conditions and do not
handle multiple object usages. Williams and Hollingsworth Williams and Hollingsworth (2005) mine
method usage patterns in which one function is directly called before another. Chang et al. Chang
et al. (2008) use a maximal frequent subgraph mining algorithm to find patterns on condition nodes
on PDGs. Their approach considered a set of nodes surrounding the control points in PDGs. In
Alattin Thummalapenta and Xie (2009), the authors formulate the alternative pattern detection based
on frequent item set mining, and then use such patterns to improve false positives in detecting neglected
et al. (2006) builds possible API usage orders determined by a predefined template for given APIs.

PR-Miner Li and Zhou (2005) uses the frequent itemset mining technique to find the functions,
variables, data types that frequently appear in same methods. No order of method calls is considered as
in GrouMiner. CP-Miner Li et al. (2006) uses frequent subsequence mining to detect clone-related bugs.
Some clone detection approaches apply graph-based techniques, but are limited in scalability Krinke
(2001). BugMem Kim et al. (2006a) mines patterns of defects and fixes from the version history.

Given an API sample, XSnippet Sahavechaphan and Claypool (2006) provides example code of that
API. In contrast, GrouMiner does not require a sample as an input and it detects anomalies. Similar tools
include Prospector Mandelin et al. (2005a) and MAPO Xie and Pei (2006). PARSEWeb Thummalapenta
and Xie (2007) takes queries of the form “from source object type to destination object type” as an
input, and suggests relevant method-invocation sequences as potential solutions. CodeWeb Michail
(2000) detects patterns in term of associate rules among classes. In Doc2Spec Zhong et al. (2009b), the
authors analyze API documentation using natural language processing to infer resource specifications.

Ammons et al. Ammons et al. (2002) observe execution traces and mine usage patterns in term of
probabilistic FSAs. Shoham et al. Shoham et al. (2007) apply static inter-procedural analysis for mining
API specifications in term of FSAs. Both approaches require the alphabet of an FSA specification to
be known. Pradel and Gross Pradel and Gross (2009) addressed also multiple object usages but using
dynamic analysis to recover FSAs from execution traces. In Tikanga Wasylkowski and Zeller (2009),
static analysis is combined with model checking to mine temporal specifications in terms of Computation Tree Logic formulas.

Gabel et al. Gabel and Su (2008) mine temporal properties between method calls in execution traces and express a specification as an automaton. However, their approach does not distinguish methods from different objects. Yang et al. Yang et al. (2006) find behavioral patterns that fit into user-provided templates. Chronicler Ramanathan et al. (2007a) use inter-procedural analysis to find and detect violations of function precedence protocols. Kremenev Kremenev et al. (2006a) use a factor graph, a probabilistic model, to mine API method calls. Lo et al. Lo and Maoz (2009) use object hierarchies over traces of inter-object method calls as a refinement mechanism to mine hierarchical scenario-based specifications. Other approaches take as input a single type and derive the valid usage patterns as an FSA using static analysis or model checking Alur et al. (2005); Henzinger et al. (2005); Liu et al. (2006).


3.5 Conclusions

The information on specific protocols among method calls of multiple interplaying objects is not always documented. This chapter introduces GrouMiner, a novel graph-based approach for mining usage patterns for multiple objects. The mined patterns can be used to assist developers in learning new code usages and be used to detect both common and program-specific usage anomalies and violations. GrouMiner includes a graph-based representation for multiple object usage patterns, an efficient graph-based mining algorithm to discover such patterns from source code, and a graph-based technique to detect object usage anomalies. The advantages of GrouMiner include useful detected patterns with control and condition structures among method calls of objects, scalable pattern discovery and anomaly detection. Our empirical evaluation shows that GrouMiner is able to find interesting patterns and to detect yet undiscovered defects.
CHAPTER 4. API PRECONDITION MINING

Modern software relies on existing application programming interfaces (APIs) from libraries. Formal specifications for the APIs enable many software engineering tasks as well as help developers correctly use them. In this work, we mine large-scale repositories of existing open-source software to derive potential preconditions for API methods. Our key idea is that APIs’ preconditions would appear frequently in an ultra-large code corpus with a large number of API usages, while project-specific conditions will occur less frequently. First, we find all client methods invoking APIs. We then compute a control dependence relation from each call site and mine the potential conditions used to reach those call sites. We use these guard conditions as a starting point to automatically infer the preconditions for each API. We analyzed almost 120 million lines of code from SourceForge and Apache projects to infer preconditions for the standard Java Development Kit (JDK) library. The results show that our technique can achieve high accuracy with recall from 75–80% and precision from 82–84%. We also found 5 preconditions missing from human written specifications. They were all confirmed by a specification expert. In a user study, participants found 82% of the mined preconditions as a good starting point for writing specifications. Using our mining result, we also built a benchmark of more than 4,000 precondition-related bugs.

4.1 Introduction

Software in our modern world is developed using frameworks and libraries, which provide application programming interfaces (APIs) via classes and their methods. To be able to correctly use these APIs, programmers must conform to their specifications. For example, in the standard Java Development Kit (JDK), a call to next() in a LinkedList needs to be preceded by a call to hasNext() to ensure the list still has elements. For each API method, there are conditions that must hold whenever it is invoked. These are called the preconditions of the API. For example, in the JDK String class, the condition ‘beginIndex <= endIndex’ must hold when the method subString(beginIndex,endIndex) is called. These conditions, as part of the API’s specification, have been shown to be useful for many automated software engineering tasks including the formal verification of program correctness Ammons et al. (2002); Ball
Manually defining specifications for libraries is time-consuming. One must read the documentation of the APIs and even the source code and convert the conditions to the formats suitable for verification tools. To ease defining specifications, several approaches have been proposed to automatically derive the specifications. Generally, there are two types of approaches that complement each other: program analysis-based and data mining-based approaches.

Among program analysis approaches, dynamic approaches Ammons et al. (2002); Beschastnikh et al. (2011); Ernst et al. (1999); Mariani and Pastore (2008); Weimer and Necula (2005) could detect data and temporal invariants and recover program behaviors. However, they require a large number of test cases, and their results might be incomplete due to the incompleteness of the test suites. On the other hand, static analysis approaches do not require dynamic instrumentation but have high false-positive specifications Engler et al. (2001); Kremenek et al. (2006b); Ramanathan et al. (2007b); Wei et al. (2011). Importantly, those static techniques focus their analyses only on an individual project, which has the call sites for only a small number of APIs.

In contrast to program analysis-based approaches, other techniques in the mining software repositories (MSR) area have applied mainly data mining to derive API specifications from code repositories Gabel and Su (2008); Li and Zhou (2005); Livshits and Zimmermann (2005); Nguyen et al. (2009c); Wasylkowski and Zeller (2009); Wasylkowski et al. (2007); Yang et al. (2006); Zhong et al. (2009a). The key difference of these mining approaches from the traditional program-analysis based approaches is that they consider the usages of the APIs at the call sites in the client programs of the APIs to derive the conditions regarding only the usage orders or temporal orders among the API calls. While some approaches detect such orders as pairs of method calls Wasylkowski et al. (2007); Gabel and Su (2008); Williams and Hollingsworth (2005) (e.g., \( p \) must be called before \( q \)), other approaches mine the sequences of calls Zhong et al. (2009a); Thummalapenta and Xie (2009) or even a graph or finite state diagram of method calls Nguyen et al. (2009c); Pradel and Gross (2009); Wasylkowski and Zeller (2009). Other mining approaches focus on associations of API entities Li and Zhou (2005); Livshits and Zimmermann (2005). Unfortunately, those mining approaches do not aim to recover pre- and post-conditions as part of specifications. Moreover, except a few methods Ramanathan et al. (2007b), they mainly rely on mining techniques without in-depth analyzing the data and control properties in the mined code.
This chapter introduces an approach that puts forth the idea of mining API specifications. Our approach combines both static analysis and source code mining from a very large code corpus in open-source repositories to *derive the preconditions of APIs* in libraries and frameworks. We expect that the **APIs’ preconditions would appear frequently in an ultra-large corpus of open-source repositories** that contain a very large number of the usages of those APIs, while **project-specific conditions will occur less frequently**. Importantly, we *combine the strength of both static analysis approaches (via control dependency analysis) and MSR approaches (via mining)* to make it scale to large corpus. Moreover, we can derive preconditions for a large number of APIs or entire library at the same time.

Specifically, we used a very large-scale data set from **SourceForge** consisting of 3,413 Java projects with 497,453 source files, 4,735,151 methods and 92,495,410 SLOCs, and from Apache consisting of 146 projects with 132,951 source files, 1,243,911 methods and 25,117,837 SLOCs. To analyze the APIs’ client code in such large data set, we did not choose the dynamic analysis approach since it would require the generation of a very large number of test cases and a great deal of execution time. Instead, we develop a light-weight, intra-procedural, static analysis technique to collect all predicates for every API method in the data set. Our technique first builds the control dependence relation for each method. It then analyzes different paths and conditions that lead to each method to recover all primitive predicates for all API methods in the data set. After that, it will start mining on the preconditions by performing normalization, merging, filtering, and ranking on them.

In our empirical evaluation, we compared the mined preconditions with the real-world JML specifications for several JDK APIs that are created and maintained by the JML team Leavens. The results show that our precondition mining technique can achieve high accuracy with recall from 75–80% and precision from 82–84% for the top-ranked results. We also found 5 new preconditions (for two JDK classes) that were not listed by the JML team. We reported to the team and got their confirmation on those preconditions. Moreover, we filed to the JML team the preconditions for 11 previously unspecified methods in 2 JDK classes, and they accepted all proposals. Importantly, our method is light-weight and scales to such large amount of code, allowing us to derive preconditions for entire JDK library. We also conducted a user study on human subjects who have experience with specifications on the usefulness and correctness of our mined preconditions. 82% of the participants found that our result is a good starting point for writing specifications for APIs under study. In addition to supporting specification writing, we show the usefulness of our mined preconditions by using them to build a benchmark of more than 4,000 API call sites that might be buggy due to missing precondition checking. It is useful for tools to detect neglected conditions Chang et al. (2008).
The key contributions of this paper include:

1. A novel approach that combines the strength from both code mining in a *ultra-large code corpus* and program analysis, to derive the preconditions of APIs in libraries and frameworks,

2. An empirical evaluation on a very large-scale data set to mine preconditions of JDK APIs.

Section 4.2 will explain an example that motivates our approach. Our key program analysis and mining technique is presented in Section 4.3. Section 4.4 is for our empirical evaluation. Related work is described in Section 4.5. Section 4.6 concludes the chapter.

### 4.2 Motivating Example

Let us consider a commonly-used API from the Java Development Kit (JDK): `String.substring(int, int)` in package `java.lang`. The method takes as input two integer values: `beginIndex`, the index of the starting character (inclusive) and `endIndex`, the index of the the ending character (exclusive). The method returns a new string that is the substring of the original string, using the two indices. Examining this API, we could learn that there are three preconditions that must hold before it is called:

1. `beginIndex ≥ 0`,
2. `endIndex ≤ this.length()` and
3. `beginIndex ≤ endIndex`.

A precondition for an API to be used could involve the *receiver object* of the API and/or one or multiple of *its arguments*. Identifying the complete set of preconditions for an API is a difficult and time-consuming task. However, this particular API is extremely popular (one of the most frequently used APIs in JDK) and there is another way to learn these preconditions, without having to even look at the documentation or source code for the method. Consider one example usage of this API as shown in Figure 4.1. The method `Request.getPathFragmentation(...)` in the SeMoA SeMoA project uses this API (lines 19–20). Examining the source, we can see several conditions that must be false in order for the control-flow to reach the API calls. For example, the if statement on line 2 must be false, meaning that both indices `servletPathStart` and `extraPathStart` must be non-negative, the indices must not be greater than the length of the string `completePath`, and `servletPathStart` must not be greater than `extraPathStart`. These are the same conditions we saw in the documentation. This gives us our first observation:

**Observation 1** **Preconditions can be inferred by looking at the conditions that must be satisfied before calling the APIs, i.e., the guard conditions of the API call sites.**
public boolean setPathFragmentation(int servletPathStart, int extraPathStart) {
    if (servletPathStart < 0 || extraPathStart < 0 ||
        servletPathStart > completePath_length() ||
        extraPathStart > completePath_length() ||
        servletPathStart > extraPathStart)
        return false;
    if (servletPathStart == completePath_length()) {
        ... return true;
    } else if (completePath.charAt(servletPathStart) != '/')
        return false;
    if (extraPathStart == completePath_length()) {
        ... return true;
    } else if (completePath.charAt(extraPathStart) != '/')
        return false;
    contentPath_ = completePath.substring(0, servletPathStart);
    servletPath_ = completePath.substring(servletPathStart, extraPathStart);
    ... return true;
}

Figure 4.1: Client code of API String.substring(int,int) in project SeMoA at revision 1929.

Let us consider line 19 of Figure 4.1. It contains another call to the API. The only difference is that at this call site, instead of a variable, constant value 0 is passed as the first argument. Thus, the conditions on an argument of an API can be derived from the properties of such value passed to the API. This gives the observation:

**Observation 2** The mining tool should take into account the properties of the arguments passed as the APIs’ parameters.

This client code however contains other conditions checked before the API call. Some of these conditions are specific to the logic of the client (lines 11 and 17). This gives our next observation:

**Observation 3** Call sites might contain client-specific conditions, which could cause noise when inferring preconditions. Thus, an approach that mines preconditions from call sites should attempt to minimize such noise.

This has been a challenge for the existing static program analysis-based approaches Ramanathan et al. (2007b) when they examine the call sites of the APIs only within the code of the APIs’ programs.

One way to minimize noise is to mine preconditions from a large number of projects. The valid preconditions should appear more frequently, while client-specific conditions should appear infrequently.
private String getCommand(int pc, boolean allThisLine, boolean addSemi) {
    if (pc >= lineIndices.length) {
        return "";
    } else if (allThisLine) {
        ...
    }
    return "";

    int ichBegin = lineIndices[pc][0];
    int ichEnd = lineIndices[pc][1];
    ...
    String s = "";
    if (ichBegin < 0 || ichEnd <= ichBegin || ichEnd > script.length()) {
        return "";
    }
    try {
        s = script.substring(ichBegin, ichEnd);
    }
}

Figure 4.2: Client code of API String.substring(int, int) in project Jmol at revision 18626.

<table>
<thead>
<tr>
<th>Receiver Object (rcv)</th>
<th>beginIndex</th>
<th>endIndex</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcv.length() &gt; 0</td>
<td>rcv.length() &gt; beginIndex</td>
<td>endIndex &gt;= 0</td>
</tr>
<tr>
<td>rcv.length() &gt;= endIndex</td>
<td>beginIndex &lt;= endIndex</td>
<td>endIndex != -1</td>
</tr>
<tr>
<td>rcv.length() &gt; beginIndex</td>
<td>beginIndex &gt;= 0</td>
<td>rcv.length() &gt;= endIndex</td>
</tr>
</tbody>
</table>

Table 4.1: Mined Preconditions for String.substring(int, int)

Figure 4.2 shows another client, Jmol Jmol, that uses the same API (line 15) in the method ScriptEvaluator.getCommand(...).

The if statement on line 12 checks the three required preconditions. Note that, in this case, the checked condition is stronger than the required one: the beginning index ichBegin is strictly less than the ending index ichEnd. This gives our next observation:

Observation 4 The relationship between conditions should be considered when mining preconditions.

For example, a stronger condition should be counted as an instance of a weaker one. A mining tool must consider the relations among conditions to derive a precondition. Similar to the previous client code, this method also contains client-specific conditions (lines 2 and 4). Again, these conditions are project-specific and not actual preconditions for the API in question. However, these conditions do not appear in the first client code which shows evidence that such noise would appear less frequently.

Motivation. This example motivates us to use an approach to mine the preconditions via the guard conditions of the call sites of the APIs under study in a very large number of projects in a large-scale corpus. That would help to minimize the project-specific conditions (as noises) because they will appear less frequently in the large corpus. The true preconditions would occur more frequently.
In this chapter, we introduce such an approach that mines the preconditions of the APIs. In fact, after running our mining tool on a very large data set from SourceForge (consisting of 3,413 Java projects with 497,453 source files, 600,274 classes, 4,735,151 methods, and 92,495,410 SLOCs), we are able to derive the preconditions for the `String.substring` method in JDK. The columns in Table 4.1 show the preconditions with highest frequencies in the corpus that we mined for the receiver `String` object, and the arguments `beginIndex` and `endIndex`, respectively. As seen, the aforementioned true preconditions have among the highest frequencies. Project-specific conditions did not make the top of the list.

### 4.3 Mining with Large Code Corpus

Let us outline our approach for mining the preconditions for API methods. Figure 4.3 gives an overview, which can be summarized as:

1. The input is the set of all API methods under analysis and client projects to mine.

2. For each method in the corpus that calls an API, we build the control dependence relation between each method call and the predicates in the method (from the control-flow graph) and identify all preconditions of API calls. (Section 4.3.1)

3. Next, we normalize the preconditions to identify and combine the equivalent ones. (Section 4.3.2)
4. We then analyze the preconditions to infer additional ones which are not directly present in the client code. (Section 4.3.3)

5. Finally we filter out non-frequent preconditions (Section 4.3.4, and rank the remaining ones in our final result. (Section 4.3.5)

4.3.1 Control Dependence and Preconditions

In order to identify the preconditions of API calls, we need to identify all predicates that guard the evaluation of each method call in the program. This can be done by building the control dependence relation Ferrante et al. (1987), based on the control-flow graph (CFG). In a CFG, each predicate node has exactly two outgoing edges labeled TRUE and FALSE representing the two corresponding branches.

**Definition 4.1** A method call \( C \) is control-dependent on a predicate expression \( p \) if and only if on the corresponding CFG, all directed paths from \( p \) to \( C \) go out of \( p \) on the same edge—TRUE or FALSE.

This means that \( C \) is control-dependent on \( p \) if \( C \) is executed in only one branch of \( p \). If \( C \) could be called in both branches of \( p \), then \( C \)'s execution does not depend on \( p \). For example, in Figure 4.2, String.substring on line 15 is called only in the FALSE branch of the predicate on line 12, thus, it is control-dependent on that predicate. Our definition is stricter than the traditional definition by Ferrante et al. Ferrante et al. (1987), which requires \( C \) always be called in one branch of \( p \) and not called in at least one path in the other branch. According to that, a method could be called in both TRUE and FALSE branches of the predicate on which it is control-dependent, thus the value of the predicate does not control the execution of the method call. This is the reason we give an adaptation in Definition 4.1.

**Definition 4.2** An API method \( M \) is control-dependent on a predicate expression \( p \) in a client method if and only if all call sites of \( M \) in the client method are control-dependent on exactly one branch of \( p \) (TRUE or FALSE).

When \( M \) is control-dependent on the FALSE branch of \( p \), the predicate that guards \( M \) will be the negation of the predicate expression in \( p \). We now define what we consider to be a precondition for calling a method.

**Definition 4.3** A precondition of a method call is a single clause in the conjunctive normal form (CNF) of a predicate on which the method call is control-dependent.

In Figure 4.2, the API call on line 15 is control-dependent on the FALSE branch of the if statement on line 12, so the predicate is negated and gives us: !(ichBegin < 0 || ichEnd ≤ ichBegin ||
ichEnd \geq 0 \land ichEnd > ichBegin \land ichEnd \leq script.length()). This predicate is represented in CNF as \(! (ichBegin < 0) \land \! (ichEnd \leq ichBegin) \land \! (ichEnd > script.length())\). Moving the negations inside, we have a set of three preconditions: ichBegin \geq 0, ichEnd > ichBegin and ichEnd \leq script.length().

For the goal of deriving general specifications, the context-specific names/expressions must be abstracted from the individual method call sites. Since each call contains a receiver and list of arguments, we are interested in the preconditions on each of these components. We use rcv and argi as the symbolic names for the receiver and the i-th argument in the list of arguments, respectively. First, we match the expression of the receiver and that of each parameter of the method call against the expression of the precondition. Then, we try all possible substitutions of occurrences of the receiver and parameters with their corresponding symbolic names. If the condition contains a variable/field, its latest value will be used in the precondition. Its latest value is the expression in the right hand side of its most recent assignment (if any). In the above example, processing the three preconditions ichBegin \geq 0, ichEnd > ichBegin and ichEnd \leq script.length() of the method call script.substring(ichBegin, ichEnd) produces the following abstracted preconditions arg0 \geq 0, arg1 > arg0 and arg1 \leq rcv.length(). A condition that does not involve any component of the call (i.e., having no symbolic names) will be discarded.

Finally, to follow Observation 2, for each expression e passed as argument argi to a method call, we create a precondition in three cases. First, if e is a constant of a primitive type, we create a precondition argi == c. Second, if e is an expression that can be recognized via its syntax as returning a non-null object, e.g., object instantiation or array initialization expression, we create a precondition argi != null. Third, if e involves any component of the call, i.e., having some symbolic names, we create a precondition argi == e', where e' is obtained from e by replacing identifiers with the corresponding symbolic names, e.g., arg1 == rcv.length(). These equality preconditions are used to support the inference of the non-strict inequality preconditions such as arg1 >= 0 or arg1 > rcv.length().
for each precondition \( p = (a == b) \)

for each precondition \( q = (a > b) \) or \( q = (a < b) \)

if \( q == (a > b) \) then \( t = (a >= b) \)

else \( t = (a <= b) \)

if \( |\Omega(p)| = |\Omega(q)| \) then \( \Omega(t) = \Omega(t) \cup \Omega(p) \cup \Omega(q) \)

else if \( |\Omega(p)| > |\Omega(q)| \) then \( \Omega(t) = \Omega(t) \cup \Omega(q) \)

else \( \Omega(t) = \Omega(t) \cup \Omega(p) \)

Figure 4.4: Inferring non-strict inequality preconditions.

Table 4.2 shows the resulting preconditions mined from our example API using this process. For each API, the preconditions are stored in a map \( \Omega \), in which \( \Omega(p) \) returns the set of calling methods containing precondition \( p \) before calling the API.

4.3.2 Precondition Normalization

Since we collect preconditions from call sites in different methods and projects, there are conditions that are equivalent but expressed in different forms. For example, the following: \( \text{arg1} > \text{arg0}, \text{arg0} < \text{arg1}, (\text{arg0} - \text{arg1}) < 0, (\text{arg1} - \text{arg0}) > 0 \) and \( \text{arg0} - 1 < \text{arg1} - 1 \) express the same conditions. Thus, we need to normalize the preconditions. The first step is to ensure every unary/binary expression is enclosed by exactly one pair of opening and closing parentheses. The next step is to order the operands in the binary operation(s) (such as \( >, < \), etc.) of the preconditions so that they are comparable between call sites. Whenever two operands of a binary operation are re-ordered, the operator is reversed correspondingly.

For any comparison expression \( E = E_l \geq E_r, \) where \( \geq \) is a comparison operator, we transform it into \( E' = E'_l \geq E'_r, \) where \( E'_l \) contains only literals and \( E'_r \) contains all symbolic and other identifier names. If \( E'_l \) contains all numeric literals, it will be evaluated. The terms in \( E'_r \) are ordered in the ascending order of its names. For example, all 5 conditions above will be normalized into the same condition \( (\text{arg0} - \text{arg1}) < 0. \) Finally, the map \( \Omega \) is updated with the normalized preconditions for each API.

4.3.3 Precondition Inference

Inferring non-strict inequality preconditions. In the client code, a non-strict inequality precondition \( (a >= b \) or \( a <= b \) might be split into strict inequality \( (a > b \) or \( a < b \) and equality \( (a == b \) conditions, and checked at different call sites. Figure 4.4 shows our algorithm for inferring the non-strict inequality precondition. When the two preconditions \( p \) and \( q \) are used equally, all call sites for both of them are counted toward the inferred condition (line 5). Otherwise, only the call sites of the less-frequently used precondition are added (lines 6 and 7). This helps us avoid counting the occurrence frequencies of incorrect conditions toward the inferred one.
Merging strong and weak conditions. Among the preconditions, some imply others (Observation 4). If a stronger condition holds, the weaker condition holds too. This means that all call sites of the stronger condition could be merged to (counted toward) those of the weaker. However, merging can lead to inferring wrong preconditions if the weaker one is in buggy code or specific to a particular client (Observation 3). We avoid this noise by using the assumption that the more frequently a precondition is checked, the more likely it is correct. Thus, if the stronger condition is less-frequently checked than the weaker one, its call sites will be merged to those of the weaker and it will be removed from the set of preconditions.

\begin{figure}[h]
\begin{lstlisting}[language=Python]
for each pair of preconditions \((p, q)\) if \(p \rightarrow q \land |\Omega(p)| \leq |\Omega(q)|\) then 
\(\Omega(q) = \Omega(q) \cup \Omega(p)\)
remove \(p\) from \(\Omega\)
\end{lstlisting}
\caption{Merging preconditions with implication.}
\end{figure}

The procedure is shown in Figure 4.5. Note that this merging will remove all equality and/or strict inequality preconditions composing the non-strict ones. For example, if two conditions \(p: (\arg == 0)\) and \(q: (\arg > 0)\) infer the condition \(t: (\arg =/= 0)\) and \(p\) is stronger and less-frequently checked than \(t\), as at line 7 in Figure 4.4, its call sites containing \(p\) will be added to those of \(t\). Then, \(p\) is removed.

Dealing with dynamic dispatch. Since the data types cannot be precisely resolved at static time, some actual API calls could be missed in our static analysis, thus, all their preconditions at those call sites could be missed too. For example, method \(\text{obj.add()}\) which is resolved at static time as \(\text{List.add()}\) because \(\text{obj}\) is declared as \(\text{List}\) could actually be \(\text{ArrayList.add()}\) at runtime. We address this with a conservative solution that whenever a set of preconditions is extracted for a call of API \(m\), that set is also considered as the preconditions of all APIs that override or implement \(m\) in the library. The rationale behind this is the assumption of behavioral subtyping in which preconditions cannot be strengthened in a subtype Liskov and Wing (1994). Thus, this heuristic will enrich the set of extracted preconditions for a sub-type with those from the super-type, which are the same or stronger than the actual ones. Those preconditions could be merged to the actual ones and increase the confidence of the actual ones.

4.3.4 Precondition Filtering

Since we mine preconditions from many projects/methods in a large-scale code corpus, there are conditions which are context-specific or might even be incorrect. These conditions are not useful for building the API specifications and should be filtered out. First, we remove all conditions which are
checked only once in the whole code corpus. Then, for each API, we remove all conditions which have low confidence in being checked before calling the API.

The confidence of a precondition for an API is measured as the ratio between the number of code locations checking the condition before calling the API over the total number of locations calling the API. We compute two values for confidence corresponding to two types of locations: one over client projects ($\text{conf}_{pr}$) and another over client methods ($\text{conf}_{m}$)

$$\text{conf}_{pr}(p) = \frac{|\Psi(p)|}{|\bigcup_q \Psi(q)|} \quad \text{conf}_{m}(p) = \frac{|\Omega(p)|}{|\bigcup_q \Omega(q)|}$$

where $\Psi(p)$ is the set of projects with condition $p$ before the API call. For each API, we keep only the preconditions that have both confidence values higher than or equal to a certain threshold $\sigma$. We use $\sigma = 0.5$ in our experiment.

### 4.3.5 Precondition Ranking

For each API, we rank the preconditions based on their total confidence, which is computed as $\text{conf}(p) = \text{conf}_{pr}(p) \times \text{conf}_{m}(p)$. Using only $\text{conf}_{m}(p)$ might favor the conditions used a lot but only in a small number of projects. In contrast, using only $\text{conf}_{pr}(p)$ might favor the conditions which are accidentally repeated in many projects but not used frequently. Thus, our approach combines both confidence values for ranking. Different from the traditional ranking scheme that puts all items in one list, our approach uses different ranked lists for the receiver object, the arguments of an API, and any combinations of them. Only the top-1 precondition in each ranked list is kept in the final result.

### 4.4 Empirical Evaluation

In this section, we aim to answer two research questions:

**RQ1.** How accurate are the preconditions mined by our approach? The answer to this question would tell whether our approach works in identifying the preconditions from usages in a large code corpus.

**RQ2.** How useful are the mined preconditions as a starting point in writing API specifications?

#### 4.4.1 Data Collection

We collected a large code corpus from two sources: SourceForge.net (SF) SourceForge and Apache Software Foundation (ASF) Apache. SF is a free source code hosting service for managing open source software projects. ASF is an American non-profit corporation who manages the development of Apache open source projects.
Table 4.3: Collected projects and API usages.

<table>
<thead>
<tr>
<th></th>
<th>SourceForge.net</th>
<th>Apache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Projects</td>
<td>3,413</td>
<td>146</td>
</tr>
<tr>
<td>Total source files</td>
<td>497,453</td>
<td>132,951</td>
</tr>
<tr>
<td>Total classes</td>
<td>600,274</td>
<td>173,120</td>
</tr>
<tr>
<td>Total methods</td>
<td>4,735,151</td>
<td>1,243,911</td>
</tr>
<tr>
<td>Total SLOCs</td>
<td>92,495,410</td>
<td>25,117,837</td>
</tr>
<tr>
<td>Total JDK public classes</td>
<td>1,275</td>
<td>1,275</td>
</tr>
<tr>
<td>Total JDK public methods</td>
<td>11,049</td>
<td>11,049</td>
</tr>
<tr>
<td>Total used JDK classes</td>
<td>806 (63%)</td>
<td>918 (72%)</td>
</tr>
<tr>
<td>Total used JDK methods</td>
<td>7,592 (63%)</td>
<td>6,109 (55%)</td>
</tr>
<tr>
<td>Total method calls</td>
<td>22,308,251</td>
<td>5,544,437</td>
</tr>
<tr>
<td>Total JDK method calls</td>
<td>5,588,487</td>
<td>1,271,210</td>
</tr>
</tbody>
</table>

For SF, we downloaded project metadata in JSON format from its website and collected information about all projects that are self-classified to be written in Java. To get higher quality code for mining the preconditions, we filtered out the projects that might be experimental or toy programs based on the number of revisions in the history. We only kept projects with at least 100 revisions. We downloaded the last snapshots of each project. We eliminated from the snapshot of a project the duplicated code from different branches/versions of the project. For ASF, we checked the list of all Apache projects Apache and downloaded the source code of the latest stable releases of all projects written in Java.

Table 4.3 shows the statistics on our datasets. SF has 3,413 projects satisfying the above criteria and ASF has 146 projects. They both have hundreds of thousand of source files. The total amount of code is almost 120 million lines of code (SLOCs) where SF contributes about four times more than ASF. The projects are written by thousands of developers and cover a variety of domains and topics.

In this experiment, we focus on the APIs in the JDK library. Analyzing all APIs from the java packages, we found that there are 1,275 public classes and 11,049 public methods in the library. We also observed that many APIs have not been used at all in the studied projects. Only 63% and 72% of the accessible JDK classes have been used in SF and Apache, respectively. The corresponding numbers of JDK methods used are 63% and 55%, respectively. In both SF and ASF, about one-fourth of the number of all method calls are the calls to JDK methods. This number shows that those open-source projects are heavily based on the JDK library.

4.4.2 Ground-truth: Java Modeling Language (JML) Preconditions

In order to evaluate the accuracy of our mined preconditions, we used a ground-truth of known-correct preconditions. The Java Modeling Language (JML) is a language for specifying the behavior of
Java classes and methods. Specifications are defined using a custom syntax inside of special comments that start with `@'. Figure 4.6 shows part of the specification in JML for the `substring(int, int)` method discussed in Section 4.2. The specification defines both normal behavior (lines 1–5) and exceptional behavior (lines 7–10), and signals certain Exception when certain preconditions hold (line 11).

The normal behavior for this method requires three conditions to hold prior to calling the method. These conditions are declared using `requires` statements and boolean expressions (lines 2–4). The specification also ensures that after finishing normal execution two conditions hold. These are declared using `ensures` statements and boolean expressions (line 5). A precondition is 1) a clause in the conjunctive normal form of the boolean expression following a `requires` keyword in a normal behavior, or 2) a clause in the conjunctive normal form of the negation of the boolean expression following a `requires` or `signals` keyword in an exceptional behavior. If a specification has multiple normal and/or exceptional behaviors, we combine them by taking the union set of the preconditions. For example, if preconditions `i > 0` and `i == 0` appear in two normal behaviors, they will be combined into a precondition `i >= 0`. The preconditions are then abstracted using the symbolic names.

The authors and maintainers of JML have written specifications for several popular Java packages from the JDK and published them on their website JML (2013). We downloaded and analyzed these specifications. As shown in Table 4.4, there are specification files for 62 classes from 6 JDK packages. After analyzing, we learned that, in 15 class files, there are no specifications for any method (column No Spec) and in 5 other files, there are specifications for some methods but not all (column Some Spec). We read the remaining 42 files, which contain specifications for all methods (column Full Spec), and extracted all preconditions for all of their methods.

Table 4.5 summarizes the number of extracted preconditions of the methods in those 42 classes. We group the methods based on the numbers of extracted predicates in the preconditions. In total, there
Table 4.4: Specifications for JDK classes from JML Website

<table>
<thead>
<tr>
<th>Package</th>
<th>Number of classes</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Full Spec</td>
<td>Some Spec</td>
<td>No Spec</td>
<td>Total</td>
<td></td>
</tr>
<tr>
<td>java.io</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>java.lang</td>
<td>14</td>
<td>3</td>
<td>1</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>java.net</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>java.sql</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>java.util</td>
<td>23</td>
<td>2</td>
<td>0</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>java.util.regex</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>All</td>
<td>42</td>
<td>5</td>
<td>15</td>
<td>62</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: JML preconditions of JDK methods in classes with full specifications

<table>
<thead>
<tr>
<th>Number of Preconditions</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods</td>
<td>78</td>
<td>465</td>
<td>144</td>
<td>62</td>
<td>36</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Number of methods: 797. Number of preconditions: 1155.

are 1,155 preconditions for 797 methods in which 78 of them have no preconditions, 465 of them have one precondition, and so on. As seen, most of them have from 0 to 3 preconditions. A much smaller percentage of methods has more than 3 preconditions.

4.4.3 RQ1: Accuracy

4.4.3.1 Result

We ran our tool on the two datasets, and compared the mined preconditions with those in the JML ground-truth. We used two metrics: precision and recall. Precision is measured as the ratio between the number of correctly-mined preconditions and the total number of mined preconditions. Recall is measured as the ratio between the number of correctly-mined preconditions and the total number of preconditions. A mined condition is considered correct if it is exactly matched with one precondition of the same method in the ground-truth using syntactic checking. If a mined condition is not in the ground-truth, we manually verified it. If it is a not-yet defined one, or semantically equivalent with a precondition (e.g., !rcv.isEmpty() and rcv.size() > 0) or implied by the preconditions of that method in the ground-truth (e.g., b > 0 is implied by a > 0 and a < b), it is counted as correct.

Table 4.6 shows the accuracy for all mined preconditions. In both datasets, the tool achieved high accuracy with recall from 75–80% and precision from 82–84%. The accuracy for two sources is comparable. The accuracy for SourceForge is a bit higher than that for Apache. When both datasets are combined, precision lies between those for two datasets. However, recall is slightly improved since a
few more API methods, which were not seen in either dataset, have been included in the result for the combined dataset.

Table 4.7 shows more detailed numbers on the mining accuracy for all the API methods. As seen, with the SourceForge dataset, our tool can cover all of the preconditions for 613 out of 797 (77%) JDK methods in the ground-truth. That is, in 77% of given methods under investigation, specification writers would just have to verify and remove some incorrect ones. Among those 613, we can derive perfectly the preconditions for 492 methods. That is, in 62% of methods, specification writers would use the set of preconditions as is. There are 118 (15%) and 3 (0.38%) methods having 1 and more than 1 extra (incorrect) preconditions, respectively. Our tool cannot produce any correct preconditions for only 8% of the methods. The numbers are comparable for Apache and the combined dataset.

Thanks to our light-weight analysis, the running time for Apache, which has more than 25M SLOCs and 1.2M JDK API calls, is just 34 minutes. The time for SourceForge and for both is much longer mainly due to accessing the local SVN repositories.

4.4.3.2 Analysis

Incorrect Cases. We first analyzed the incorrect cases. Since the JML specifications were manually built by the JML team, it is possible that some preconditions are still missing from the current version of their specifications. Thus, for the mined preconditions that are not in the ground-truth from the JML team, we manually verified them to see if they are truly incorrect cases. We found 5 correctly mined preconditions that were missing in the ground-truth (Table 4.8). We sent them to the main author of JML. He kindly confirmed all five cases. This is evidence that our tool could help specification writers reduce their effort and mistakes.
Table 4.8: Newly found preconditions in JML specifications

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>getChars(int,int,char[],int)</td>
<td>arg3 &gt;= 0</td>
</tr>
<tr>
<td>StringBuffer</td>
<td>append(char[])</td>
<td>arg0 != null</td>
</tr>
<tr>
<td>BitSet</td>
<td>flip(int, int)</td>
<td>arg0 &lt;= arg1</td>
</tr>
<tr>
<td></td>
<td>set(int, int)</td>
<td>arg0 &lt;= arg1</td>
</tr>
<tr>
<td></td>
<td>set(int, int, boolean)</td>
<td>arg0 &lt;= arg1</td>
</tr>
</tbody>
</table>

Table 4.9 shows the summary of the incorrectly-mined preconditions, which are classified into 3 types. For majority of the incorrect cases, the mined preconditions are **stronger** than the actual ones. The reason is that our tool cannot distinguish between the precondition as part of an API usage and the one as part of the API specification. For example, the API `java.util.List.add(Object)` accepts a null argument. However, in many usages of that API in the client code, developers often perform null checking for the argument before calling it. Thus, our tool reported the incorrect condition: `arg0 != null`. Another example is the API `File.mkdir()`, which does not require any preconditions in its specification. If the operation fails for some reason, it will return null. However, to avoid unnecessary operations to the file system and control the reason of the failure, developers often check file existence with `!exists()` before calling `mkdir()`. Another example is the method `valueOf(Object obj)`. Our tool detects the null checking on the argument `arg0 != null` from several client projects, but it is not part of its specification. These examples show an interesting gap between the actual API usages from client code and the intended usages from the API designers. This suggests a further investigation for API designers on how to adjust to support developers better in the APIs’ client code.

In the second type of incorrect cases, the conditions along the path to an API call are **irrelevant** to the preconditions of the API. For example, it is frequent that developers check if both arguments are positive before calling `Math.min()`. Those checks might make sense in term of the logic of the program, however, they are not relevant as the preconditions.

For the third type, a few incorrect cases are caused by the **imprecision in our light-weight static program analysis**. An example is incorrectly-mined precondition `arg0 <= 0` of `StringBuffer.-ensureCapacity(int)`. In the code, the call to this API belongs to the branch satisfying `arg0 <= 0`, however, the sign of `arg0` is reversed before the call. Our analysis did not keep track of the value change in the code leading to the call, thus, extracted incorrect condition. To track value changes, we can use dynamic symbolic execution.

**Missing Cases.** To better understand the missing cases, we examined all the preconditions which are in the ground-truth but were not mined by our tool. We classified the missing cases into four categories...
Table 4.9: Different types of incorrectly-mined preconditions

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Total</th>
<th>Stronger</th>
<th>Irrelevant</th>
<th>Analysis.Err.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourceForge</td>
<td>173</td>
<td>118</td>
<td>53</td>
<td>2</td>
</tr>
<tr>
<td>Apache</td>
<td>187</td>
<td>121</td>
<td>65</td>
<td>1</td>
</tr>
<tr>
<td>Both</td>
<td>195</td>
<td>129</td>
<td>66</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.10: Four Types of Missing Preconditions

<table>
<thead>
<tr>
<th></th>
<th>No-call</th>
<th>Private</th>
<th>No occur</th>
<th>Low freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>4%</td>
<td>4%</td>
<td>9%</td>
<td>3%</td>
</tr>
<tr>
<td>Apache</td>
<td>5%</td>
<td>5%</td>
<td>12%</td>
<td>3%</td>
</tr>
<tr>
<td>Both</td>
<td>2%</td>
<td>5%</td>
<td>10%</td>
<td>4%</td>
</tr>
</tbody>
</table>

as shown in Table 4.10. Each cell of the table shows the ratio between the number of missing cases in the corresponding category over the total number of preconditions in the ground-truth.

The first category (column 'No-call') consists of the preconditions of the API methods that have their JML specifications in the ground-truth, but have never been called in the client code in our datasets. For SourceForge, there are 46 such methods with 45 preconditions. For Apache, the corresponding numbers are 49 and 58. For those methods and preconditions, which contribute about 4% and 5% of the total numbers of preconditions, respectively, our tool can not mine the preconditions.

The second category (column Private) contains the preconditions involving the APIs’ private and internal fields or methods, which are inaccessible from client code. Examples of this category are

1. Precondition !changed of Observable.notifyObservers(): changed is a private field of the Observable class to represent the internal state of the object. The method notifyObservers is called only if the object’s state was changed.

2. Precondition parseable(s) of Integer.parseInt(String s): this condition requires the string argument of parseInt to be parseable.

3. Precondition capacityIncrement >= 0 of Stack.push(Object): The stack can only be pushed if its internal capacity is larger than 0.

The first two categories are due to the inherent limitation of mining approaches on client code, however, their percentages are small.

The last two categories contain the preconditions which could occur in the client code but are not in our result due to the limitations of our static analysis that cannot detect the occurrences of the conditions (No occur.) or due to the cut-off thresholds (Low freq.).
4.4.3.3 Accuracy by data size

When computing accuracy, we also analyzed the impact of the size of dataset in our algorithm. We ran our tool on various data sizes. From each full dataset, SourceForge and Apache, we created the datasets of size $B$ by randomly selecting the projects of the full dataset into bins having the same number of $B$ projects. Using each bin as input, we ran our tool on it and recorded the accuracy (precision and recall) for that bin. Then, we computed the average accuracy over all bins, and used that accuracy for that size. In this experiment, we chose $B = 2^i$, meaning that we kept increasing the data sizes by a power of 2 until reaching the full dataset. To consider both precision and recall, we used Fscore. Fscore is the harmonic mean of precision and recall, which is computed as

$$Fscore = \frac{2 \times Precision \times Recall}{(Precision + Recall)}.$$  

Figure 4.7 shows the result. The values on the lines at $B = 1$ shows the accuracy for the dataset containing individual projects and those at $B = Full$ shows the accuracy for the full dataset as input. As the data size increases, precision decreases and recall increases. The gain in recall is much higher than the loss in precision making their harmonic mean Fscore increases significantly: 7% to 82% for SourceForge and 21% to 79% for Apache.

4.4.3.4 Accuracy Sensitivity Analysis

In this experiment, we studied the impact of different components in our method on the accuracy. In Figure 4.8, the baseline (group Base) is the solution that extracts the preconditions by looking at only the guard conditions (e.g., the ones in if statement(s)) on the path leading to the API calls. This baseline does not consider the properties of the passed arguments, normalization and merging, nor deal with dynamic dispatch. Then, we successively add other components one by one to the baseline solution.
Figure 4.8: Mining accuracy as technical components are successively added to see changes in accuracy. The second solution (Arg) adds the preconditions that are obtained from the properties of the passed arguments, e.g., \( \text{arg0} == 0, \text{arg1} != \text{null} \). The third one includes normalization of preconditions and the fourth one includes merging. The last one covers all components in our approach by adding the subtyping information in which the preconditions of a method are also collected from the call sites of its overridden methods to deal with dynamic dispatch.

As more components are added, recall increases significantly from 60 to 79% in SourceForge and from 55 to 75% in Apache, while precision is maintained. Among the components, adding properties of arguments passed to APIs improves the recall 6% in SourceForge and 8% in Apache. The respective improvements from adding merging conditions are 7% and 5%. Adding subtyping contributes 4% and 6%. Normalization contributes 2% for both.

4.4.4 RQ2: Usefulness

We also studied how useful our automatically mined preconditions are for writing specifications via two experiments.

4.4.4.1 Suggesting preconditions in specifications

Our first experiment looks at the mined preconditions for API methods that currently do not have a JML specification provided. We run our tool to automatically mine preconditions for the APIs and then manually transformed them into JML syntax. We then sent these JML-styled specifications to one of the original authors of JML. If he agreed these specifications are correct, it lends evidence that our approach is useful as a tool for suggesting preconditions when writing the initial specification for APIs.

Our results are summarized in Table 4.11. In total, we prepared specifications for 11 API methods from 2 JDK classes which previously had no JML specifications. Our tool generated a total of 29
mined preconditions (column $M$). For our approach, one author transformed the automatically generated preconditions into JML specifications. A second author, who has extensive experience with JML’s syntax including designing and implementing the JML research compiler JAJML, then performed a manual validation of the results and removed 4 preconditions (column $Rm$) which are incorrect for the corresponding APIs. Five preconditions are deemed close (column $Fix$), but require modifications of the comparison operator from strictly greater than ($>$) to greater than or equal to ($\geq$). The remaining 20 were accepted exactly as the tool mined them. After this step, the specifications containing 25 preconditions (including the 5 modified) were sent to the JML team member.

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>$M$</th>
<th>$Rm$</th>
<th>$Fix$</th>
<th>Accept</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringBuffer</td>
<td>delete(int,int)</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>replace(int,int,String)</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>$Y^*$</td>
</tr>
<tr>
<td></td>
<td>setLength(int)</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>subSequence(int,int)</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>substring(int,int)</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td>LinkedList</td>
<td>add(int,Object)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>addAll(int,Collection)</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>get(int)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>listIterator(int)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>remove(int)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>set(int,Object)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>29</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

As seen, the JML team member agreed on 10 out of 11 methods’s specifications, such that the set of suggested preconditions is complete and precise ($Y$ in column $Accept$). For only one method StringBuffer.replace ($Y^*$ in column $Accept$), the preconditions are correct however two other ones are missing.

### 4.4.4.2 Web-based survey

In the second experiment, we created a web-based survey and asked human subjects who have experience with using JDK library and/or formal specification languages such as JML to evaluate the resulting preconditions. We had a total of 15 respondents. Participants were asked to rate their experience with Java, JML, reading specifications, and writing specifications. Two thirds self-indicated having more than 6 months experience writing specifications and many with experience in JML specifically.

Participants were shown an example method (e.g., the substring method from Section 4.2) along with the set of proposed preconditions we mined for that method. We then pre-selected the correct answers (based on the JML ground-truth) for each condition and explained why it was “correct”, “a
good starting point”, or “incorrect”. “Correct” means that this precondition can be used as-is in the specification. “Good starting point” means that it might need small modifications to be used in a specification, such as changing a comparison operator from strict to non-strict. “Incorrect” means that the condition is irrelevant in building the specification.

Next, users were shown 5 methods one at a time and the mined preconditions for them. They were asked to rate each individual precondition as mentioned. We also asked them to give an overall, more subjective, rating for the entire method on whether our mined preconditions are useful. After 5 methods, they were given an opportunity to write general feedback. They also had an opportunity to continue rating more preconditions for other methods. On average each participant graded 20 preconditions.

When randomly choosing methods for a user, we enforced that the first two were APIs that existed in the ground-truth and the last three were APIs that did not. Using the responses from the first two, we were able to grade the users on their expertise by calculating the answers that matched the ground-truth out of the total number of ground-truth answers. For this study, we only keep responses from users who scored 100% on this grading. In total, there were 9 users grading 75 methods with 104 preconditions.

The following table shows the correctness of the preconditions as rated by participants. Excluding the ‘Not Sure’ responses, the participants rated 63% as Correct. What the results in Section 4.4.3 could not show however was the amount of almost correct preconditions, which the participants rated as almost 19%.

<table>
<thead>
<tr>
<th>Correct</th>
<th>Good Starting Point</th>
<th>Incorrect</th>
<th>Not Sure</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>19</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>63%</td>
<td>19%</td>
<td>18%</td>
<td>–</td>
</tr>
</tbody>
</table>

Overall, participants found that 82% of the mined preconditions are useful as the starting point for writing the specification. The following table shows the responses for rating the tool’s usefulness:

<table>
<thead>
<tr>
<th>Agree+</th>
<th>Agree</th>
<th>No Opinion</th>
<th>Disagree</th>
<th>Disagree+</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>33</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>33%</td>
<td>48%</td>
<td>–</td>
<td>13%</td>
<td>6%</td>
</tr>
</tbody>
</table>

Again, excluding the ‘No Opinion’ responses, the participants rated the tool as useful for 81% of the methods shown!

### 4.4.4.3 A benchmark of precondition-related bugs

In this section, we show an application of our mined preconditions in building a benchmark of bugs caused by missing precondition checking. An example of this type of bug is that a developer does not
check the condition \( \text{beginIndex} \leq \text{endIndex} \) before calling \text{String.substring}(\text{int, int}) \) when the logic of the program does not ensure it. This type of benchmark is very useful for bug detection tools that look for neglected condition checking such as Chang et al.’s tool Chang et al. (2008) and AlattinThummalapenta and Xie (2009). It was reported that neglected conditions are an important but difficult-to-find class of defects Chang et al. (2008).

To build the benchmark, we processed all 1,966,563 revisions with changed Java files for all 3,413 Java projects in SourceForge dataset. For a project \( P \), we first identified the fixing revisions by the popular method Zimmermann and Weißgerber (2004) that uses the heuristic of searching in commit logs for the phrases indicating fixing activities. For each fixing revision \( r_i \), we used our prior origin analysis tool to compare it with the previous one \( r_{i-1} \). We detected the mapped methods and API calls between two revisions. For each pair of mapped API calls in a method, we computed two sets of guard conditions. We compared each set with the mined preconditions of the API to find the set of preconditions that are implied by a guard condition. If there exists such a precondition in \( r_i \) but not in \( r_{i-1} \), we add the API call sites and \((r_{i-1}, r_i)\) to our benchmark. In total, there are 369,532 fixing revisions. Among them, 3,130 (0.85\%) in 931 projects are detected as related to missing preconditions. The total number of call sites related to those fixes is 4,399. To check its quality, we manually checked a sample of 100 call sites in the benchmark, and found that 80 of them are related to preconditions. We will manually check all and make our benchmark available. We found that null-pointer and index-out-of-bounds exceptions are the two most common sub-types in those bugs. Our result confirms this type of bug and calls for detection tools. This shows the usefulness of our mined preconditions in building the benchmark. Our mined preconditions can also be used in such detection tools.

4.4.5 Threats to Validity.

The two chosen datasets might not be representative. The criteria of 100 revisions might not have filtered out all experimental and toy projects. We conducted experiments only on JDK. The ground-truth was built by us. Thus, human errors could occur. The two chosen classes in the usefulness study might not be representative. Our human study suffers from selection bias, as not all participants have the same level of expertise on formal specifications. There is possible construct bias as we chose the APIs in JDK. We did not compare our tool to a related one in Ramanathan et al. (2007b). Similar to ours, their tool is also based on both mining and program analysis. However, their tool is for C code and re-implementing it for Java code is infeasible due to their algorithm’s complexity as well as the differences between two languages. Moreover, their approach operates on a single project while we rely on large number of projects. Thus, the two approaches require inputs with different nature.
Other mining-based approaches do not work for preconditions, while other static and dynamic analysis methods for specification inference do not have a mining component (Section 4.5).

4.5 Related Work

The condition mining work that is closest to our approach is from Ramanathan, Grama, and Jagannathan (RGJ) Ramanathan et al. (2007b). Similar to ours, the RGJ approach tightly integrates program analysis with data mining techniques. They proposed a static inference mechanism to identify the preconditions that must hold when a method is called. They first analyze the call sites of the method in its containing program and then use a path-sensitive inter-procedural static analysis to collect the predicates at each program point. To compute preconditions, RGJ collects a predicate set along each distinct path to each call-site. The intersection of predicate sets is then constructed at the join points where distinct paths merge. Predicates computed within a procedure are memorized and used to compute preconditions that capture inter-procedural control- and data-flow information. RGJ then runs frequent itemset mining on data-flow predicate sets, and sub-sequence mining for control-flow conditions to derive preconditions. They reported a precision level of 77.13%.

Our approach has several key differences. First, it operates on a very large-scale corpus of client programs of the libraries that contain the call sites of APIs. In contrast, RGJ is designed to perform its inter-procedural analysis on only an individual client program containing the APIs’ call sites. Thus, RGJ can be used to improve our analysis technique when running on each project. Second, their mining algorithm works on the data-flow predicate sets in an individual program, while our mining technique operates on the comparable preconditions across an ultra-large number of projects. In contrast, they find conditions using sophisticated data- and control-flow analyses on a single program. Their mining algorithm does not consider the predicates across projects.

Our work is also related to static approaches for mining specifications. Those static approaches rely more on data mining, while using more light-weight static analyses than our approach and RGJ. Gruska et al. Gruska et al. (2010) introduce the idea of wisdom of the crowds similar to our approach on 6,000 Linux projects (about 200MLOCs). However, their technique mines only temporal properties in term of pairs of method calls. They used 16 million mined temporal properties to check the anomalies in a new project. Our prior work, GrouMiner Nguyen et al. (2009c) performs frequent subgraph mining to find API programming patterns. JADET Wasylikowski et al. (2007), Dynamine Livshits and Zimmermann (2005), Williams and Hollingsworth Williams and Hollingsworth (2005), CodeWeb Michail (2000) mine pairs of calls as patterns. MAPO Zhong et al. (2009a); Acharya et al. (2007) expresses API

Other static approaches to mine API specifications and then leverage them to detect bugs Engler et al. (2001); Kremenek et al. (2006b); Li et al. (2006); Li and Zhou (2005); Liu et al. (2006). FindBugs Hovemeyer and Pugh (2004) looks for specified bug patterns. Tools suggest code examples related to specific APIs and types Mandelin et al. (2005b); Sahavechaphan and Claypool (2006); Thummalapenta and Xie (2007); Xie and Pei (2006). All above static approaches do not recover APIs’ preconditions.

There are several dynamic approaches in mining specifications Ammons et al. (2002); Cousot et al. (2011); Dallmeier et al. (2005); Ernst et al. (1999); Gabel and Su (2008); Liu et al. (2006); Lo and Maoz (2009); Pradel and Gross (2009); Wei et al. (2011); Yang et al. (2006). Daikon Ernst et al. (1999) automatically detects invariants in a program via running test cases. Wei et al. Wei et al. (2011) infer complex post-conditions from simple programmer-written contracts in the code. Weimer et al. Weimer and Necula (2005) mine method pairs from exception control paths and identify temporal safety rules. In brief, our approach can complement well to dynamic approaches.

There are other approaches that require annotations on partial specifications on desired invariants, and then verify program properties and detect violations Alur et al. (2005); Fischer et al. (2005); Henzinger et al. (2005). Our approach is automatic.

Our work is also related to research to derive the behavior model of a program or software component for verification de Caso et al. (2012); Lo et al. (2009); Lorenzoli et al. (2008). These approaches aim to recover the formal model for a program with pre/post-conditions of the states’ transitions. In contrast, our approach focuses at a more fine-grained level of individual APIs.

### 4.6 Conclusions

In this chapter, we propose a novel approach to mine the preconditions of API methods using a large code corpus. Our key idea is that the true API preconditions appear frequently in their usages from a large code corpus with large number of API usages, while project-specific conditions occur less frequently. We mined the preconditions for JDK methods on almost 120 million SLOCs on SourceForge and Apache projects. Comparing to the human-written preconditions in JML, our approach achieves high accuracy with recall from 75–80% and precision from 82–84% for the top-ranked results. In our user study, participants found 82% of the mined preconditions as a good starting point for writing specifications.
CHAPTER 5. CHANGE REPETITIVENESS ANALYSIS

In this chapter, we present a large-scale study of repetitiveness of code changes in software evolution. We collected a large data set of 5,682 Java projects, with 1.9 billion source lines of code (SLOC) at the latest revisions, 3.4 million code change revisions (0.9 million fixes), 12.9 million changed files, and 4.8 billion changed SLOCs. A change is considered repeated within or cross-project if it matches another change having occurred in the history of the project or another project, respectively. We developed techniques, OAT and Treedit, for detecting code changes at both coarse-grained and fine-grained levels. From the results of our study, we report the following important findings. First, repetitiveness of changes could be as high as 60–100% at small sizes and decreases exponentially as size increases. Second, repetitiveness is higher and more stable in the cross-project setting than in the within-project one. Third, fixing changes repeat similarly to general changes.

5.1 Introduction

In software development, software reuse is a pragmatic approach that engineers often follow to save development efforts. Software reuse could occur at different levels of abstraction. Multiple software projects could share common specifications, designs, or algorithms. Engineers may reuse the same libraries and frameworks, resulting in patterns or common programming idioms in source code. Common programming tasks expressed in programming languages may lead to similarity in source code. Such similar code may lead to similar changes and repeated defects and fixes within or across multiple projects.

Exploring that phenomenon, several mining software repositories (MSR) approaches have made advances in its applications to automate several software evolution and maintenance tasks. An example of such application is automatic program repairing Goues et al. (2012); Kim et al. (2013) based on previously seen fixing patterns in the same or different projects. PAR Kim et al. (2013) is an automatic pattern-based program repair method that learns common patterns from prior human-written patches. FixWizard Nguyen et al. (2010b) recommends fixes based on code clones and code with similar API usages. GenProg Goues et al. (2012) is a patch generation method that is based on genetic programming.
Other types of application are *automated library update, language/library migration*, etc. SemDiff Dagenais and Robillard (2008) is a method to learn from previous updating changes to a framework in order to update its client code. LibSync Nguyen et al. (2010a) learns adaptation change patterns from client code to update a given program to use the new library version. MAM Zhong et al. (2010) is an approach to mine common code transformations to support language migration.

While those approaches have gained much success in MSR, they focus on respective application domains and are often studied on small-scale settings with small sets of subject projects. There is no existing large-scale, systematic study on how repetitive software changes are across the histories of software projects, what are the repetitiveness characteristics of software changes, or whether fixing changes exhibit different levels of repetitiveness than general ones. To address them, we conducted a large-scale study with the following key research question: how code changes repeat in software evolution. The answer for this question not only provides the empirical evidences but also could enhances those aforementioned MSR approaches. For example, a genetic programming-based automatic program repair could avoid unnecessary mutations by considering the information on the popular types and sizes of program elements that have been used in fixes for certain program contexts, thus, reducing their search space for possible fixes. Language migration or library update methods could benefit in similar manners when the repetitive characteristics of changes are considered.

In our study, we collected from SourceForge and GitHub a large data set of 5,682 Java projects, with 1.9 billion source lines of code (SLOC) at the latest revisions, 3.4 million code change revisions (0.9 million fixing ones), 12.9 million changed files, and 4.8 billion changed SLOCs. We extracted consecutive revisions and compared the abstract syntax trees (ASTs). A change is modeled as a pair of subtrees \((s,t)\) in the ASTs. A change \((s,t)\) is considered as matching with another change \((s',t')\) if \(s\) and \(s'\), and \(t\) and \(t'\) structurally match when abstracting on the literal and local variables’ nodes. The size of a change \((s,t)\) is measured as the height of the sub-tree \(s\) in the source AST. The change type is defined as the AST node type of \(s\). We perform the analysis in two settings: within and cross-project. In the within-project setting, a change in a project is considered as repeated if it matches another change previously occurred in the project’s history. In the cross-project setting, it is considered as repeated if it matches another change occurring in another project. Change repetitiveness is computed via the number of repeated changes over the total number of changes. We studied repetitiveness in three dimensions: size, type, and general/fixing changes. Our key findings include the following:

1. Repetitiveness is very high for changes of small sizes, e.g., up to 60–100% for the changes of sizes 1 and 2. Changes of size 1 are on literals, identifiers, etc. However, it decreases exponentially as size
increases. Repetitiveness of changes with sizes larger than 6 is small. Thus, the aforementioned automatic tools should consider change fragments of sizes from 2–6.

2. Repetitiveness also varies by syntactic types of changes. Changes involving simple structures (e.g., array accesses, method calls) are highly repetitive, while those with compound structure (e.g., control/loop statements) are less. In addition, the most popular types of fixing changes include method calls, infix expressions, conditions (e.g., if) and loop statements (e.g., for, enhance for). Thus, automatic program repair tools could focus on those types with small sizes in the search space and then combine them.

3. Cross-project repetitiveness is generally higher and more stable than within-project one. While cross-project repetitiveness of fixing changes is as high as that of general changes and even higher in small change sizes, within-project repetitiveness of fixing changes is low. This implies that program repair tools should not rely solely on the changes in a single project, but rather make use of repeated bug fixes across projects. Importantly, despite large project-specific jargons, after concrete names are abstracted, there is high structural similarity among changes across projects.

Section 5.2 introduces our research question and methodology. Section 5.3 describes the representations for code and code changes. Section 5.4 and Section 5.5 present techniques for extracting code changes, building change database and computing repetitiveness. Section 5.6 presents the results and our analysis. Section 5.7 is for the related work. Section 5.8 concludes the chapter.

5.2 Research Question and Methodology

This section will state the research questions in this study, and explain the process to collect the data and overview of how to extract and build the database of code changes.

5.2.1 Research Question

We are interested in studying the popularity of repeated code changes and fixes. Therefore, our research question is: How repetitive code changes and bug fixes are in software evolution?

We are interested in repeated code changes in different dimensions. First, we want to know how large they are (i.e., size of change) and what kind of program constructs that they often occur on (i.e., type of change). Such information will help designers of development tools use repeated changes to focus more on the sizes and types of changes that most likely repeat. In addition, whether changes repeat within- or across projects is also important. If they repeat frequently within a project, historical changes/fixes
of the project will be a useful source to predict and recommend future changes/fixes of that project. If they repeat frequently in the cross-project setting, we can learn changes/fixes from other projects to use for a project, especially when it is newly developed. Lastly, we want to study whether the repetitiveness of fixes, an important type of changes, is different from that of general changes.

5.2.2 Data Collection

To answer the question, we conducted an empirical study on a large dataset of code changes. We collected data from SourceForge SourceForge and Github GitHub, two hosting services for open-source projects. We downloaded from all repositories the Java projects using SVN on SourceForge and those using Git on GitHub. Those repositories were stored and processed locally on our machine. We filtered out the projects with very short histories (having less than 100 revisions).

Table 5.1 summaries our final dataset. SourceForge has 2,841 projects meeting our criteria. Those projects have in total 16 million Java source files and 1.7 billion source lines of code (SLOC) in their last snapshots. For GitHub data, we filtered out not only the projects with short histories but also the ones that are forked from some other projects. The remaining projects contains more than 20 thousands projects. To make it balanced with the data from SourceForge, we sampled from them 2,841 projects, which is the same as the number of projects from SourceForge. These projects contain 1 million Java source files and 0.2 million SLOCs in their last snapshots. The amount of code in the last snapshots in SourceForge projects is much larger than that in GitHub projects because SourceForge projects contain more releases in the repositories than GitHub projects do. In total, we have 5,682 projects with 17 million Java source files and 1.9 billion SLOCs. Those projects cover variety of domains and topics, and
have been written by thousands of developers. We downloaded their repositories to our local machine for faster processing.

In terms of changes, the numbers are quite comparable between the two data sources. In total, the projects in our dataset have 6.4 million revisions. Among them, 3.4 million revisions have code changes and 0.9 millions have fixing changes. To detect fixing changes, we used the popular keyword-based approaches Zimmermann et al. (2007), in which if the commit log message of a revision has the keywords indicating fixing activities, the code changes in that revision are considered as fixing changes.

We processed all 6.4 million revisions and parsed in total 12.9 million changed source files with the total size of 4.8 billion SLOCs. Our change detection algorithm (Section 5.4) detected 16.3 million changed methods with the total size of 2.5 billion AST nodes. From those methods, it detected 419 million fine-grained code changes made from 173 million changed AST nodes.

5.2.3 Experimental Methodology

From the collected dataset, we extracted code changes/fixes to build our change database, search for repeated ones, and compute their repetitiveness. This process on each revision consists of three steps:

1. Detecting code changes for each revision: focusing on fine-grained changes, we collect only changes within the bodies of individual changed methods.

2. Updating detected changes to our database: the database is globally accessed for all projects to improve the performance in the study of cross-project repeated changes.

3. Computing the repetitiveness for all changes in both within- and cross-project settings for different dimensions: size, type, and fixing/non-fixing.

In the next sections, we will explain in detail how we represent code and code changes, and each step in the process.

5.3 Code Change Representation

5.3.1 Illustration Example

Let us start with an illustration example on code change and repetitiveness. Figure 5.1 shows two changes on two if statements. They are considered as fine-grained changes because they occur within individual methods. Both of them include a replacement of a literal (1 or 10) by a variable (b or y) and an addition of an else branch. The variables and literals in the pairs a and x, b and y, 1 and 10 have the
same roles. That is, if we replace $a$, $b$, and 1 with $x$, $y$, and 10 respectively, we can derive the second change from the first. Therefore, we consider the second change a repeated change of the first one (and vice versa).

We aim to study the characteristics of such repeated changes, e.g., how often they occur, how large they are, what are the popular types, etc. Next, we will formally define important concepts in our study.

### 5.3.2 Representation

As writing and modifying code, developers would think of code in terms of program constructs such as functions, statements, or expressions rather than lines of code or sequences of lexical tokens. For example, in Figure 5.1, one would think of the code (before change) as an *if statement*, and modify it by replacing an *operand* in an *infix expression* by another, and adding an *else branch*.

To address this phenomenon, in this study, we model source code and code change in terms of program constructs rather than the lower levels of representation such as code tokens or lines of code. In a programming language, a program construct is often defined as a syntactic unit and represented as a subtree in an Abstract Syntax Tree (AST). For example, an if statement is represented as a subtree in an AST, in which the root node specifies its type (i.e., *if statement*), and the children nodes represent its sub-constructs, i.e., an expression for the predicate, and two code blocks for two branches.
**Definition 5.1 (Code Fragment)** A code fragment in a source file is a (syntactically correct) program construct and is represented as a subtree in the abstract syntax tree (AST) of the file.

We consider a code change as a replacement of a code fragment by a different code fragment. Since a code fragment is modeled via an AST, we formulate code change as follows:

**Definition 5.2 (Code Change)** A code change is represented as a pair of ASTs \((s, t)\) where \(s\) and \(t\) are not label-isomorphic.

In this definition, \(s\) and \(t\) are called *source* and *target* trees, respectively. Either of them (but not both) could be a null tree. \(s\) or \(t\) is a null tree when the change is an addition or deletion of code, respectively. Since AST are labeled trees, the condition of *not being label-isomorphic* is needed to specify that the code fragments before and after change are different.

To check two code changes for repetitiveness, we could match their source and target trees. However, as seen in the illustration example, repeated changes might have different variable names or literal values. Therefore, we need to perform normalization to remove those differences before matching. An AST tree \(t\) is normalized by re-labeling the nodes for local variables and literals. For a node of a local variable, its new label is the node type (i.e., \(\text{ID}\)) concatenated with the name for that variable via alpha-renaming. For a literal node, its new label is the node type (i.e., \(\text{LIT}\)) concatenated with its data type.

Figure 5.2 shows the AST's subtrees for the code changes in the illustration example after normalization. The nodes for variables \(a\) and \(x\) are re-labeled as \(\text{ID } v1\) while the ones for \(b\) and \(y\) have the label of \(\text{ID } v2\), since \(v1\) and \(v2\) are their respective names after alpha renaming. The nodes for literals \(1\) and \(10\) have the same label \(\text{LIT } \text{NUM}\). Thus, after normalization, two changes have the same tree-based representation. We define repeated code changes as follows:

**Definition 5.3 (Repeat Code Change)** A code change \((s, t)\) is a repeated change of another one \((s', t')\) when \(s'\) and \(s\), and \(t'\) and \(t\) are label-isomorphic after normalization.

We want to study the repetitiveness of changes in a project in both scenarios: within its history, and across different projects. Therefore, we define:

**Definition 5.4** A change in a project \(P\) is a repeated change *within a project* if it is a repeated change of another one occurring in an earlier revision of \(P\). It is a *cross-project repeated change* if it is a repeated change of another in other project(s).

Since we want to study the repetitiveness of code changes on types and sizes, we need to define them. We use the AST type of the source tree as the type of the change. The size of a tree can be defined
as the number of its nodes. However, for source code, the number of nodes of ASTs highly vary. For
example, a method call might have only one child (e.g., no parameter) or many children (e.g., many
parameters). In our experiment, some trees might have thousands of nodes. In contrast, tree height
(i.e., the number of nodes along the longest path from the root to a leaf node) varies less (often from 1
to 10). Thus, we choose tree height as a measurement of change size.

**Definition 5.5 (Change Type and Size)** Type and size of a code change \((s, t)\) are AST type and
the height of \(s\) (or of \(t\) if \(s\) is a null tree), respectively.

In the illustration example, two code changes have the type of if and size of 4.

### 5.4 Code Change Detection

To detect code changes, we develop two program differencing techniques for coarse-grained code
elements, i.e. packages, files/classes and methods, and fine-grained code elements within the method
bodies, i.e. statements and expressions.

#### 5.4.1 Coarse-grained Differencing

This section discusses the origin analysis technique (OAT) to map corresponding coarse-grained code
elements (packages, classes, and methods) between two program versions. OAT views a program \(P\) as
a project tree \(T(P)\), where each node represents a package, class, interface, or method. Each node has
the following set of attributes:

- **Declaration** (\(\text{declare}(u)\)): For a package node, it is a fully qualified name. For a class node, it
  is a simple name followed by the names of the classes and interfaces that the node extends or
  implements. For a method node, it is a simple name, a list of parameter types, all modifiers, a set
  of exceptions, a return type, and associated annotations such as deprecated.

- **Parent** (\(\text{parent}(u)\)): It refers a node’s container code element.

- **Content** (\(\text{content}(u)\)): It represents a set of descendant nodes. For a method node, it represents
  the body of the method. When the source code is available, it represents the abstract syntax tree
  of the method body. Otherwise, it represents a sequence of byte code instructions extracted from
  a jar file.

Section 5.4.1.1 describes the types of transformations that OAT supports, Section 5.4.1.2 describes
similarity measures that were used to derive one-to-one mapping between tree nodes, and Section 5.4.1.3
describes our tree alignment algorithm that maps nodes between two trees using a set of similarity measures and derives transformations from the alignment result.

5.4.1.1 Transformation Types

Suppose that two versions \(P_i\) and \(P_j\) of a program \(P\) are represented as two trees \(T(P_i)\) and \(T(P_j)\). The changes between two versions are represented as the following types of transformations from one tree to another tree: add\((u)\), delete\((u)\), move\((u)\) — changes to the location of node \(u\), and update\((u)\) — changes to \(u\)'s attributes. An updated node could be moved as well. For a class, an update can be performed on its name, its superclass, or its interfaces. For a method, the update can be renaming, change to its input signature, change to a return type, change to visibility modifiers, etc. The above types of transformations are derived from an alignment result by considering unmapped nodes as added or deleted, and mapped nodes as moved or updated.

5.4.1.2 Similarity Measures

The similarity score between two nodes is computed by summing up their declaration attribute similarity, \(s_d\), and their content attribute similarity \(s_c\). OAT defines \(s_d\) and \(s_c\) for each type of nodes differently.

Method Level Similarity. \(s_d\) is computed based on weighted sum of textual similarities of return types, method names, and a list of parameter types:

\[
s_d(u, u') = 0.25 \times \text{strSim}(\text{return}, \text{return}') + 0.5 \times \text{seqSim}(\text{name}, \text{name}') + 0.25 \times \text{seqSim}(\text{paras}, \text{paras}')
\]

in which seqSim computes the similarity between two word sequences using the longest common subsequence algorithm Hunt and Mcilroy (1976), and strSim computes a token-level similarity between two strings by borrowing the idea from our prior work (Kim et al.’s text-similarity measures used for API matching between 2 versions Kim et al. (2007)).

For example, given the two methods, \text{getNewServiceObject(Context, String)} and \text{makeNewServiceObject(SOAP,Context,String)}, \(s_d\) is 0.875.

\[
s_d(u, u') = 0.25 \times \text{strSim}(\text{Object}, \text{Object}) + \\
0.5 \times \text{seqSim}(\text{getNewServiceObject, makeNewServiceObject}) + \\
0.25 \times \text{seqSim}([\text{Context, String}], [\text{Context, String}])
\]

\[
= 0.25 \times (1/1) + 0.5 \times (3/4) + 0.25 \times (2/2) = 0.875
\]

If the content is represented as an AST, \(s_c\) is computed by extracting a characteristic vector \(v(u)\) from a method \(u\) using our prior work Exas Nguyen et al. (2009a). If the content consists of bytecode
function MaxMatch(C, C', sim) // find maximum weighted matching
L = ∅
for (u, u') ∈ C × C'
    if (sim(u, u') ≥ δ)
        L = L ∪ (u, u')
sortDescendingly(L)
while (L ≠ ∅)
    (u, u') = L.top()
    M = M ∪ (u, u')
    for (v, v') : (v, u') ∈ L ∨ (u, v') ∈ L
        L.remove((v, v'))

Figure 5.3: Greedy Matching Algorithm

instructions, its characteristic vector v(u) is an occurrence-counting vector of all the opcodes. Then, the similarity between two methods u and u' is computed using the following formula:

\[ s_c(u, u') = \frac{2 \times |\text{Common}(v(u), v(u'))|_1}{||v(u)||_1 + ||v(u')||_1} \]

in which v(u) a vector representation of the method content, and Common(V, V') is defined as Common\((u, v)[i] = \text{min}(u[i], v[i])\). This formula means to take the ratio of the common part over their average size to measure the similarity.

Class and Package Level Similarity. The declaration similarity \(s_d\) is defined similarly to the that of methods. The content similarity \(s_c\) is computed via the number of their mapped children.

\[ s_c(C, C') = \frac{2 \times |\text{MaxMatch(content(C), content(C'), sim)}|}{|\text{content(C)}| + |\text{content(C')}|} \]

The MaxMatch function (Figure 5.3) takes two sets of entities C and C' and returns a set of pairs such that sim(u, u') is greater than a chosen threshold and there exists no u'' such that sim(u, u'') > sim(u, u').

5.4.1.3 Mapping Algorithm

OAT takes two project trees as input, aligns them, and computes transformations from one tree to another. It maps nodes in a top-down order, mapping parent nodes before their children. When method m is class C’s child and C’ is mapped to C, we first try to map m to C’’s children. If a match is not found, we assume that m is moved to another class. We adopted this strategy from UMLDiff Xing and Stroulia (2005) to reduce the number of candidate matches that need to be examined.

Figure 5.4 summarizes our origin analysis algorithm. At any time, each node is placed in one of three sets: (1) AM if it is already mapped to another node, (2) PM if its parent node is mapped but it is not mapped to any node, and (3) UM if the node and its parent are not mapped. OAT maps nodes in
function Map(T, T') // find mapped nodes and change operations
UM.addAll(T, T')
for packages p ∈ T, p' ∈ T' // map on exact location
  if location of u and u' is identical then Map(p, p')
for packages p ∈ T ∩ UM, p' ∈ T' ∩ UM // unmapped packages
  if Sim(p, p') ≥ δ then SetMap(p, p') // map on similarity
for each mapped pairs of packages (p, p') ∈ M
  MapSets(Children(p), Children(p')) // map parent–mapped classes
for classes C ∈ T ∩ UM, C' ∈ T' ∩ UM // unmapped classes
  if (C and C' are in a text–based/LSH–based filtered subset and sim(C, C') ≥ δ then SetMap(C, C') // map on similarity
for each mapped pairs of classes (C, C') ∈ M
  MapSets(Children(C), Children(C')) // map parent–mapped methods
for methods m ∈ T ∩ UM, m' ∈ T' ∩ UM // unmapped methods
  if (m and m' are in a text–based or LSH–based filtered subset and sim(m, m') ≥ δ and dsim(m, m') ≥ µ then
    SetMap(m, m') // map on similarity
Op = ChangeOperation(M)
return M, Op

function SetMap(u, u') // map two nodes
M.add((u, u'))
UM.remove(u, u')
PM.add(content(u), content(u'))

function MapSets(S, S') // map two sets of nodes
M2 = MaxMatch(S, S', sim) // use greedy matching
for each (u, u') ∈ M2
  SetMap(u, u')

Figure 5.4: Tree-based Origin Analysis Algorithm

UM first and the children of mapped ones are put in PM for further consideration. For example, when a package is mapped, its sub-packages are put in PM. The mapped ones are moved to AM, and the remaining ones that were not mapped to their parent’s children are put back to UM for later processing.

When there are a large number of unmapped nodes, a pair-wise comparison of all nodes in UM would be inefficient. To overcome this problem, OAT uses the following hash-based optimizations: OAT first hashes the nodes in UM by their name and only compares the nodes with the same name to find the mapped nodes in UM. For the remaining nodes in UM, it then hashes those nodes in each set by their structural characteristic vectors using the LSH hashing scheme Andoni and PiotrIndyk. This LSH-based filtering helps OAT divide the remaining nodes in UM into the subsets with the same hashcode, and apply the MaxMatch function (Figure 5.3) on the nodes in each subset.

The characteristic vector of a class is summed-up from the normalized vectors of their methods. We normalize methods’ vectors to have the same length of 1 before summing them up to build the vector of the containing class to avoid the problem of unbalance sizes between those methods. However, in other cases for comparing methods, their corresponding vectors will not be normalized.
When mapping method nodes in $UM$, in addition to sub-dividing the nodes using their hash values, we also use $s_d$ to quickly identify methods with a similar declaration.

### 5.4.2 Fine-grained Differencing

The fine-grained changes within methods, e.g., changes to statements and expressions, are detected by a novel tree editing algorithm called Treedit. It determines one-to-one mappings of the nodes in two ASTs representing two versions of a file or method (i.e., tree alignment). For example, un-aligned nodes are considered as added/deleted. The aligned nodes with different attributes or locations are considered as updated or moved. Treedit is a heuristic algorithm for the tree alignment problem to avoid the complexity of the optimal tree editing algorithms which could be too computationally expensive on large-scale systems. The heuristics are based on the following observations:

1. If two nodes are aligned, their ancestors and descendants are likely to be aligned. Thus, already-aligned nodes can be used to find candidates for alignment in their ancestors and descendants.

2. The aligned leaf nodes often have similar textual attributes. Especially, the leaf nodes belonging to unchanged text regions are unchanged. This suggests the use of textual similarity as an alignment criteria to map leaf nodes in two versions.

3. Two versions of a compound entity generally have similar structures, which could be measured by a similarity function $sim$. If the sub-tree rooted at $u$ is highly similar to the sub-tree rooted at $u'$, then $u$ and $u'$ are likely to be aligned. In other words, this suggests using structural similarity as an alignment criteria for inner nodes.

Based on those observations, in Treedit, the alignment process has three phases: 1) initial mapping, 2) bottom-up mapping, and 3) top-down mapping. First, Treedit initially maps as many leaf nodes as possible. Then using such initial mapping, Treedit maps the nodes at higher levels in a bottom-up manner, and uses the above observations to find the candidates and to choose the mapped nodes based on their structural similarity. After going up to the root node, the algorithm goes top-down and attempts to align the unmapped descendants of the mapped nodes, based on the similarity of their structures (inner nodes) or textual attributes (leaf nodes).

#### 5.4.2.1 Initial Mapping

The initial mapping step has two key phases. The first one aims to map the leaf nodes of unchanged text segments. Two ASTs are unparsed into their text-based versions (Treedit uses the unparsed
text instead of the original text to discard the differences in formatting and other cosmetic changes). Then, Treedit performs text-based differencing using the longest common subsequence algorithm on two sequences of text lines in order to detect and align the unchanged text lines. The alignment of unchanged text lines will partition the text lines into the (un)changed segments as in Figure 5.5a). If the textual content of a leaf node (e.g. the string literal) belongs to more than one segment, those segments will be joined into one segment. The joined segment is considered as changed if it is joined from a changed segment. With this joining step, each leaf node belongs to only one segment.

Then, Treedit traverses two trees, exports their sequences of leaf nodes, and marks the leaf nodes belonging to unchanged segments as “unchanged”. Such unchanged leaf nodes in two sequences are mapped one-by-one in order. For example, in Figure 5.5, two sequences of leaf nodes are \([a, b, a, a, 1]\) and \([a, b, a, a, b, ok, true]\). Because the first two nodes of those sequences belong to an unchanged text line (if \((a > b)\)), they are mapped one to one: \(a \rightarrow a, b \rightarrow b\).

In the second phase of initial mapping, Treedit aligns the leaf nodes belonging to the changed text segments. Such segments contain all changed leaf nodes and might contain unchanged leaf nodes. For example, two sequences of nodes \([a, a, 1]\) and \([a, a, b, ok, true]\) correspond to changed text segments. However, the first node \(a\) is unchanged in the statement \(a = a + 1\). To find the aligned nodes, for each pair of the aligned segments of changed lines in the two versions, Treedit finds the longest common subsequence (LCS) between the corresponding sequences of the leaf nodes. Two nodes are considered as matched if they have the same Type and textual content. The matched nodes of the resulting subsequences are aligned to each other as unchanged leaf nodes. In the above sequences, Treedit finds the common subsequences \([a, a]\) and maps to the corresponding nodes in \([a, a]\).
5.4.2.2 Bottom-Up Mapping

After matching all possible leaf nodes, Treedit aligns the inner nodes in a bottom-up fashion. If an inner node $u \in T$ has a descendant $u_1$ mapped to $v_1$, $u$ will be compared to every ancestor of $v_1$. Then, $u$ will be aligned to a candidate node $v$ if the subtrees rooted at $u$ and $v$ are sufficiently similar in structure and type (measured via Exas similarity measurement). If no such $v$ exists, $u$ is considered as unmapped. For example, in Figure 5.2, both $E_1'$ and the root if nodes contain the mapped nodes to the nodes in $E_1$. However, because $E_1$ is identical to $E_1'$, they are mapped to each other. Similarly, $A$ is also mapped to $A'$, even they are not identical in structure.

5.4.2.3 Top-Down Mapping

The bottom-up process might be unable to align some nodes, such as the moved nodes and renamed identifier nodes. Thus, after bottom-up mapping, Treedit examines the two trees in the top-down fashion and maps those not-yet aligned nodes based on the already-aligned nodes.

Given a pair of mapped nodes $u$ and $v$ from the bottom-up pass, it proceeds the mapping between their descendant nodes by the following mechanism: First, it performs a greedy algorithm to find additional mapped nodes between the children nodes of $u$ and $v$. The already mapped nodes are kept. If an un-aligned child node is an inner one, their descendants are compared based on Exas structural similarity as in the previous step. If it is an un-aligned leaf nodes, Treedit computes their similarity based on their attributes. For the AST’s leaf nodes, since their textual contents are generally identifiers and literals, such attributes are first separated as the sequences of words, using well-known naming conventions such as Hungarian or Camel. For example, “getFileName” is separated into “get”, “file”, and “name”. The similarity of two words is computed via the Levenshtein distance, a string-based similarity measure.

Then, after the children nodes are mapped, a longest common subsequence algorithm is run on those two sequences of mapped nodes. The aligned nodes not belonging to the resulting longest common subsequence are considered as moved. Figure 5.5b) illustrates the top-down mapping process with node 4 being moved.

5.4.3 Collecting Code Changes

For each pair of trees $T$ and $T'$ of a changed method, we aim to collect all changes with different heights (sizes). Our tool traverses them in pre-order from the roots to get the changes. If a node $n$ in $T$ is mapped to a node $n'$ in $T'$ and they change in either labels or children nodes, a code change
represented by a pair of trees \((T(n), T'(n'))\) is extracted, where \(T(n)\) and \(T'(n')\) are the trees rooted at \(n\) and \(n'\), respectively. If a node \(n\) in \(T\) does not have any mapped nodes in \(T'\), a change of \((T(n), \text{null})\) is extracted. Similarly, if a node \(n'\) in \(T'\) is un-mapped, a change of \((\text{null}, T'(n'))\) is extracted. Note that, if a tree is deleted or added, all of its sub-trees will also be collected into the change database because the changes of the sub-trees constitute to the changes of that root tree.

Figure 5.6 shows all collected changes with different heights (sizes) from 1–3 for the illustration example in Figure 5.1. The change of height 4 is shown in Figure 5.2. Note that, a change of small size is included in a larger one. We analyze change repetitiveness at different sizes of code fragments.

5.5 Change Database Building and Change Repetitiveness Computing

5.5.1 Design Strategies

We design our data structure and algorithm with the key idea that a change and its repeated one have the same type and size, and the same pair of source and target ASTs after normalization. If we create a hashcode for each change by concatenating the hashcodes of its normalized source and target trees, repeated changes will have the same hashcode. Thus, if the changes are grouped based on hashcodes computed via that scheme, repeated changes will be hashed to the same group, which have the same size and type. We used those groups to compute the number of repeated changes by size and by type.

Based on that idea, we extracted the changes in our dataset into a change database. This database is a dictionary of change groups indexed by hashcodes computed as explained above. Each change group contains a hash table to map a project’s id to the number of changes having the same hashcode in that project. This hash table is used to compute the repetitiveness levels in within and cross-project settings. That is, if a project \(p\) has a count \(n_p\), then \(p\) will have \(n_p - 1\) changes repeated within \(p\). If the hash table has another project, then all \(n_p\) changes of \(p\) are counted toward cross-project repetitiveness.
Figure 5.7: Algorithm for Extracting and Computing Repetitiveness

5.5.2 Detailed Algorithm

Figure 5.7 lists the algorithm for building the change database (function BuildDatabase, lines 1–9) and computing the repetitiveness (function Compute, lines 11–22). To build the change database, the algorithm processes each change $c$ in each project $p$. First, it computes the hashcode for $c$ (line 5). If the database does not have a change group with that hash code, a new change group is created for it (lines 6–7). Then, the count value for $p$ is updated (line 8).

Function Compute (line 11) computes repetitiveness in size. $N[p,s]$ is the total number of changes of size $s$ in project $p$. $Nw[p,s]$ and $Nc[p,s]$ are the numbers of changes repeated within and across projects, respectively. They are updated using the above strategy (lines 15–18). Then, the repetitiveness values $Rw[p,s]$ and $Rc[p,s]$, are computed as the ratios of $Nw[p,s]$ and $Nc[p,s]$ over $N[p,s]$, respectively. The computation for type is in a similar manner.

5.6 Analysis Results

5.6.1 Boxplot Representation of General Change and Fix Repetitiveness

Figure 5.8 shows repetitiveness results of general and fixing changes in both within- and cross-project settings. For each change size $s$ from 1–10, we computed the repetitiveness $R(s)$ for all corresponding changes of every project. Thus, for each size $s$, we have a distribution of 5,682 projects as data points.
Figure 5.8: The Repetitiveness of Code Changes and Fixes over Change Size for all 5,682 Projects in the Corpus

This distribution is plotted as a box plot, with five quantiles: 5% (the lower whisker), 25% (the lower edge of the box), 50% (the middle line), 75% (the upper edge of the box), and 95% (the upper whisker). There are 10 box plots for 10 sizes. Let us explain the topmost boxplot in Figure 5.8 for the within-project repetitiveness of general changes of size 1.

1. The 50% quantile, i.e., median, is at 73%. Since the median could be seen as the center of the distribution, one could say that on average, the projects in our dataset have 73% of their size 1 changes repeated within individual project.

2. The 25% quantile is at 68%, implying that more than 75% of the projects have at least 68% those changes repeated within a project.

3. The 75% quantile is at 77%, meaning that at least 25% of projects have those changes repeated more than 77%.

4. The 95% quantile is 86%, meaning that at least 5% of projects have 86% of size 1 changes repeated within a project.

5. The inter-quartile (difference between 25% and 75% lines) is 9%, referring to the spread of the distribution.
5.6.2 Exponential Relationship of Repetitiveness and Size

Comparing the box plots for different change sizes in both within and cross-project settings, we see that repetitiveness is very high for small changes, but it significantly decreases when the change size increases, as expected. For example, in the cross-project setting, size 2 changes have median repetitiveness of more than 60%, but that for size 6 changes drops below 5%. The repetitiveness of larger changes (size of 7–10) is small (less than 2% on average).

We modeled $R(s)$ and $s$ with several classes of simple curves, and found that the exponential curve $R(s) = \alpha e^{\beta s}$ represents their relationship the best. We used the least square method to compute two parameters $\alpha$ and $\beta$ for every project. The goodness of fit is measured by the coefficient of determination $R^2$. The closer $R^2$ is to 1, the better the fit is.

![Box plots for repetitiveness over change size](image)

Figure 5.9: $R^2$ of Fitted Exponential Curve to Repetitiveness over Change Size

Figure 5.9 shows the boxplots summarizing the distributions of the goodness of fit over all projects in our dataset. As seen, it is very high for both general and fixing changes in both within- and cross-project settings. For example, for general changes in the within-project setting, median $R^2$ is 0.978. That means, on average, the relation between repetitiveness and size of at least 97.8% of the within-project changes can be explained by the fitted exponential model. In all four cases, the values of medians are higher than 0.970. We can also see that 75% of the projects (lower edge of the boxes in the plots) have $R^2$ of at least 0.90. We conclude that the repetitiveness of code changes decreases exponentially when change size increases.

As an implication, the automatic program repair tools should focus on the change fragments with the syntactic units of the height from 2–6 to reduce the search spaces of solutions (size 1 changes on literals/variables have less value in suggestion).
5.6.3 Within and Cross-project Repetitiveness Comparison

In Figure 5.8, the box plots for sizes 1–5 in the cross-project setting are higher than those in the within-project setting. For example, for size 1 changes, the cross-project repetitiveness median is 86%, while the within-project one is 73%. For size 2 changes, the corresponding numbers are 64% and 46%.

To statistically verify this observation, we use a paired Wilcoxon test to compare the distributions $R(s)$ in within-project and cross-project settings. All the tests for sizes 1–5 infer that cross-project repetitiveness is statistically higher than within-project repetitiveness. For large sizes, changes repeat the same or slightly less frequently in the cross-project setting.

For all sizes except size 1, the inter-quartiles of box plots in the cross-project setting are always shorter than those in the within-project one. For example, the inter-quartile for cross-project repetitiveness with size 2 changes is 14%, while that in the within-project setting is 15%. The difference tends to increase as the size increases. Only that at size 1 shows the opposite trend with the corresponding numbers of 10% and 9%. Nevertheless, that result implies that repetitiveness in cross-project setting is more stable. Thus, repeated changes are more likely to be found across projects.

5.6.4 Repetitiveness of Bug Fixes

Figure 5.8 and Figure 5.9 show that repetitiveness of fixes is similar to that of general changes. At small sizes (1–2), fixes repeat frequently, with repetitiveness usually higher than 60% in the cross-project setting. At larger sizes (6–10), fixes repeat less frequently, with repetitiveness often less than 5%. Thus, automatic program repair methods should focus on the change fragments with the small sizes of 2–5.

Importantly, we conducted a paired Wilcoxon test and found that at smaller sizes (from 1 to 5), cross-project repetitiveness of fixing changes is statistically higher than that in the within-project setting. As an example, the median of the cross-project repetitiveness for size 2 fixing changes is 66% in comparison with 22% in the within-project setting. The corresponding numbers for size 3 fixing changes are 42% and 8%. As seen, the within-project repetitiveness of fixing changes is low. Those results suggest that automatic program repairing tools should not rely solely on the changes in an individual project, but rather make use of repeated bug fixes across different projects.

As seen in Figure 5.8, the repetitiveness of cross-project changes is comparable to that of cross-project fixes. However, paired Wilcoxon test results showed that at the small sizes (1–3), repetitiveness of fixing changes is statistically higher than that of general changes. This suggests that bug fixes tend to be at small sizes. Thus, automatic patching tools could start with small changes from other projects and gradually compose them. For larger sizes, they could use within-project changes.
5.6.5 Repetitiveness on Individual Datasets in SourceForge and GitHub

In the previous sections, we presented the repetitiveness on the full dataset combined from both SourceForge and GitHub. In this section, we show the results for the study on individual ones. The line charts in Figure 5.10 summarize the trends of repetitiveness on the three datasets: SourceForge, GitHub and the combined one. The horizontal and vertical axes are for the sizes of changes and the corresponding median values of repetitiveness over all projects in a dataset, respectively. Each line shows the trend of repetitiveness of changes over size for a dataset. We ran experiments in both within- and cross-project settings for both general and fixing changes as in the previous study.

The result shows that the repetitiveness on individual datasets exhibits the same trend as in the combined one. First, the repetitiveness level is high for changes of the small sizes (from 1 to 6) and exponentially decreases with size. We also observe that the cross-project repetitiveness tends to be higher than within-project repetitiveness for both general and fixing changes. Finally, on all datasets, the repetitiveness of fixing changes is higher than that of general changes at small sizes (1–3). This result suggests that between those two datasets, the observations from one could be applied to the other.

Figure 5.10: Repetitiveness of Code Changes and Fixes over Change Size for Individual Datasets. (The horizontal and vertical axes are for the sizes of changes and the corresponding median values of repetitiveness over all projects in the dataset, respectively)
Figure 5.11: Cross-project Repetitiveness of General Changes and Fixing Changes. (The horizontal and vertical axes are for the sizes of changes and the corresponding repetitiveness values in each project, respectively)

Figure 5.12: Within-project Repetitiveness of General Changes and Fixing Changes. (The horizontal and vertical axes are for the sizes of changes and the corresponding repetitiveness values in each project, respectively)
5.6.6 Repetitiveness on Representative Projects

While previous sections present the results on all 5,682 projects in our full dataset, this section presents the results for a small set of six representative projects for further detailed analysis. Among them, three are selected from SourceForge projects with different ranges of code revisions: one with some ten thousands, one with some thousands and one with some hundreds of revisions. Three others are selected from GitHub projects in the same way.

Figure 5.11 plots the cross-project repetitiveness values of general changes (in solid lines) and fixing changes (in dashed lines) for those projects. The three plots in the upper row are for projects from SourceForge and the lower ones are for the projects from GitHub. The projects in each row are ordered by the number of their revisions. As seen, although following the same trends, the curve for one project might look different from that of another. For example, the curve for general changes in openkm is higher than that in jmol (they have similar \( \beta \) parameters, however, \( \alpha \) for the former is larger than that for the latter). Figure 5.11 also illustrates that at smaller sizes, some projects have the repetitiveness of fixing changes higher than that of general changes, such as jmol, elfframework and harmonyclasslib.

Figure 5.12 plots for the same set of projects in the within-project setting. As seen, the repetitiveness of fixing changes is lower than that of general changes. In the projects such as openkm and crazyproger/kotlin, the difference is significant.

5.6.7 Repetitiveness and Change Type

5.6.7.1 Change Type

We perform another analysis for the repetitiveness of changes classified based on the types of the corresponding code structures. Given a change as a pair of sub-trees \((s, t)\) in the ASTs, its type is defined as the AST node type of \(s\). If \(s\) is null, the AST node type of \(t\) will be used.

The repetitiveness of a change type is computed as the ratio of the number of repeated changes of that type over the total number of changes of that type in all projects. From the previous results, we focused on the repetitiveness in the cross-project setting. We did not compute separately for each project. In addition to general changes, we computed the repetitiveness of fixing changes.

We choose 30 most popular AST node types and divide them in 4 groups. The Array group contains the nodes representing program elements related to arrays, such as an array access or array declaration. The Call group contains nodes representing the elements related to method/constructor calls and field accesses. The Expression group is for expressions. The Statement group contains all statements such as if, while, try, throw, etc.
Figure 5.13: Repetitiveness of Code Changes and Fixes over Change Type

5.6.7.2 Repetitiveness

Figure 5.13 shows that the repetitiveness for changes vary according to change types. It is very high for changes related to arrays, expressions, and calls (often 50–80%), while it is much lower for common statements such as if or while (often less than 40%). It is interesting that changes to method calls are the most popular and frequently repeated (40% repetitiveness), while changes to if statements are also popular but repeat less frequently (only 30% repetitiveness). Change size is a possible explanation for this observation. Array accesses and method calls (especially super calls) are structurally simpler than the compound statements (e.g., if or while), thus they could repeat more. For example, more than 90% of changes to array accesses have sizes of 1–3, while only 3% of changes to if statements have such small sizes. In addition, among statements, the small ones such as case, throw, and assert statements also repeat more frequently than the larger ones (45–75%).
Importantly, as seen in Figure 5.13, cross-project repetitiveness of bug fixes is high, especially for small program constructs. It is as high as in general changes and much higher than the fixing changes in the within-project setting. This result on change repetitiveness over change size and type suggests that the aforementioned automated program repair should focus on the fixes with small sizes and of highly repetitive types such as array access, method calls, and if/case statements.

5.6.8 Threats to Validity

Although our dataset contains a large number of projects, they are all written in Java. Thus, the observations on the repetitiveness of changes on size and type might not be generalized for projects written in other languages or paradigms. In addition, all subjects are open-source, thus, their repetitiveness characteristics, especially in the cross-project setting, might not be the same for commercial software.

Our results have the inherent threats from our prior technique for program differencing Nguyen et al. (2012) and orgin analysis Nguyen et al. (2010a). The results are limited due to the use of the keyword-based approach Zimmermann et al. (2007) to identify bug fixes among all revisions.

5.7 Related Work

5.7.1 Large-scale Studies on Uniqueness and Repetitiveness of Source Code

Our study is related to the large-scale study by Gabel and Su Gabel and Su (2010) on the uniqueness of source code on more than 420 million LOCs in 6,000 software projects. They consider a file as a sequence of syntactical tokens with the abstraction on variables’ names. They reported syntactic redundancy at levels of granularity from 6–40 tokens. At the level of granularity with 6 tokens, 50–100% of each project is redundant. Later, in a study about 20 projects, Hindle et al. Hindle et al. (2012) have used n-gram model to show that source code has high repetitiveness, and n-gram model has good predictability and could be useful in code suggestion. Another large-scale study on code reuse at the file level was from Mockus Mockus (2007, 2009) on 13.2 millions source files in continually-growing 38.7 thousand unique projects. They reported that more than 50% of the files were used in multiple projects.

5.7.2 Studies on Code Changes

Our study is also related to the studies on code changes. Recently, Barr et al. Barr et al. (2014) investigated a history of 15,723 commits in a large number of open-source Java projects to determine the extent to which these commits can be composed/reconstituted from existing code. They reported a high degree of graftability, independent of size and type of commits. They also found that changes
are 43% graftable from the exact version of the software being changed Barr et al. (2014). Our study relied on changes to ASTs, while in their study, they used lines of code. Moreover, we focused on the repetitiveness of changes and fixes, while they aimed to examine the degree of graftability. Both studies provide a foundation for automatic program repair tools on reusing existing code changes and bug fixes.

Ray et al. Ray and Kim (2012) conducted a large case study on 18 years of BSD product family and reported large number of patches being ported between forked projects. In the study, they used Repertoire Ray et al. (2012), a tool to identify ported edits by comparing the content of individual patches. Meng et al. Meng et al. (2013) propose LASE, a tool to automate similar changes from examples. It creates context-aware edit script, identifies the locations and transforms the code. Negara et al. Negara et al. (2014) aim to discover frequent code change patterns by using closed frequent itemset mining on a continuous sequence of code changes expressed as tree editing operations.

5.7.3 Code Clones

There are a large body of research and tools on clone detection, which is concerned with the detection of copy-and-paste fragments of code Bellon et al. (2007); Roy et al. (2009). Generally, they can be classified based on their code representations. The typical categories are text-based Ducasse et al. (1999); Marcus and Maletic (2001), token-based Baker (1997); Kamiya et al. (2002); Li et al. (2006); Mende et al. (2009), tree-based Baxter et al. (1998); Jiang et al. (2007a); Fluri et al. (2007), and graph-based Komondoor and Horwitz (2001). There have been several empirical studies on software changes Tao et al. (2012), non-essential changes Kawrykow and Robillard (2011), change-based bug prediction Shivaji et al. (2013); Giger et al. (2011), code clone changes Kim et al. (2005a), cloning across projects Al-Ekram et al. (2005), patch identification Tian et al. (2012), threats when using version histories to study software evolution Negara et al. (2012), etc. Giger et al. Giger et al. (2012) proposed an approach to predict type of changes such as condition changes, interface modifications, inserts or deletions of methods and attributes, or other kinds of statement changes. They use the types and code churn for bug prediction Giger et al. (2011). Our prediction study does not have different types of changes, but focuses more exact fine-grained changes.

5.7.4 Applications of Repetitiveness of Code Changes

There are advanced approaches in automatically generating/synthesizing the program fixes based on the previously seen fixes in the projects’ histories Kim et al. (2013). Weimer et al. Goues et al. (2012) proposed GenProg, a patch generation method that is based on genetic programming. Kim et al. Kim et al. (2013) introduced PAR, an automatic pattern-based program repair method, that learns common
patterns from prior human-written patches. Our study provides empirical evidences for such automatic patch generation approaches. Our prior study in FixWizard Nguyen et al. (2010b) and a study by Kim et al. Kim et al. (2006a) have confirmed the recurring nature of fixes. However, those studies were conducted in a much smaller scale than our study with less than ten projects.

5.8 Conclusions

In this chapter, we present a study of repetitiveness of code changes in software evolution. Repetitiveness is defined as the ratio of repeated changes over total changes. We model a change as a pair of old and new AST sub-trees within a method. First, we found that repetitiveness of changes could be very high at small sizes and decreases exponentially as size increases. Second, repetitiveness is higher and more stable in cross-project setting than in within-project one. Third, fixing changes repeat similarly to general changes.
CHAPTER 6. CODE CHANGE SUGGESTION

Prior research has shown that source code and its changes are repetitive. Several approaches have leveraged this information to detect change/fix patterns and support software engineering tasks. An important task among them is suggesting relevant changes/fixes. In this chapter, we propose TasC, a novel statistical model that leverages the context of change tasks in software development history to suggest fine-grained code change/fix at the program statement level. We use latent Dirichlet allocation (LDA) to capture the task context where topics are used to model change tasks. We also propose a novel technique for measuring the similarity of code fragments and code changes using the task context.

We conducted an empirical evaluation on a large dataset of 88 open-source Java projects containing more than 200 thousand source files and 3.5 million source lines of code (SLOCs) in their last revisions. In terms of changes, our dataset contains 88 thousand code revisions, of which 20 thousand are fixing changes. We extracted almost 500 thousand statement-level changes and almost 100 thousand statement-level fixes from 423 thousand changed methods with a total of 55 million AST nodes. Our result shows that TasC improves the suggestion accuracy relatively up to 130%–250% in comparison with the base models that do not use contextual information. Compared with other types of contexts, it outperforms the models using structural and co-change contexts.

6.1 Introduction

Prior studies have shown that code changes are repetitive Barr et al. (2014); Negara et al. (2014); Nguyen et al. (2013b). Due to the practice of software reuse, source code in software projects could contain code fragments with a certain degree of similarity. That could lead to the repeated/similar changes and bug fixes to source code. Moreover, the repeated/similar programming tasks often require the repeated/similar changes as well. The research relevant to repeated changes can be broadly classified into two groups: 1) detecting change/fix patterns, and 2) leveraging change patterns to support software engineering (SE) tasks. For example, the frequent changes to adapt to the changes to a framework or a library’s APIs are used to support framework/API adaptation Dagenais and Robillard (2008); Nguyen
et al. (2010a). Fine-grained change patterns Negara et al. (2014) are detected, and the IDE designers can build automated support for such frequent editing changes. Migration patterns are learned to support code migration from one language to another Zhong et al. (2010); Nguyen et al. (2014a). Another important application is to suggest the relevant changes/fixes to some defects Nguyen et al. (2010b); Kim et al. (2013) by leveraging the detected fix patterns in the same project or other projects.

The approaches to detect change/fix patterns have examined the changes/fixes in their context with relation to the other changes/fixes in the same tasks, commits, or in the change history of a project Negara et al. (2014); Livshits and Zimmermann (2005). However, the existing approaches to leverage change/fix patterns for change and bug-fix suggestion are limited in using such context. For example, Ying et al. Ying et al. (2004) and Zimmermann et al. Zimmermann et al. (2004) use history to suggest co-changes only at the coarse-grained levels of methods and files. FixWizard Nguyen et al. (2010b) leverages similar structural code context to adapt fixes from one place to another.

The key limitation of such co-change context and structural context is that the relations and connections among changes for the same task in multiple transactions in the change history are not captured due to their local perspectives. For example, structural context considers only structures of code in the current file. Co-change context can consider the co-changed files or co-changed fragments in the same transaction. However, none of the state-of-the-art approaches to support change/fix suggestion takes into account a broader view of changes in the change task. In fact, the recent changes could help in suggesting the change to other fragments since those changes are parts of the same task that requires them to co-change.

In this work, we leverage the context of change tasks in the project’s recent history to develop TasC, a statistical model for suggesting the next change/fix at the fine-grained level for a given code statement in a program. The key idea is that the knowledge on the task(s) of the changes in the recent and current transactions will be useful to predict the next change/fix because the changes for the same task are related and might need to go together. Let us call such information the task context. In our work, the task context is modeled via latent Dirichlet allocation (LDA) Blei et al. (2003) where the topics are discovered to model the tasks for changes. TasC can be used in an IDE, which processes the current editing changes/fixes and uses them as well as recent changes/fixes in the history to suggest a potential change/fix to a given code fragment.

We conducted a large-scale empirical evaluation with a large data set of 88 open-source Java projects from SourceForge.net, with 3.6 million source lines of code (SLOC) at the latest revisions, 88 thousand code change revisions (20 thousand fixing revisions), 300 thousand changed files, and 116 million changed SLOCs. We extracted consecutive revisions from the code repositories of those projects and built the
changes at the abstract syntax tree (AST) level. A change is modeled as a pair of subtrees \((s, t)\) in the ASTs for all statements. The (sub)trees are normalized via alpha-renaming the local variables and abstracting the literals.

Our empirical result shows that with the task context, \(TasC\) is able to make a significant relative improvement over an existing model that is based solely on the repeated changes and does not consider any context. Specifically, \(TasC\) relatively improves up to 250% top-1 accuracy. This improvement suggests that topic modeling is a good way to capture the tasks in code changes. We also observed that using tasks derived from recent transactions, i.e. within certain window of commits, achieves better accuracy than using tasks from the entire history. This shows the temporal locality of tasks/topics for code changes. We also observed the spatial locality of tasks/topics for code changes where using task context within a project works better than using task across projects. The suggestion accuracy for bug fixes, a special type of changes, is lower than that for general changes in within-project setting, and higher in the cross-project setting. We also compared \(TasC\) with the model using the co-change context, which consists of fine-grained changes that go together in the same transactions. Our result shows that \(TasC\) relatively improves up to 130% top-1 accuracy over the model with the co-change context. It also outperforms a model that uses the structural context of the given code fragment that needs to be changed. In that model, if two code fragments with the same structural context in term of containing structural units, the change to one fragment is used to suggest the other. The key contributions of this chapter include

1. A novel technique using topic modeling to measure the similarity between code fragments and code changes,
2. A statistical model, \(TasC\), to suggest the change/fix, considering the task context, and
3. A large-scale empirical evaluation on \(TasC\)'s accuracy.

Section 6.2 formulates the change suggestion problem and concepts in our solution. Section 6.3 explains how tasks are modeled using LDA. Section 6.4 describes the suggestion algorithm. Section 6.5 presents the empirical evaluation on \(TasC\). Section 6.6 is for related work. Conclusions appear last.

### 6.2 Problem Formulation

Let us first define the change suggestion problem and then introduce new concepts in our solution.

**Definition 6.1 (Change suggestion)** Given a fragment as a source, the change suggestion problem is to suggest the most likely change to the source fragment to produce the target fragment.
In this work, we develop a novel statistical model to suggest a change to a given code fragment with the following key ideas. First, we represent code fragments as subtrees in Abstract Syntax Tree of the program. A change is modeled as a pair of subtrees \((s, t)\) in the ASTs for all statements. Second, we leverage the context of change tasks in the project’s recent history and to extract features to represent the task context with Latent Dirichlet Allocation (LDA) Blei et al. (2003). Finally, our suggestion model learns the changes with the task context and recommends the most likely target fragment of the given source fragment in a program.

6.2.1 Transactions and Tasks

We are interested in the changes committed to a repository in the same transaction.

**Definition 6.2 (Transaction)** A transaction is a collection of the code changes that belong to a commit in a version control repository.

**Definition 6.3 (Task Context)** Task context of a change is the set of tasks being realized in a change transaction.

Developers change code to fulfill certain purposes/goals to complete one or more tasks such as reading and processing tokens with a text scanner (Figure 6.1) and/or fixing an IndexOutOfBoundsException. Those tasks are realized in source code via concrete code changes. We use topic modeling to recover this hidden information and use it as context for code changes. Details will be discussed in Section 6.3.2. Note that for the problem of suggesting changes in this chapter, the input is a statement that a developer wants to change and the output is a ranked list of most likely target statements for the change.

6.2.2 Fine-grained Code Change Representation

In this change suggestion problem, we also choose to represent code fragments at the syntax level in the same way as in our change repetitiveness study in Section 5.3. A **code fragment** in a source file is defined as a (syntactically correct) statement and is represented as a subtree in the Abstract Syntax Tree (AST) of the file.

When a statement is changed, its AST is changed to another AST representing the new fragment. A code change is represented as a pair of ASTs corresponding to the fragments of the two statements before and after the change. Figure 6.1 illustrates an example of such changes. The source fragment shows the code that checks if there exist more tokens to be read and whether the number of tokens read is still smaller than the maximum value \(\text{MAX}\). If the condition is satisfied, the tokens are processed and
appended into a new line whenever the number of tokens is divisible by 10. The source fragment is changed into the target fragment. Comparing the source and target fragments, we can see that 1) the operator ‘<’ is replaced with ‘<=’, 2) the literal string “\r\n” is changed into `System.lineSeparator()` to support different OSs (since Linux does not use “\r\n”), and 3) the `else` part is newly added to insert a whitespace after each token. Figure 6.2 shows the two ASTs representing the source and target fragments of the change. \footnote{For simplicity, we do not draw the nodes of type `ExpressionStatement` which are the parents of the `method call` nodes under the if statements.}

**Collapsing process.** Since some statements can be compound statements, i.e., having other statement(s) in their bodies, when a statement is changed, all containing statements could be automatically considered as changed. For example, a single change to a literal in the code can cause the whole method to be considered as changed. This would lead to a huge number of changes with large sizes. We avoid this effect by replacing the body statement(s), if any, of compound statements with empty blocks. We call this process *collapsing*. For example, an `if` statement will be represented as an AST which roots at an `if` node, and contains a child sub-tree for its condition expression, a `block` node for its `then branch` and possibly another `block` node for its `else branch`. The tree (b) in Figure 6.3 shows such an example for the `if` statement represented by the lower tree in Figure 6.2. We represent code change as follows.

**Definition 6.4 (Code Change)** A code change at the statement level is represented as a pair of ASTs \((s,t)\) where \(s\) and \(t\) are not label-isomorphic. The trees \(s\) or \(t\) can be a null tree or a tree representing a statement obtained from the original statement by replacing all sub-statements with empty block statements.

In this definition, \(s\) and \(t\) are called source and target trees, respectively. Either of them (but not both) could be a null tree. \(s\) or \(t\) is a null tree when the change is an addition or deletion of code,
respectively. Since AST are labeled trees, the condition of not being label-isomorphic is needed to specify that the code fragments before and after change are different. The fragments are normalized via alpha-renaming as in Section 5.3.

Figure 6.2 shows the AST’s subtrees for the code changes in the illustration example after normalization. The node for the variable tokenSc is labeled as ID v1 while the one for n is labeled as ID v2 since they are local variables and, thus, alpha-renamed into v1 and v2, respectively. The node for the literal value 10 is labeled as LIT NUM. MAX is a constant of the class, not a local variable, thus, it is not alpha-renamed. During alpha-renaming, the same local variable name could be relabeled to different names in different code fragments depending on its locations in the corresponding fragments. For example, the local variable n is renamed to v2 in the while fragment (Figure 6.3a) while it is renamed to v1 in the if fragment (Figure 6.3b) since it is the second local variable in the former fragment, while it is the first in the latter one.
6.2.3 Fine-grained Code Change Extraction

This step derives the statement changes within the body of each changed method. We apply the same process for detecting and extracting code changes in Section 5.4. The only difference is that we keep only changes at statement level.

Figure 6.3 shows all collected changes for the illustration example in Figure 6.1. The second pair is for the modification to the if statement. Note that, the statements in its body (then and else branches) are replaced with the empty block statements after collapsing. The first pair is for the change in the operator. The third one is for the change from a string literal to a method call. The last one is an addition of the method call append.

6.3 Modeling Task Context with LDA

6.3.1 Key design strategies

We model task context for changes via a LDA-based topic model. The idea is that if the purpose(s)/task(s) of the current change transaction and those of the recent changes can be discovered, we can leverage such knowledge to predict the next change since a task might require changes that go together as parts of the task.

To find the tasks, we model the task context using LDA as follows. A change is considered as a sentence with multiple words involving in the changed fragments. In the context of change suggestion problem, let us use the term token, instead of “word”. A transaction/commit is a collection of...
Vocabulary $V$ of all tokens in changes $\{w_1, w_2, w_3, w_4, \ldots\}$

Token-distribution vectors $\phi_k$ for all tasks $1$-$K$

<table>
<thead>
<tr>
<th>Task</th>
<th>$\phi_1$</th>
<th>Task</th>
<th>$\phi_2$</th>
<th>Task</th>
<th>$\phi_K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1$</td>
<td>0.25</td>
<td>$w_2$</td>
<td>0.3</td>
<td>$w_1$</td>
<td>0.25</td>
</tr>
<tr>
<td>$w_2$</td>
<td>0.2</td>
<td>$w_3$</td>
<td>0.25</td>
<td>$w_2$</td>
<td>0.03</td>
</tr>
<tr>
<td>$w_3$</td>
<td>0.12</td>
<td>$w_4$</td>
<td>0.18</td>
<td>$w_3$</td>
<td>0.25</td>
</tr>
<tr>
<td>$w_4$</td>
<td>0.12</td>
<td>$w_5$</td>
<td>0.04</td>
<td>$w_5$</td>
<td>0.02</td>
</tr>
<tr>
<td>$w_5$</td>
<td>0.10</td>
<td>$w_6$</td>
<td>0.03</td>
<td>$w_7$</td>
<td>0.01</td>
</tr>
<tr>
<td>$w_6$</td>
<td>0.10</td>
<td>$w_7$</td>
<td>0.02</td>
<td>$w_9$</td>
<td>0.03</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>0.10</td>
<td>$w_9$</td>
<td>0.02</td>
<td>$w_{11}$</td>
<td>0.01</td>
</tr>
<tr>
<td>$w_{11}$</td>
<td>0.02</td>
<td>$w_{11}$</td>
<td>0.02</td>
<td>$w_{16}$</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Transaction $t$ with tokens in changes:

Transaction

Task assigning vector $z_t$ for all tokens in all changes in $t$.

Task proportion $\theta_t$ for $t$.

Figure 6.4: LDA-based Task Context Modeling

changes (sentences), thus, also a collection of tokens, and can be viewed as a *document* in LDA. All tokens are collected in a vocabulary $V$. A *topic* in LDA is used to model a change task, which can be seen as the purpose of a change or a set of changes. A task is represented by a set of changes with associated probability for each change. For example, for the task of fixing bug #01, the probability for the change #1 to occur is 25%, while that for the change #2 to occur is 35%, and so on. Since each change is viewed as a sentence with multiple tokens involving in the changed fragments, a task can be represented by a set of such tokens with associated probabilities (see the Tasks in Figure 6.4).

A transaction (document) of changes can be for multiple purposes/tasks (topics). A transaction $t$ has a task proportion $\theta_t$ to represent the significance of each purpose in $t$. Assume that in the entire history, we have $K$ tasks. Then, $\theta_t[k]$ with $k$ in $[1,K]$ represents the proportion of task $k$ in $t$. Thus, if we use topic modeling on the set of transactions in a project, we will have the task proportion of the transaction $t$, i.e., the proportion of each task in the transaction $t$. 
6.3.2 Details on Modeling Task Context

Figure 6.4 illustrates our modeling. For each change, we collected all syntactic code tokens in the AST after normalizing the source fragment of the change. If the source is null, i.e., the change is an addition, the target fragment will be used. In the illustration example, we would collect while, ID v1, hasNext, &&, ID v2, ++, <, MAX, etc. All tokens $w_i$’s collected for all the changes in the recent history up to the current transaction are placed into the vocabulary $V$. To perform a task $k$ among all $K$ tasks, one might make different changes with different tokens from $V$. Moreover, a change $c$ in $V$ might contribute to multiple tasks. Thus, each token $w$ in a change $c$ has a probability to achieve a task $k$.

We use a token-distribution vector $\phi_k$ of size $V$ for the task $k$, i.e., each element of $\phi_k$ represents the probability that a token $w$ in a change $c$ achieves the task $k$. Putting together all of those vectors for all $K$ tasks, we have a matrix called per-task token distribution $\phi$.

A task $k$ is represented by a set of changes with the corresponding probabilities of the tokens in those changes. Those changes contribute to achieve that task. A change that does not contribute to achieve a task will have its probability of zero. Vocabulary, tasks, and per-task token-distribution matrix are meaningful for all transactions in the history.

A transaction $t$ has several changes with $N_t$ tokens. Each transaction has two associated parameters:

1. task proportion $\theta_t$: A transaction $t$ can be for multiple tasks. Thus, as in LDA, we associate each transaction $t$ with a proportion/distribution to model the contribution of the transaction $t$ to each task $k$. The higher the value $\theta_t[k]$, the higher the changes in the transaction $t$ contribute toward the task $k$. The total of all values $\theta_t[k]$ for all tasks $k = 1...K$ is 100%. For example, if $\theta_t = [0.2, 0.3, 0.4, ...]$, 20% of the changes in transaction $t$ contribute toward task 1, 30% is toward task 2, etc.

2. task assignment vector $z_t$: This vector for transaction $t$ models the assignment of the tokens in all changes in $t$ to the tasks.

To find the tasks of a transaction $t$, as in LDA, we assume that the transaction $t$ is an “instance” generated by a “machine” with 3 variables $\theta_t$, $z_t$, and $\phi$. Given a transaction $t$, the machine generates the vector $z_t$ assigning each position in $t$ a task $k$ based on the task proportion $\theta_t$. For each position, it generates a token $w$ for a change $c$ based on the task $k$ assigned to that position in $t$ and the token-selection vector $\phi_k$ of that task $k$.

The changes in all transactions in the history are observed from data. This LDA-based model can be trained to derive those 3 variables. For a new transaction $t'$, we can derive the task assignment $z_{t'}$ and the proportion $\theta_{t'}$ of the tasks in $t'$. Thus, we can derive the tasks for all transactions.
6.4 Change Suggestion Algorithm

Based on our modeling of task context via LDA, we develop a change suggestion algorithm for any given fragment of code. Our algorithm is developed with two key design ideas:

1. *Source fragments that contribute similarly to the tasks in the change transactions would be changed in the similar manner.* Thus, given a source fragment $s$ for suggestion, the likely (candidate) target fragment could be found in the *candidate changes in the past having similar source fragments with $s$ in term of their tasks.*

2. The more frequently a target has been seen in the past, the more likely it is the actual target of a given source fragment.

Let us explain how we use tasks inferred from topic modeling in Section 6.3 to measure the similarity between code fragment and then explain the detailed algorithm next.

6.4.1 Task-based Similarity Measurement for Code Fragments

The idea for this measurement is that the similarity between code fragments can be measured via their levels of contributions to the tasks. The task contributions of a fragment can be computed by combining the task contributions from the tokens in the fragment (which are computed by topic modeling).

We realize that idea by using the per-task token distribution $\phi$ computed by topic modeling. Note that in Figure 6.4, $\phi$ is the matrix formed by putting together all vectors $\phi_k$ for $k = 1..K$. We first build a *task vector for each token* via $\phi$. The size of the vector for a given token is the number of topics/tasks, each index corresponds to a topic/task and the value of an index $k$ is the probability of that token being contributed toward the task $k$. For example, in Figure 6.4, if the number of tasks $K=3$, the task vector for token $w_1$ is $v_1 = [0.25, 0.0, 0.25]$ and that for token $w_2$ is $v_2 = [0.2, 0.3, 0.03]$. Since the tasks/topics in LDA Blei et al. (2003) are assumed to be uniformly distributed over all documents in the corpus, such a task vector represents the contributions of that token to the tasks. For example, among those two tokens, $w_1$ contributes to task 1 more than $w_2$ does.

For each fragment, we first collect from its AST a sequence of syntactic tokens. This step is done after normalizing code fragments in the code change extraction process as mentioned in Section 6.2.3. *The summation of those task vectors for all tokens of a code fragment will represent the contributions of the corresponding fragment to the tasks.* For example, if a fragment is composed by two above tokens $w_1$ and $w_2$, its combined task vector is $v = [0.45, 0.3, 0.28]$, which means that it contributes the most to task 1. We normalize the combined task vector from all tokens so that the sum of all values is 1.
The normalized version the above vector \( v \) is \( \bar{v} = [0.43, 0.30, 0.27] \). Then, we use the normalized vector as the task vector for the corresponding fragment. Such task vector represents the probability of the fragment contributing to a task. The task similarity between two code fragments is measured by their shared contributions to the tasks normalized by the maximum of their contributions.

\[
Sim(f_1, f_2, \phi) = Sim(v_1, v_2) = \frac{\sum_{t=1}^{K} \min(v_1[t], v_2[t])}{\sum_{t=1}^{K} \max(v_1[t], v_2[t])}
\]  

(6.1)

### 6.4.2 Detailed Algorithm

Figure 6.5 shows the pseudo-code of the algorithm to suggest the target fragment. The input of the algorithm is a source fragment \( s \) to be changed and the database of all past changes. The algorithm will output a ranked list of likely target fragments for \( s \). To do that, the algorithm first builds the task model for the past changes by running LDA on the change transactions (line 2). The output of this step is the distributions of tokens for each task in the past. Then, we use those distributions to find the source fragments with similar tasks. The algorithm looks for all prior changes \((u, v)\) whose source fragment \( u \) is similar to the given source \( s \) with respect to their tasks (lines 4–6). The similarity measurement is shown in the formula (6.1) (Section 6.4.1). If it finds such a change \( c \), it will update the target of \( c \) in the store \( T \) of all candidate target fragments. The algorithm gives higher scores to the targets that both have occurred more frequently in the past and belong to the changes whose sources are more similar to the given source \( s \) (line 7). Since a candidate target can belong to multiple changes (with similar sources), we use the best score from all those changes when updating the store \( T \) of candidate targets (line 8). Finally, all candidate targets in \( T \) are ranked based on their scores.

### 6.5 Empirical Evaluation

We conducted empirical experiments to evaluate the quality of using task context to suggest code changes. We aim to answer two research questions:
Table 6.1: Collected Projects and Code Changes

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Projects</td>
<td>88</td>
</tr>
<tr>
<td>Total source files</td>
<td>204,468</td>
</tr>
<tr>
<td>Total SLOCs</td>
<td>3,564,214</td>
</tr>
<tr>
<td>Java code change revisions</td>
<td>88,000</td>
</tr>
<tr>
<td>Java fixing change revisions</td>
<td>19,947</td>
</tr>
<tr>
<td>Total changed files</td>
<td>290,688</td>
</tr>
<tr>
<td>Total SLOCs of changed files</td>
<td>116,481,205</td>
</tr>
<tr>
<td>Total changed methods</td>
<td>423,229</td>
</tr>
<tr>
<td>Total AST nodes of changed methods</td>
<td>54,878,550</td>
</tr>
<tr>
<td>Total detected changes</td>
<td>491,771</td>
</tr>
<tr>
<td>Total detected fixes</td>
<td>97,018</td>
</tr>
</tbody>
</table>

1. Does our model TasC using task context improve the quality of code change suggestion over the base models using only repeated changes?

2. Does the model TasC using task context improve the quality of code change suggestion over the models using other types of context such as structure Nguyen et al. (2010b) and co-change relations Zimmermann et al. (2004)?

We evaluated the suggestion quality for both general changes and bug fixing changes (fixes). We also studied several characteristics of task context in code change suggestion.

6.5.1 Data Collection

We collected code change data from open-source projects in SourceForge.net SourceForge. We downloaded and processed all Subversion (SVN) repositories of the Java projects on our local machine. To filter out the toy projects among them, we kept only projects that satisfy two criteria: 1) having standard trunks (i.e., the main line of development) in their SVN repositories, and 2) having at least 1,000 revisions of source code changes. Since the numbers of revisions greatly vary among these projects (from some thousands to some ten thousands), we collected into our dataset only the first 1,000 revisions of Java code to the trunks from those projects.

Table 6.1 summaries our dataset. There are 88 projects satisfying the criteria. They contain more than 200 thousand Java source files and 3.5 million source lines of code (SLOCs) in their last snapshots.

In terms of changes, our dataset contains 88 thousand revisions having source code changes. Among them, 20 thousands are fixing changes. To detect fixing changes, we used the keyword-based approaches Zimmermann et al. (2007), in which if the commit log message of a revision has the keywords indicating fixing activities, the code changes in that revision are considered as fixing changes. We processed all revisions and parsed 290 thousand changed source files with 116 million SLOCs. Our tool
detected 423 thousand changed methods with the total size of 55 million AST nodes. From those methods, it extracted almost 500 thousand statement-level changes and almost 100 thousand statement-level fixes.

6.5.2 Evaluation Setup and Metric

Since TasC uses LDA topic modeling to capture the task context, given a source fragment at a commit for suggestion, we need the data on the change history before that commit for training our model. We use a longitudinal setup. For each project, we divide equally the 1,000 revisions into 10 folds, each of which has 100 consecutive revisions. Folds are ordered by the commit time of their revisions.

A testing change is picked from a testing fold \( i (i = 2..10) \). The changes in the previous folds (0 to \( i - 1 \)) are used to compute the task context via topic modeling.

We measure the quality of change suggestion via top-ranked accuracy. Given a source fragment of a testing change, our tool produces a ranked list of candidate target fragments. If the actual target matches the one at the position \( k \) of the list, we count it as a hit for top-\( k \) suggestion. The accuracy of top-\( k \) suggestion is computed as the ratio between the number of top-\( k \) hits over the number of tests. We recorded both the accuracy for each project and that for the whole dataset (all projects).

To evaluate the suggestion quality in the cross-project setting, given a testing change in a project, we use the changes from all previous folds of that project along with the changes from all folds of the other projects as the training data. For topic modeling implementation, we built our model on top of the LDA library from MAchine Learning for LanguagE Toolkit (MALLET McCallum (2002)). For the parameters of LDA, we experiment different values for the number of tasks \( K \) to see its impact to the suggestion accuracy in Section 6.5.3. For other parameters, we used the suggested values from MALLET, i.e., \( \alpha=0.01, \beta=0.01 \) and the number of iterations is 1,000. In our empirical evaluation, we also performed sensitivity analysis on the similarity threshold listed at line 6 in Figure 6.5) (see Section 6.5.3).

6.5.3 Parameter Sensitivity Analysis

In this first experiment, we analyzed the impact of the similarity threshold and the number of tasks \( K \) to the suggestion accuracy. We chose to use project ONDEX. To analyze the threshold, we fixed the number of task \( K = 10 \) and run TasC with different values of the similarity threshold from 0.5 to 0.9. Figure 6.6a shows the accuracy results for different top-\( k \) suggestions. When the threshold is small, the number of candidates will be large, thus, one would expect that the accuracy is low. However, from the results, we can see that when the threshold is less than or equal to 0.8, varying it does not affects the
accuracy. This happens because of two reasons. First, we compute the ranking score by multiplying the similarity with the frequency in Figure 6.5 line 7. Second, the frequencies of candidate changes are usually small. Therefore, the candidates with low similarity have low chance to be ranked high in the suggestion list. When the threshold is increased from 0.8 to 0.9, the number of candidates drops leading to the decrease in accuracy. We use threshold of 0.8 in the next experiments because it gives the best accuracy as well as finding the minimum set of candidates.

To analyze the impact of the number of tasks \( K \), we used the similarity threshold of 0.8 and varied the value of \( K \). The accuracy results are shown in Figure 6.6b. From top-5 to top-50, the model is not sensitive to \( K \) because the numbers of candidates in the ranked lists are usually small. The best accuracy can be achieved at \( K = 10 \). When \( K \) is small, many code fragments are considered similar because the size of the topic vector is small and many fragments are grouped into the same LDA topics/tasks even though they are for different change tasks. When \( K \) is large, the task vectors of source fragments become distinct. Thus, many actual targets are not collected into the ranked list resulting in the decrease in the accuracy.
6.5.4 Locality of Task Context

In this experiment, we would like to study how the locality of training data for topic modeling affects change suggestion accuracy. We study two aspects of locality: time and space. For temporal locality, we investigated whether using recent transactions and entire change history would produce different accuracy, and if yes, which one would give better accuracy? For spatial locality, we performed the experiment to compare the accuracy in two cases: 1) the training data from within the histories of individual projects and 2) the training data from the current project as well as from the change histories of other projects.

6.5.4.1 Temporal locality of task context

We carried out this experiment in the within-project setting. For each testing change, we ran our tool with two different training datasets for LDA. The first one simulates the use of recent transactions by using only a window of a small number of revisions before the revision of the testing change. The second training dataset uses the full history prior to the revision of the testing change. In this experiment, we used the most recent fold as the window of recent transactions. The comparison result for suggestion accuracy over all projects is shown in Figure 6.7a. As seen, for all the top-k accuracy, the accuracy in the setting using a small window of prior revisions is higher than the accuracy in the setting using the full change history. Examining the results for each individual project, we observed the same trend consistently. We used a paired Wilcoxon test to compare the distributions of the accuracy over all projects in our dataset between using window of history and entire history settings. The test result shows that the accuracy for the former is significantly higher than that for the latter.

This result suggests that using a window of recent changes would be more beneficial than using the full history in capturing the task context in the problem of change suggestion. Using recent data would not only increase accuracy but also reduce the running time when suggesting changes. The intuition behind this would be that task context is local in time, which means that a task is usually realized within a certain window of transactions, rather than spanning over many transactions in the whole development history. This result is consistent with the findings by Hindle et al. Hindle et al. (2009).

6.5.4.2 Spatial locality of task context

We studied this locality by comparing the accuracy between within-project and cross-project settings. In this experiment, we used the training data from the windows of change histories. The process is similar to that of the experiment for temporal locality. The result is shown in Figure 6.7b. As seen,
Table 6.2: Suggestion accuracy comparison between the model using task context and base models.

(a) Within-project suggestion accuracy comparison

<table>
<thead>
<tr>
<th>Top-1</th>
<th>Top-2</th>
<th>Top-5</th>
<th>Top-10</th>
<th>Top-20</th>
<th>TasC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact</td>
<td>0.20</td>
<td>0.21</td>
<td>0.21</td>
<td>0.21</td>
<td>0.21</td>
</tr>
<tr>
<td>Similar</td>
<td>0.32</td>
<td>0.34</td>
<td>0.35</td>
<td>0.35</td>
<td>0.35</td>
</tr>
<tr>
<td>TasC</td>
<td>0.51</td>
<td>0.52</td>
<td>0.53</td>
<td>0.54</td>
<td>0.54</td>
</tr>
</tbody>
</table>

(b) Cross-project suggestion accuracy comparison

<table>
<thead>
<tr>
<th>Top-1</th>
<th>Top-2</th>
<th>Top-5</th>
<th>Top-10</th>
<th>Top-20</th>
<th>TasC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact</td>
<td>0.22</td>
<td>0.23</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>Similar</td>
<td>0.35</td>
<td>0.37</td>
<td>0.38</td>
<td>0.38</td>
<td>0.39</td>
</tr>
<tr>
<td>TasC</td>
<td>0.45</td>
<td>0.46</td>
<td>0.46</td>
<td>0.46</td>
<td>0.47</td>
</tr>
</tbody>
</table>

using training data from individual projects gives better accuracy for all top ranks than using data from other projects. We also observed this result consistently in all projects in our dataset. A paired Wilcoxon test to compare the distributions of accuracy over projects between two settings shows that the difference is statistically significant.

This result implies that the task context captured by topic modeling with LDA is local in space: tasks/topics are not shared among different projects. Adding data from different projects might not improve the suggestion quality. In contrast, it increases complexity and yet could add noise to the task inference, thus, reducing accuracy.

6.5.5 Change Suggestion Accuracy Comparison with Base Models

In this experiment, we aim to answer the question if our model using task context improves the suggestion quality over the base models that use only repeated changes and do not use context information. Those base models also use the suggestion algorithm in Figure 6.5. However, they do not use topic modeling result to compute similarity in finding the candidate changes. Instead, the first base model, named Exact, uses all the changes whose source fragments are exactly matched to the given source $s$ (i.e., their normalized ASTs are isomorphic). In the second base model, named Similar, the similarity of two fragments is measured via the similarity between their respective syntactic code tokens (after normalization). Specifically, the similarity is measured as the ratio between the length of the longest common sub-sequence of the two code sequences and the maximum length of their sequences. The similarity threshold is set to be 0.8 which is the same as that for task similarity.

The result is shown in Table 6.2. The first base model misses many cases and achieves no more than 22% for top-1 suggestion. The reason is that exact matching in finding candidate changes would be too strict. That is why when we use the similar matching in the second base model, the accuracy increases more than 150% relatively.

Importantly, our model using task context relatively improves much over both the base models: more than 250% over Exact model and almost 130% over Similar model. The large improvement is observed consistently for all top-$k$ accuracy in both within-project and cross-project settings. This improvement
could be attributed to the use of topic modeling to capture a higher level of abstraction in the tasks of the code changes. We will show some examples to demonstrate this in Section 6.5.8.

Comparing between within- and cross-project settings, we can see that TasC achieves better accuracy in the former than in the latter. In contrast, the base models achieve better accuracy in the latter than in the former. While adding more change data from other projects introduces noise to task inference and reduces the accuracy in TasC, using more changes in the base models increases the chance that a test change has been seen in the past, thus, reduces the number of missing cases and increases the accuracy.

6.5.6 Fix Suggestion Using Task Context

We also performed experiments on bug fixing changes to see how our model works for this special change type. The accuracy is shown in Figure 6.8. Similarly to the general changes, the fix suggestion accuracy is higher in within-project setting than in cross-project setting. Comparing between fixes and general changes, fix suggestion accuracy is lower than change suggestion accuracy in within-project setting. However, fix suggestion accuracy is higher in cross-project setting. This result implies that the fixing tasks are more likely to be repeated across projects than within a project, while the general change tasks are more likely to be repeated within a project than across projects.

6.5.7 Task Context versus Structural and Co-Change Contexts

In this experiment, we compare the suggestion quality between our model with task context and the models using two existing types of contexts: 1) structural context (e.g., used in FixWizard Nguyen et al. (2010b)), and 2) co-change context (e.g., used in Ying et al. Ying et al. (2004) and Zimmermann et al. Zimmermann et al. (2004)). Let us briefly explain the concepts and ideas of using those contexts and then show the comparison results.
6.5.7.1 Some concepts

Definition 6.5 (Structural Context) The structural context of a code fragment is the set of code fragments that contain it. The structural context of a code change is the structural context of the source fragment of the change.

The structural context captures the context of the surrounding code of a change. This context is a set due to the nesting structure of syntactic units, i.e., a fragment can be nested in more than one fragments. Since we extract only the changes at the statement level, the structural context of a change is also the statements surrounding the source of the change. The context statements are also normalized and collapsed in the same manner as in code change extraction. In the example in Figure 6.1, the structural context of the method call is the containing if and while statements. The ASTs of their source fragments are shown in Figures 6.3a and 6.3b.

In this work, we aim to compare our model with the co-change context at the finer granularity. Thus, we define the co-change context as follows.

Definition 6.6 (Co-change Context) The co-change context of a code change is the set of changes that occur in the same transaction with the change.

The idea of using this context is that changes might often go together. Then, given a change co in the same transaction with the test change, candidate changes that have co-appeared with co in the past will be more likely to be the actual suggested change.

6.5.7.2 Using other contexts

Using structural context. We add structural context to the base model Similar to build model Structure as follows. If among the candidate changes \{c = (u, v), Sim(u, s) ≥ threshold\}, there exist changes that share structural context with the given source s, we will keep only those changes. That is, we will skip all the changes that do not share structural context with s. Otherwise, the candidate changes will be the same as in model Similar. A change c = (u, v) shares structural context with s if the set of code fragments as the structural context of u overlaps with that of s. That is, at least one ancestor code fragment of u is exactly matched with some ancestor fragment of s. The scoring and ranking schemes are the same as in the model Similar.

Using co-change context. In the model Co-change, we assume that we are given all other changes in the same transaction with the change under suggestion. Then, if we find the candidate changes that have co-occurred with a change in the same transaction, i.e., sharing the co-change context with the
change to be suggested, we keep only those changes as the candidates. Otherwise, the candidate changes will be the same as in the model Similar. The scoring and ranking schemes are the same as in the model Similar.

We also investigated the combination of those two contexts and the task context. Our expectation is that adding structural and/or co-change contexts will push the actual target fragments up in the ranked list, thus, could improve the accuracy. We combined the task and structural contexts to create the model named Task+Struct, and combine the task and co-change contexts to create the model named Task+Co. The method to add each context to our original task model is the same as the method to add each context to model Similar that was described above.

Finally, we combine all three contexts to create the model named All. If we find the candidate changes that share either structural or co-change context with the change to be suggested, we will keep only those changes as the candidates. Otherwise, the candidate changes will be the same as in the model TasC.

6.5.7.3 Comparison results

The result is shown in Table 6.3 for general changes and in Table 6.4 for fixes. For both types of changes and in both settings, our model TasC outperforms the structural and co-change models. The differences at the top-1 accuracy in which our model improves the accuracy almost 130% relatively. This trend is consistent for all top-\(k\) accuracy. Some case studies where using task context could correctly suggest while using other contexts could not will be shown in Section 6.5.8.

Comparing the models with combined contexts and our model TasC, we see that adding other contexts does not improve the accuracy. We investigated the reason for this by examining the sets of candidate changes from different models. We observed that the number of candidates that share the structural or co-change context is much smaller than the number of those that do not. It means that most of the time, those models behave the same as the TasC model (without adding other contexts). Among the candidates that share other contexts, the number of choices for target fragments is small, mostly one, which means that most of them have been seen only once in the past. This makes most of the suggestions from those candidates are very close to those from task-only model.

6.5.8 Case Studies

This section will show some cases where TasC correctly suggests at top-1 of the ranked list while the other models could not.
Table 6.3: Change suggestion accuracy comparison between using task context and using other contexts

(a) Within-project suggestion for general changes

<table>
<thead>
<tr>
<th></th>
<th>Top-1</th>
<th>Top-2</th>
<th>Top-5</th>
<th>Top-10</th>
<th>Top-20</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single context</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaskC</td>
<td>0.51</td>
<td>0.52</td>
<td>0.53</td>
<td>0.54</td>
<td>0.54</td>
</tr>
<tr>
<td>Structure</td>
<td>0.32</td>
<td>0.34</td>
<td>0.35</td>
<td>0.351</td>
<td>0.35</td>
</tr>
<tr>
<td>Co-change</td>
<td>0.33</td>
<td>0.34</td>
<td>0.35</td>
<td>0.351</td>
<td>0.35</td>
</tr>
<tr>
<td><strong>Combined context</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task+Struct</td>
<td>0.51</td>
<td>0.52</td>
<td>0.53</td>
<td>0.54</td>
<td>0.54</td>
</tr>
<tr>
<td>Task+Co</td>
<td>0.50</td>
<td>0.52</td>
<td>0.53</td>
<td>0.53</td>
<td>0.54</td>
</tr>
<tr>
<td>All</td>
<td>0.50</td>
<td>0.52</td>
<td>0.53</td>
<td>0.53</td>
<td>0.54</td>
</tr>
</tbody>
</table>

(b) Cross-project suggestion for general changes

<table>
<thead>
<tr>
<th></th>
<th>Top-1</th>
<th>Top-2</th>
<th>Top-5</th>
<th>Top-10</th>
<th>Top-20</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single context</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaskC</td>
<td>0.45</td>
<td>0.46</td>
<td>0.46</td>
<td>0.46</td>
<td>0.47</td>
</tr>
<tr>
<td>Structure</td>
<td>0.35</td>
<td>0.37</td>
<td>0.38</td>
<td>0.38</td>
<td>0.39</td>
</tr>
<tr>
<td>Co-change</td>
<td>0.36</td>
<td>0.37</td>
<td>0.38</td>
<td>0.38</td>
<td>0.39</td>
</tr>
<tr>
<td><strong>Combined context</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task+Struct</td>
<td>0.45</td>
<td>0.46</td>
<td>0.46</td>
<td>0.46</td>
<td>0.47</td>
</tr>
<tr>
<td>Task+Co</td>
<td>0.44</td>
<td>0.45</td>
<td>0.46</td>
<td>0.46</td>
<td>0.47</td>
</tr>
<tr>
<td>All</td>
<td>0.44</td>
<td>0.45</td>
<td>0.46</td>
<td>0.46</td>
<td>0.47</td>
</tr>
</tbody>
</table>

Table 6.4: Fix suggestion accuracy comparison between using task context and using other contexts

(a) Within-project suggestion for fixing changes

<table>
<thead>
<tr>
<th></th>
<th>Top-1</th>
<th>Top-2</th>
<th>Top-5</th>
<th>Top-10</th>
<th>Top-20</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single context</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaskC</td>
<td>0.49</td>
<td>0.51</td>
<td>0.52</td>
<td>0.52</td>
<td>0.52</td>
</tr>
<tr>
<td>Structure</td>
<td>0.23</td>
<td>0.24</td>
<td>0.26</td>
<td>0.26</td>
<td>0.27</td>
</tr>
<tr>
<td>Co-change</td>
<td>0.24</td>
<td>0.24</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</td>
</tr>
<tr>
<td><strong>Combined context</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task+Struct</td>
<td>0.49</td>
<td>0.51</td>
<td>0.52</td>
<td>0.52</td>
<td>0.52</td>
</tr>
<tr>
<td>Task+Co</td>
<td>0.48</td>
<td>0.50</td>
<td>0.51</td>
<td>0.51</td>
<td>0.52</td>
</tr>
<tr>
<td>All</td>
<td>0.48</td>
<td>0.50</td>
<td>0.51</td>
<td>0.51</td>
<td>0.52</td>
</tr>
</tbody>
</table>

(b) Cross-project suggestion for fixing changes

<table>
<thead>
<tr>
<th></th>
<th>Top-1</th>
<th>Top-2</th>
<th>Top-5</th>
<th>Top-10</th>
<th>Top-20</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single context</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaskC</td>
<td>0.48</td>
<td>0.48</td>
<td>0.48</td>
<td>0.49</td>
<td>0.49</td>
</tr>
<tr>
<td>Structure</td>
<td>0.32</td>
<td>0.33</td>
<td>0.34</td>
<td>0.35</td>
<td>0.35</td>
</tr>
<tr>
<td>Co-change</td>
<td>0.32</td>
<td>0.33</td>
<td>0.34</td>
<td>0.35</td>
<td>0.35</td>
</tr>
<tr>
<td><strong>Combined context</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task+Struct</td>
<td>0.48</td>
<td>0.48</td>
<td>0.48</td>
<td>0.49</td>
<td>0.49</td>
</tr>
<tr>
<td>Task+Co</td>
<td>0.47</td>
<td>0.48</td>
<td>0.48</td>
<td>0.48</td>
<td>0.49</td>
</tr>
<tr>
<td>All</td>
<td>0.47</td>
<td>0.48</td>
<td>0.48</td>
<td>0.48</td>
<td>0.49</td>
</tr>
</tbody>
</table>

Figure 6.9 shows the first one which is in project SWGAide, a utility for players of SWG. The test is a change at revision 802. For each change, the upper code fragment is the source and the lower one is the target. In this case, our task model found a candidate change at revision 728 that contains the correct target. The base model Exact could not suggest any target because this source fragment had never appeared before. The other base model Similar could find some candidates in the past changes but none of them contain the correct target. It missed the candidate in Figure 6.9 because the two source fragments are too much different in terms of code token sequence: one calls the check isEmpty and one checks size against 0. However, those two checks are actually alternative usages for checking if the set (SWGResourceSet) is empty or not. Both of them are identified by LDA as contributing very similarly to the tasks in the past changes. The concrete values are (3 = 0.014, 7 = 0.007) for isEmpty and (3 = 0.015, 7 = 0.011) for size. In each pair of numbers, the left is the task and the right the probability/contribution of the token in that task. Thus, even two code fragments look quite different, they are still considered similar in terms of tasks.

The second case is a test in project ONDEX, an open source framework for text mining, data integration and data analysis (Figure 6.10). Again, the base models could not find this candidate because
the code of two source fragments is different: one uses modifier `final` primitive type `int` and one uses class `Integer` with additional keyword `new` for class instantiation. However, the tokens `final`, `int` and `Integer` appear in all over places for all the tasks, thus, their contributions to tasks are very low. The concrete values for them are (1 = 0.008, 8 = 0.008, 10 = 0.057) for `final`, (8 = 0.002, 9 = 0.001, 10 = 0.002) for `int`, and (6 = 0.001, 9 = 0.001) for `Integer`. Thus, they do not affect the task similarity between two sources. `TasC` could match two sources and suggest the correct target.

### 6.5.9 Threats to Validity

We conducted our empirical evaluation with open-source Java projects repositories. Thus, the results could not be generalized for closed-source projects or the projects written in other languages. There are also many datasets using other version control systems and/or hosted on other hosting services that we have not covered. We plan to extend our evaluation to include projects hosted on GitHub and written in C/C++ in the future work. Our comparison suffers from the threat that the methods we used to integrate the context might not be the most suitable ones.

### 6.6 Related Work

There are automated approaches to suggest a fix or a change for a given code fragment. Such suggestion is in the context of automated **program repairing**. GenProg Goues et al. (2012) is a patch generation method that is based on genetic programming. To evolve a program variant through transformation, it reuses the existing program statements in the current program and creates the combinations. However, it does not consider the change/fixing history in the process. PAR Kim et al. (2013), another program auto-repair technique, derives a patch by mining fixing patterns from prior human-written patches. To derive a patch, it does not use the task context in a fixing history. FixWizard Nguyen et al.
(2010b) suggests a fix for a given fragment based on the similarity of that fragment and other previously fixed code. The similarity is defined based on similar code structures and/or similar API usages. Ray et al. Ray and Kim (2012); Ray et al. (2012)’s Repertoire is a tool to identify ported edits between patches in forked projects. They compare the content of individual patches. LASE Meng et al. (2013) is a tool to automate similar changes from examples. It creates context-aware edit script, identifies the locations and transforms the code. Similar to FixWizard, Repertoire and LASE are based on comparing the code and apply the changes from one place to another. In our prior work Nguyen et al. (2013b), we leverage repeated changes and fixes to suggest a change/fix for a given code fragment. In comparison, in that work, the suggestion tool reply solely on the repeated changes recorded in the past or in other projects. None of those existing approaches consider the task context in recent history.

Ying et al. Ying et al. (2004) and Zimmermann et al. Zimmermann et al. (2004) propose approaches to suggest a co-change at the file and function levels. Ying et al. examine the co-changing files in the history and use association rule mining algorithm to find frequently co-changed files. From such information, they predict possible files for changing when given a newly changed file. Zimmermann et al. support not only co-changes at the file level, but also at the function and field level. They also use association rule mining algorithm with support and confidence. There are two key differences between TasC and those approaches. First, we aim to suggest at the finer-grained level of AST. Second, we use a statistical approach, rather than deterministic pattern mining algorithms in those approaches. Giger et al. Giger et al. (2012) predict type of changes, e.g., condition changes, interface modifications, inserts or deletions of methods and attributes, or other kinds of statement changes. They use the types and code churn for bug prediction Giger et al. (2011).

Other approaches detect the patterns of changes to support software maintenance. SemDiff Dagenais and Robillard (2008) mines the updating patterns to a framework to support automated updating for its client code. MAM Zhong et al. (2010) mines common graph transformations representing the code after migration to learn migration rules. Similarly, LibSync Nguyen et al. (2010a) learns adaptation patterns from client code and uses them to update a program to use a new version of a library. Negara et al. Negara et al. (2014) detect frequent change patterns by using closed frequent itemset mining on a sequence of changes expressed as tree editing operations.

Our prior empirical study Nguyen et al. (2013b) on a large scale data set has shown that code changes are repetitive. Barr et al. Barr et al. (2014) reported a high degree of graftability, independent of size and type of commits. They also found that changes are 43% graftable from the exact version of the software being changed. Gabel and Su Gabel and Su (2010) found code also contains much syntactic redundancy: at the level of granularity with 6 tokens, 50–100% of each project is redundant.
6.7 Conclusion

In this chapter, we propose TasC, a novel statistical model that uses the task context of changes in software development history to suggest fine-grained code change/fix at the program statement level. We use latent Dirichlet allocation (LDA) to capture the task context where topics are used to model change tasks. We also propose a novel technique for measuring the similarity of code fragments and code changes using the task context.

We conducted an empirical evaluation on a large dataset of 88 open-source Java projects containing more than 200 thousand source files and 3.5 million source lines of code (SLOCs) in their last revisions. Our result shows that TasC improves the suggestion accuracy relatively up to 130%–250% in comparison with the base models that do not use contextual information. Compared with other types of contexts, it outperforms the models using structural and co-change contexts.
CHAPTER 7. GRAPH-BASED API ADAPTATION

Reusing existing library components is essential for reducing the cost of software development and maintenance. When library components evolve to accommodate new feature requests, to fix bugs, or to meet new standards, the clients of software libraries often need to make corresponding changes to correctly use the updated libraries. Existing API usage adaptation techniques such as CatchUp! or SemDiff support simple adaptation such as replacing the target of calls to a deprecated API, but cannot handle complex adaptations such as creating a new object to be passed to a different API method, and adding an exception handling logic that surrounds the updated API calls.

This chapter presents LibSync that guides developers in adapting API usage code by learning complex API usage adaptation patterns from other clients that migrated to a new library version already (and also from the API usages within the library’s test code). LibSync uses several graph-based techniques (1) to identify changes to API declarations by comparing two library versions, (2) to extract associated API usage skeletons before and after library migration, and (3) to compare the extracted API usage skeletons to recover API usage adaptation patterns. Using the learned adaptation patterns, LibSync recommends the locations and edit operations for adapting API usages. The evaluation of LibSync on real-world software systems shows that it is highly correct and useful with a precision of 100% and a recall of 91%.

7.1 Introduction

Reusing existing software components by accessing their implementations through their Application Programming Interfaces (APIs) can reduce the cost of software development and maintenance. When libraries provide their functionality through public interfaces (e.g., types, methods, and fields in Java), clients are expected to respect the contract assumed by the libraries by using the correct names of the APIs, passing the right arguments, following the intended temporal orders of API invocations, etc.

When library components evolve to accommodate new feature requests, to fix bugs, and to meet new standards, changes in API declarations in libraries could cause existing clients to break. For example,
when the names of APIs change through renaming and moving of the APIs, client code may not compile. When an API signature modification requires more input parameters or a different return type, clients need to pass additional input arguments or to process a returned object differently.

Existing analysis techniques that can be used for adapting API usage code in client applications have the following limitations. First, existing research techniques such as CatchUp! Henkel and Diwan (2005) and MolhadoRef Dig et al. (2007) require library maintainers and client application developers to use the same development environment to record and replay refactorings. Other techniques require library developers to manually write expected adaptations in client code as rules Chow and Notkin (1996). Second, existing API usage modeling and extraction techniques Wasylkowski et al. (2007); Engler et al. (2001); Acharya et al. (2007); Williams and Hollingsworth (2005) are limited by simplified representations such as a sequence of method calls. Thus, they cannot capture the complex control and data dependencies surrounding the use of APIs. For example, SemDiff Dagenais and Robillard (2008) models API usages in terms of method calls, so it can support changing the target of calls to modified APIs but cannot add the control structure that surrounds the calls to a new replacement API.

Hypothesizing that changes to API usage caused by the evolution of library components may involve complex changes, we developed a set of graph-based models and algorithms that can capture updates in evolving libraries and updates in client applications that are associated with changes in the libraries, and an algorithm that generalizes common edit operations from a set of API usage code fragments before and after library migration.

Our API usage code adaptation framework takes as input the current version of a client application, both the old version and the new version of a library under focus, and a set of programs that already migrated to the new library version. Our framework consists of four main components: (1) an ORIGIN ANALYSIS TOOL (OAT) that maps corresponding code elements between two versions, (2) a CLIENT API USAGE EXTRACTOR (CUE) that extracts API usage skeletons from client code and the use of APIs within the library, (3) an API USAGE ADAPTATION MINER (SAM) that automatically infers adaptation patterns from a set of API usage skeletons before and after the migration from the old library version to the new library version, and (4) LibSync that recommends which API usage code needs to be adapted and how those code fragments need to be updated by locating API usage fragments in the client that need to be adapted and suggesting edit operations required for adaptation.

OAT is an origin analysis to map corresponding code elements between two program versions. It is used for two different purposes: to detect changes to API declarations provided by libraries and to map corresponding API usage code fragments between two versions of client code. Details of OAT are described in Section 5.4.1.
CUE extracts the skeleton of API usage code in client systems and the test code of the APIs. We make the assumption that client systems use library components by accessing their APIs via invocation—directly calling API methods or instantiating objects from API classes—and via inheritance—declaring classes in the client by extending subtyping API classes. We call those two ways of using APIs as API i-usage (invocation) and API x-usage (extension). We also use the term API usage to refer to both types of API usages. In particular, CUE extends Nguyen et al.’s graph-based object usage model (GROUM) Nguyen et al. (2009c) that represents both control and data dependencies among method calls and field accesses. In order to capture API usage, we extended this GROUM model to explicitly note the usages of external APIs via invocation and extension by modeling the types of objects passed to APIs as arguments and by modeling overriding and inheritance relationships between client methods and methods provided by external API types. In particular, CUE represents API i-usage by an invocation-based, graph-based object usage model, iGROUM, in which, action nodes represent method invocations, control nodes represent surrounding control structures, and data nodes represent use of types provided by external libraries. The edges represent dependencies between the nodes, for example, usage orders, input and output relations, etc. CUE represents API x-usage by another graph-based model called xGROUM, in which each node represents a method-header and two kind of edges represent overriding and inheritance relationships between a method-header in the client and a method-header in the library. For example, if method $m_c$ in a client class overrides method $m_l$ in a library, then a call to $m_l$ would directly invoke $m_c$ or its overriding implementation $m'_c$ in its subtypes. Thus, when $m_l$’s declaration changes (e.g. change parameter type, add more parameter, throw exception), the declaration of $m_c$ should be changed accordingly to properly override $m_l$.

SAM uses an approximate graph alignment and differencing algorithm to map nodes between two usage models based on their similarity of labels and neighborhood structures, and calculates the editing operations based on the alignment. For example, the aligned nodes having different attributes are considered as replaced or updated, while unaligned nodes are considered as deleted or added. Since API usage changes are detected as change sets of editing operations, SAM mines the frequent subsets of such change sets using a frequent item set mining algorithm Agrawal and Srikant (1994) and considers them as API usage adaptation patterns.

LibSync has a knowledge base of API usage adaptation patterns for each library version. Given a client system and the desired version of library to migrate to, LibSync identifies the locations of API usages in the client system that are associated with changed APIs. It then matches each usage with the best matched API usage pattern in its knowledge base and derives the edit operations for adapting API usages.
We have conducted an empirical evaluation of *LibSync* on three large open-source subject systems, each uses up to 300 libraries. We have done several experiments to evaluate the correctness and usefulness of our tool in two usage scenarios: (a) API usage adaptation in different locations within a client program, and (b) adaptation in different branches of a client program (e.g. back-porting). The evaluation shows that *CUE* detect changes to API usages with precision of 93%, and *LibSync* provides useful recommendation in most cases, even when API usage adaptation involves complex changes to the control logic surrounding API usages.

The key contributions of the chapter include:

1. *CUE*, a graph-based representation that models the context of API usages by capturing control and data dependencies surrounding API usages; in particular, it supports API usages via method *invocations* and *subtyping* of the API types provided by an external library under focus.

2. *SAM*, a graph alignment algorithm that identifies API usage changes in client applications and an API usage adaptation pattern mining algorithm that generalizes common edit operations from multiple API usage skeletons before and after migration.

3. *LibSync*, a tool that takes as input a client system and a given library version to be migrated to, and recommends the locations and edit operations for adapting API usage code in the client.

4. An empirical evaluation for the correctness and usefulness of *LibSync* in adapting API usage code.

Section 7.2 presents motivating examples that require complex API usage adaptation in client applications that are caused by updates to libraries. Sections 7.3, 7.4, and 7.5 detail individual models and algorithms that we have developed to build *LibSync*. Section 7.6 shows the empirical evaluation of *LibSync*. Section 7.7 describes related work and Section 7.8 summarizes this chapter’s contributions.

### 7.2 Motivating Examples

jBoss subversion (a) is a large project that has been developed more than 6 years ago, with 47 releases. It has about 40,000 methods and uses up to 262 different libraries. Using the Subversion subversion (b) and code search functionality in Eclipse, we manually scanned the version history of jBoss and the external libraries used by jBoss. We examined more than 200 methods that changed due to the modification of external APIs based on associated documentation, change logs, and bug reports. This section presents representative API usage adaptation examples that motivate our graph-based technique for supporting API usage adaptations.
XYSeries set = new XYSeries(attribute, false, false);
for (int i = 0; i < data.size(); i++)
    set.add(new Integer(i), (Number)data.get(i));
DefaultTableXYDataset dataset = new DefaultTableXYDataset(set, false);
dataset.addSeries(set);
JFreeChart chart = ChartFactory.createXYLineChart(..., dataset, ...);

Figure 7.1: API usage adaptation in jBoss caused by the evolution of jFreeChart

7.2.1 Examples of API usage via method invocations

Figure 7.1 illustrates an API usage adaptation example in jBoss with respect to the use of jFreeChart library. The code changes are represented with added code and deleted code. The changes from jBoss version 3.2.7 to 3.2.8 were due to the modification of external APIs, XYSeries and DefaultTableXYDataset, in jFreeChart from version 0.9.15 to 0.9.17. To enable a new auto-sorting feature, the XYSeries constructor with two input arguments is deprecated and a new constructor with three input arguments is provided instead. The DefaultTableXYDataset constructor that accepts XYSeries as an input is also deprecated. The new constructor accepts a boolean input to activate the new auto-pruning feature for data points in DefaultTableXYDataset. It implies that the XYSeries object must be added after the initialization of the DefaultTableXYDataset object. Thus, the parameter set is replaced by a value false, and a call to DefaultTable XYDataset.addSeries is added. This example illustrates the following:

1. jBoss uses jFreeChart via creating objects from API classes (e.g. XYSeries, DefaultTableXYDataset) and calling API methods (e.g. DefaultTableXYDataset.addSeries, ChartFactory.createXYLineChart). Since an object instantiation is represented as a constructor call to an external API type, we consider this type of API usage as usage via invocation.

2. API usage must follow specific protocols due to the dependencies between API elements. For example, a DefaultTableXYDataset object needs to be created before any XYSeries object could be added to the set. A chart needs to be created with an object of Dataset.

3. As API evolves, such usage protocols could change, requiring corresponding API usage adaptations. For example, the calls to deprecated methods are replaced with newly provided ones, or new method calls are added, etc. In this example, the following edit operations occurred in the client code: replacement (e.g. the constructor of XYSeries), addition (e.g. DefaultTable XYDataset.addSeries), and update of input/output dependencies (e.g. the object XYSeries no longer immediately depends on DefaultTableXYDataset.<init> but instead on DefaultTableXYDataset.addSeries).

This example shows that existing state-of-the-art adaptation approaches (e.g. SemDiff Dagenais and Robillard (2008), CatchUp Henkel and Diwan (2005)) could not support the API usage adaption because
SnmpPeer peer = new SnmpPeer(this.address, this.port, this.localAddress, this.localPort);
peer.setPort(this.port);
peer.setServerPort(this.localPort);

Figure 7.2: API usage adaptation in JBoss caused by the evolution of OpenNMS

<table>
<thead>
<tr>
<th>Package</th>
<th>Class</th>
<th>Method</th>
<th>Old Signature</th>
<th>New Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.apache.axis.encoding</td>
<td>Serializer</td>
<td>writeSchema</td>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>org.jboss.net.jmx.adaptor</td>
<td>AttributeSerializer</td>
<td>writeSchema</td>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>org.jboss.net.jmx.adaptor</td>
<td>ObjectNameSerializer</td>
<td>writeSchema</td>
<td>boolean</td>
<td>boolean</td>
</tr>
</tbody>
</table>

Figure 7.3: API usage adaptation in JBoss caused by the evolution of Axis

they assume that the adaptation needed in client code is simply individual method-call replacements or type declarations. They do not consider the context of API usages, the dependencies between method calls, and the differences between the extension and invocation of API methods.

Figure 7.2 shows another API usage adaptation example. From version 1.6.10 to 1.7.10, in the OpenNMS library, a new constructor with four parameters is added for initializing SnmpPeer. Such API change requires adding a call to the new constructor and the removal of two subsequent calls to setter methods as in Figure 7.2. This adaptation from JBoss version 3.2.5 to 3.2.6, although simple in meaning, is complex in term of edit operations: it involves one constructor-call replacement and two method-call deletions. Importantly, all edited calls are dependent. SemDiff Dagenais and Robillard (2008) could suggest the replacement of the old constructor call, however, it does not suggest the setter method-call deletions because it does not consider the API usage context when recommending adaptations.

7.2.2 Examples of API usage via inheritance

Figure 7.3 shows an API usage example via inheritance. The API class Serializer in the org.apache.axis.encoding package provides the writeSchema method. The AttributeSerializer class in JBoss inherits from the Serializer class and overrides the writeSchema method. When the input signature of the writeSchema is changed by requiring a Class type argument and returning Element instead of boolean, the signature of the overriding method needs to be updated accordingly to properly override the writeSchema
Change in Apache Axis API

```java
package org.apache.axis.providers.java;

class EJBProvider {

  protected Object getNewServiceObject makeNewServiceObject(...)

  ...
```

Change in jBoss

```java
package org.jboss.net.axis.server;

class EJBProvider extends org.apache.axis.providers.java.EJBProvider {

  protected Object getNewServiceObject makeNewServiceObject(...)

  ...
```

Figure 7.4: API usage adaptation in jBoss caused by the evolution of Axis method.

Figure 7.4 shows another example. Class \texttt{C=EJBProvider} in jBoss inherits the class with the same name \texttt{A=EJBProvider} in the Apache Axis library. The method method \texttt{m=getNewServiceObject} of \texttt{A} was renamed into \texttt{makeNewServiceObject}. Thus, its overriding method \texttt{C.m} is also renamed accordingly.

7.2.3 Observations

We make the following observations based on API usage adaptation examples. First, in object-oriented programming (OOP), there are two common ways to use the API functionality (1) via method invocation, directly calling to API methods or creating objects of API classes; and (2) via inheritance, declaring classes in client code that inherit from the API classes and override their methods. Second, to use APIs correctly, client code must follow specific order of method calls or override certain methods. Thus, API usage model and API usage adaptation model must capture complex context surrounding API usages: (1) data and ordering dependencies among API usages, (2) control structures around API usages, and (3) the interaction among multiple objects of different types.

Those observations imply the necessity of a recommendation tool that helps developers in complex API usage adaptation to cope with evolving libraries. The tool should provide recommendations regarding where and how to do API usage adaptation. That is, given a client program using libraries and the changes to external API declarations, the recommendation tool should identify the locations where API usage adaptations are required and suggest adaptation edit operations.

7.3 Client API Usage Extractor (CUE)

This section describes \texttt{CUE}, a client API usage extractor, that extracts the skeleton of API usage code from client, test, and demo code of the APIs. Section 7.3.1 presents the model and extraction
algorithm for API usages via invocation. Section 7.3.2 presents the model and extraction algorithm for API usages via inheritance.

7.3.1 API Usage via Invocation

7.3.1.1 i-Usage Model

An API provides the functionality via its elements. Those elements provide the computation (via methods) or the storage of data (via objects). Thus, to use a function provided by an API via invocation, a client program could call the computational elements (e.g. invoking a method) or process the data elements (e.g. initializing an object, using it as an input/output parameter). When several API methods/objects are used, the relations, e.g. the orders and dependencies, between those elements are important because they must follow the intended API usage specifications. Such usages are often related to the control structure (such as if, while) due to the branching or repetition of the computation and data processing.

CUE represents the API i-usages in clients via a graph-based model called iGROUM (invocation-based, GRaph-based Object Usage Model). In general, each usage is represented by a labeled, directed, acyclic graph, in which, the usages of API elements are represented as nodes, while the dependencies are modeled by edges. An action node represents a method invocation (i.e. a usage of an API computation element). A data node represents an object (i.e. a usage of an API data element). The label of each node is the fully qualified name and the signature of the corresponding method or class. An edge from an action node to another node represents the control and data dependencies. An edge from a data node to an action node shows that the corresponding object is used as an input of the corresponding call. Similarly, an edge with the opposite direction shows an output relation. Action nodes have some attributes to represent their input signature (e.g. a list of parameter types, modifiers, exceptions that could be thrown, a return type).

Definition 7.1 (iGROUM) An invocation-based, graph-based object usage model is a directed, labeled, acyclic graph in which:

1. Each action node represents a method call;
2. Each data node represents a variable;
3. Each control node represents the branching point of a control structure (e.g. if, for, while, switch);
4. An edge connecting two nodes $x$ and $y$ represents the control and data dependencies between $x$ and $y$; and
5. The label of an action, data, control, and operator node is the name, data type, or expression of the corresponding method, variable, control structure, or operator, along with the type of the corresponding node.

Using this model, CUE represents the usage of computation and data API elements (via action and data nodes), the use of control structures (via control nodes), and the control and data dependencies between them (e.g. orders and conditions among calls, inputs/outputs, use of shared data).

Figure 7.5 shows two graph-based API usage models extracted from the code in Figure 7.1. The usage changes between two models are illustrated by the gray nodes with bold edges. For simplicity, in the figure, a label is displayed with only class and method names, even though our model actually retains the fully qualified class name and the signature of a method. In Figure 7.5b, an edge from the action node \( y' = \text{DefaultTableXYDataset.<init>} \) to the action node \( z' = \text{DefaultTableXYDataset.addSeries} \) represents that \( y' \) is used before \( z' \). An edge from the action node \( x = \text{XYSeries.<init>} \) to the data node \( s = \text{XYSeries} \) shows that \( s \) is used to store the output of \( x \). An edge coming out of \( s \) changes its target from \( y \) to \( z' \). That means, \( s' \) is now used as an input to \( z' \) instead of \( y' \). Note that \( x \) and \( x' \) represent different API elements – \( x \) is a deprecated constructor with two parameters while \( x' \) is a new constructor with three parameters. The figure also shows a for loop related to the invocation of method XYSeries.add.

7.3.1.2 i-Usage Extraction

CUE extends our prior work (Nguyen et al.’s graph-based object usage model extraction Nguyen et al. (2009c)) to build API usage models from each method in client code. It parses the source code into Abstract Syntax Trees (AST), traverses the trees to analyze the AST nodes of interest such as
method invocations, object declarations and initializations, and control statements (e.g. if, while, for) within a method and builds the corresponding action, data, and control nodes along with control and data dependencies between them. Static methods, type casting, and type checking operations of a class are considered as special invocations of the corresponding objects. After extraction, CUE removes all action and data nodes and the edges that do not represent the usages of API elements or have no dependencies with those API elements. In other words, CUE determines a sub-graph of the original object usage model that is relevant to the usage of API elements by performing program slicing from the API usage nodes via control and data dependency edges. Moreover, since a particular API could be used by multiple methods in client, CUE uses a set of iGROUM models to represent API i-usage.

While building an iGROUM, CUE also takes into account subtyping information, which is described further in detail in Section 7.3.2. CUE uses the inheritance information of the system to create nodes and labels more precisely. For example, if a method C.m is called in an iGROUM, CUE checks whether C.m is inherited from a method A.m, i.e., C.m is not explicitly declared in the body of the class C. If that is the case, the action node corresponding to the call would be a node with the label built from A.m, rather than from C.m. If C.m overrides A.m, the label is built from C.m.

Furthermore, CUE also performs a simple intra-procedural analysis on object instantiation, assignment, and type casting statements to keep track of the types of variables used within a method. For example, if it encounters a method call o.m with o being an object declared with type C, and later finds that o is casted into an object of class C’, then the label of action node for o.m is built from C’.m, rather than C.m.

7.3.2 API Usage via Inheritance

This section presents our graph-based representation for API usages via inheritance and the corresponding extraction procedure.

7.3.2.1 Method Overriding and Inheritance

Assume that class C in a client code directly inherits from an API class A. Method C.m overrides a non-static method A.m if C.m is declared in class C and has the same signature with A.m. In OOP, method A.m is not considered to be overridden in C when the method C.m with the same signature as A.m is not explicitly declared in C. However, because C.m could still be invoked, CUE still considers that C.m exists and inherits from A.m. If A.m and C.m are static, CUE does not consider that C.m overrides A.m because they are called based on the declaring types. If A.m is static and C.m is not explicitly declared in C, CUE does not consider the existence of C.m.
1. If $C.m$ inherits $A.m$, a call to $C.m$ will be a call to $A.m$. Thus, if $A.m$ is changed, not only the calls to $A.m$ need to be adapted in response to the change of $A.m$, but also all the calls to $C.m$ need to be considered for adaptation. For example, if $A.m$ has a newly added parameter, all method calls to $A.m$ and $C.m$ must be considered for the adaptation of adding a new parameter. Otherwise, the program might not be compiled or such calls might be accidentally dispatched as a call to another method that has the same-signature as $A.m$ (e.g. a parent method $A.o.m$ of $A.m$).

2. If $C.m$ overrides $A.m$, they need to have the same signature. Thus, if $A.m$ is changed, $C.m$ needs to be considered to be changed correspondingly (see Figures 7.3 and 7.4).

3. A call to $A.m$ might be a call to $C.m$ in run-time due to dynamic dispatching. Thus, if $C.m$ is changed, not only all the calls to $C.m$ and $C_1.m$, with $C_1$ being a descendant class of $C$, are considered for adaptation correspondingly, but also all calls to $A.m$ must be taken into consideration.

In CUE, the overriding and inheritance relationships are defined in the same way as above among the methods of two API classes $A$ and $A_1$ in which $A_1$ inherits from $A$, and among the methods of two client classes $C$ and $C_1$ in which $C_1$ inherits from $C$.

### 7.3.2.2 x-Usage Model

Now, let us describe the model and extraction algorithm for API usages via inheritance. CUE uses xGROUM (Extension-based, GRaph-based Ojbect Usage Model) to represent all API x-usages in the
client system and all libraries by considering each library a sub-system of the client system under investigation.

An xGROUM is a directed, labeled, acyclic graph in which each node represents a class or a method in the client system and its libraries. The label of a node is its fully qualified name and signature. Interfaces are considered as special classes.

Edges between class nodes represent subtyping relations. Edges from class nodes to method nodes represent the containment relations. Between method nodes, there are two kinds of edges: o-edge (overriding) and i-edge (inheriting):

- An o-edge from a node \( C.m \) to \( A.m \) shows that \( C.m \) overrides \( A.m \). This means that \( C \) inherits from \( A \), and \( C.m \) is declared in \( C \) and has the same signature as \( A.m \).

- An i-edge from a node \( C.m \) to \( A.m \) shows that \( C.m \) inherits from \( A.m \). This means that \( C \) inherits from \( A \), and \( C.m \) is not explicitly declared in \( C \) even though \( C.m \) could be invoked and has the same signature as \( A.m \). \( C.m \) is called an i-node in xGROUM, and other method nodes are called o-nodes.

Figure 7.6 illustrates the xGROUM for Figure 7.4. Figures 7.6L and 7.6C show the API class \( A = \text{EJBProvider} \) in package \( \text{org.apache.axis.providers.java} \) and the client class \( C = \text{EJBProvider} \) in package \( \text{org.jboss.net.axis.server} \). Figures 7.6L’ and 7.6C’ show those two classes in their new versions. The o-edges and o-nodes, such as \( C.\text{getNewServiceObject} \), are illustrated with solid double lines, meaning that they are of interest. The i-nodes such as \( C.\text{getEJBHome} \) and the i-edges are shown in dashed lines, meaning that they are just placeholders and not being really declared or created.

Added nodes such as \( A.\text{getContext(Properties)} \) are painted in gray color. Updated nodes are represented in double lines along with bi-directional arrows between them in the graphs of two versions. For example, an arrow with the label rename from node \( A.\text{getNewServiceObject} \) (in Figure 7.6L) to \( A.\text{makeNewServiceObject} \) (in Figure 7.6L’) shows that those two nodes represent a renamed method. An arrow from node \( A.\text{getServiceClass(Context, String)} \) in Figure 7.6L to node \( A.\text{getServiceClass(String, SOAP, Context)} \) in Figure 7.6L’ signifies the change in the parameter list of the corresponding method. As shown in Figure 7.6 C’, class \( C \) in jBoss is adapted accordingly to the changes to class \( A \) in Axis.

To build xGROUM, CUE extends the inheritance hierarchy by adding o-edges between methods. The i-nodes and i-edges are not explicitly created, but being computed on demand. Note that only one xGROUM is built for the entire client system and its libraries.
function GroumDiff(U, U') //align and differ two usage models
for all u ∈ U, v ∈ U' //calculate similarity based on label and structure
sim(u, v) = α • lsim(u, v) + β • nsim(u, v)
M = MaximumWeightedMatching(U, U', sim) //matching
for each (u, v) ∈ M:
    if sim(u, v) < λ then M.remove((u, v)) //remove too low matches
else switch // derive change operations on nodes
    case Attr(u) ≠ Attr(v): Op(u) = "replaced", Op(v) = "replaced"
    case Attr(u) = Attr(v), nsim(u, v) < 1: Op(u) = "updated"
    default: Op(u) = "unchanged"
for each u ∈ U, u ∉ M: Op(u) = "deleted" //un-aligned nodes are
for each v ∈ U', v ∉ M: Op(v) = "added" //deleted or added
Ed = EditScript(Op)
return M, Op, Ed

Figure 7.7: API Usage Graph Alignment Algorithm

7.4 Usage Adaptation Miner (SAM)

Our approach uses iGROUMs to represent API i-usages in any client code as well as in the library’s test and demo code. Thus, the adaptation of API usages could be modeled as a generalization of changes to the corresponding individual iGROUMs. This section describes our API usage adaptation miner, SAM, that uses a graph alignment algorithm to identify API i-usage changes that are caused by changes to APIs, and a mining algorithm that generalizes common edit operations from multiple API usage changes to find API usage adaptation patterns. LibSync uses such patterns to recommend the locations and edit operations.

7.4.1 i-Usage Change Detection

Using OAT (Section 5.4.1), given two versions i and i' of a client program P, LibSync derives sets ∆L and ∆P containing the changed entities (including packages, classes, methods) of the library and the client program respectively. It is able to align such code entities between two versions as well. Thus, for any method m ∈ ∆P, LibSync builds two iGROUMs U and U' for m in two corresponding versions. Then, it uses GroumDiff, our graph-based alignment and differencing algorithm, to find the changes between the corresponding usage models U and U'.

Our graph alignment algorithm, GroumDiff, maps the nodes between two iGROUMs such that the aligned nodes represent the unchanged, updated, or replaced nodes while unmapped nodes represent the added/deleted nodes. Let us illustrate the pseudo-code of our GroumDiff algorithm (Figure 7.7) via the example in Figure 7.5. The mapped nodes in Figure 7.5 would be the ones with identical labels (e.g. x and x', y and y'). New nodes like z', b1, b2 would not be mapped. As we could see, mapped nodes tend to have or highly similar labels and structures. For example, unchanged API elements would
have identical names; replaced ones tend to have similar names; and both types tend to have similar
neighborhood structure with the others.

The idea of GroumDiff algorithm is to map the nodes between two graphs based on the similarity of
their labels and neighborhood structures with other nodes. The similarity of node labels, \( lsim(u, v) \), is
based on string-based Levenshtein distance Hunt and Szymanski (1977). It takes into account also the
renamed API elements: the similarity level of the labels representing renamed or moved API elements is
set as high as that for unchanged ones. Neighborhood structures of nodes is approximated by Exas char-
acteristic vectors Nguyen et al. (2009a), thus, their similarity measurement \( nsim(u, v) \) is based on the
distance of such vectors. GroumDiff calculates and combines the similarity of labels and neighborhood
structures of all pairs of nodes \( u, v \) between two graphs as

\[
sim(u, v) = \alpha \cdot lsim(u, v) + \beta \cdot nsim(u, v)
\]

(line 3). Since each node \( u \) in a graph should be mapped to only one node \( v \) that has the highest pos-
sible similarity, GroumDiff finds the maximum weighted matching on such nodes using the calculated
similarity values as weights (line 4). The resulting pairs of matched nodes are the alignment results
(lines 7-10). Matched nodes having little similarity would be reported as unmapped nodes (lines 11-12).

Then based on the alignment results, SAM derives a sequence of graph edit operations. That is, the
un-aligned (un-mapped) nodes are considered added/deleted (lines 11-12). Aligned nodes with different
labels, or the same labels but different structures or attributes are considered updated/replaced (lines
8-9). Other nodes are considered unchanged (line 10). From this information, GroumDiff derives an
edit script to describe the changes as a sequence of graph operations (line 13). This edit script is then
used to mine the API usage adaptation patterns.

Let us revisit Figure 7.5. GroumDiff aligns nodes with identical names in Figures 7.5a and 7.5b.
Node \( z' = DefaultTableXYDataset.addSeries \) and two nodes, \( boolean b_1 \) and \( b_2 \), are not mapped; thus,
they are considered as added. The nodes with the label \( <init> (x \text{ and } x') \) are replaced. The node
\( s=XYSeries \) is updated because its neighboring nodes changed. Thus, the derived editing script is

Replace \( XYSeries.<\text{init}>(..., \text{boolean}) \) \( XYSeries.<\text{init}>(..., \text{boolean}, \text{boolean}) \)
Replace \( DefaultTableXYDataset.<\text{init}>(XYSeries) \) \( DefaultTableXYDataset.<\text{init}>(\text{boolean}) \)
Add \( DefaultTableXYDataset.addSeries(XYSeries) \)

**Improvement.** To improve the alignment accuracy and to deal with renamed nodes, SAM uses
OAT to find API methods and classes whose declaration changed. Then, it makes the action nodes
representing the calls to them to have the same labels in two iGROUMs under comparison. That is,
if \( m \) is updated into \( m' \) in the library through renaming, the label of an action node representing an
invocation of \( m' \) is replaced by the label built from \( m \). Note that those two nodes must also have similar
protected JFreeChart createXyLineChart() throws JRException {
    ChartFactory.setChartTheme(StandardChartTheme.createLegacyTheme());
    JFreeChart jfreeChart = ChartFactory.createXYLineChart(..., getDataset() ...,);
    return jfreeChart
}

Figure 7.8: API Usage Changes in JasperReport

neighborhoods. In brief, SAM uses the knowledge of the origin analysis result to improve the alignment of nodes in the corresponding iGROUMs.

7.4.2 x-Usage Change Detection

Changes to an xGROUM are detected by OAT and represented as editing operations: (1) Add/Delete nodes and edges: e.g., a new class is added, a method is deleted, or an overriding edge changes its target method; (2) Replace/Update nodes and edges: e.g, an edge is changed from i-edge to o-edge when a new method overrides a parent method. It is changed from o-edge to i-edge when an overriding method is deleted.

Note that when the signature of a method $C.m$ is changed into $C.m'$ that overrides some parent method $A.m'$, SAM considers this change as the addition of a new o-node for $C.m'$, the old node $C.m$ having the same signature with $A.m$ will become an i-node.

7.4.3 Usage Adaptation Pattern Mining

Given a library $L$ and a client system $P$, SAM identifies the locations and edit operations required to adapt API usages when migrating to the version $i$ of $L$. Since individual API usages can have different edits operations between two corresponding iGROUMs, our goal is to find a common subset of edit operations that occur frequently among multiple API usages; we call such frequent edit operations as an adaptation pattern.

For example, JasperReport version 3.5.0 migrated to use jFreeChart API version 1.0.12. Analyzing JasperReport’s code, we found that the addition of the invocation statement `ChartFactory.setChartTheme(StandardChartTheme.createLegacyTheme());` before the call to `ChartFactory.create*Chart` occurs in 53 methods. That is, jFreeChart at version 1.0.12 has a new feature, which specifies the style or theme of a chart object. This new feature requires that the instantiation of a chart object needs to create a `ChartTheme` object first. jFreeChart’s `ChartFactory`, the factory class for creating chart objects, now has a new method `ChartFactory.setChartTheme` to set the theme for a
1 \textbf{function} \text{ChangePattern}(\Delta P_i, \Delta L_i) \quad // \text{mining usage change patterns}
2 \textbf{for each} \((U, U') \in \text{UsageChange}(\Delta P_i, \Delta L_i) \quad // \text{compute usage changes}
3 \quad \text{Add}(\text{GroumDiff}(U, U')) \text{ into } E \quad // \text{add to dataset of sets of operations}
4 \quad F = \text{MaximalFrequentSet}(E, \sigma) \quad // \text{mine maximal frequent subset of edits}
5 \quad \textbf{for each} \ f \in F:
6 \quad \text{Find}\ U, U' : f \subseteq \text{GroumDiff}(U, U') \quad // \text{find the usages changed by } f
7 \quad \text{Extract}\ (U_o(f), U'_o(f)) \text{ from } (U, U') \quad // \text{extract reference models}
8 \quad \text{Add}\ (U_o(f), U'_o(f)) \text{ into } \text{Ref}(f) \quad // \text{add to the reference set for } f
9 \quad \text{return } F, \text{Ref}

Figure 7.9: Adaptation Pattern Mining Algorithm

chart object. jFreeChart also provides a class \texttt{StandardChartTheme} as the default implementation of \texttt{ChartTheme} which has a method named \texttt{StandardChartTheme.createLegacyTheme()} to create and return a \texttt{ChartTheme} that doesn’t apply any changes to the JFreeChart defaults.

Mining Algorithm. The algorithm to recover the API i-usage adaptation patterns is showed in Figure 7.9. It receives two inputs: a set \(\Delta L_i\) of API elements changed before or at version \(i\) and a set \(\Delta P_i\) of program entities in client code changed after migration to the version \(i\) of \(L\). \(\Delta L_i\) is computed by applying OAT to the version history of the library \(L\) backward from the version \(i\). Similarly, \(\Delta P_i\) is computed by running OAT on two versions of the client program before and after migration to \(L_i\).

The first step is to determine all API i-usages that changed due to the changes \(\Delta L_i\) (\texttt{UsageChange}(\Delta P_i, \Delta L_i)\) in line 2). This step is necessary since some i-usage changes are irrelevant to the API changes. To do that, SAM determines all methods in both \(L\) and \(P\) that are affected by the change in \(L_i\) by using the information produced from the location detection algorithm (Section 7.5.2.1 will detail this algorithm). More specifically, it uses the output of that algorithm, i.e. the change set \(IU(P, \Delta L_i)\) that contains the methods and classes in the client code and library that are affected by the API’s changes at version \(i\) via method overriding and inheritance relations. Then, SAM removes API i-usage changes that have nothing to do with the set \(IU(P, \Delta L_i)\).

Next, for each of such usage changes, SAM extracts from the corresponding surrounding code the pairs of usage models \((U, U')\) before and after the change at version \(i\) (line 2). To do this, for each changed method \(m \in \Delta P_i\) containing such usage changes, SAM builds the corresponding usage models \((U, U')\), and determines whether \(U\) and/or \(U'\) have any action nodes that represent any method(s) in the change set \(IU(P, \Delta L_i)\). If such a pair exists, their changes would be related to the API changes. SAM uses GroumDiff to find the changes between \(U\) and \(U'\) in term of a set of graph editing operations. That set of operations is added into the set \(E\) of API usage changes caused by the API’s changes (line 3).

Then, SAM mines the maximal frequent subset of editing operations for all the sets in \(E\), using the frequent itemset mining algorithm in Agrawal and Srikant (1994). This algorithm finds every set \(f\) that
occurs in the sets in $E$ with a relative frequency (i.e. confidence) at least $\sigma$, with $\sigma$ is a chosen threshold, and with its size as large as possible (line 4). For each of such $f$, SAM finds all pairs $(U, U')$s whose change operations include $f$ (line 6). For each pair $(U, U')$, it extracts the usage skeletons $U_o(f)$ and $U'_o(f)$ (line 7). This pair of usage skeletons are called reference models, which provide the context of the change $f$ (will be explained next). Then, it adds that pair into a set $Ref(f)$ for each mined frequent subset $f$ (line 8), which is now considered as an adaptation pattern of API usages (i.e. frequent changes on API usage models).

**Relative frequency.** The relative frequency of a set of change operation $f$ is calculated as follows. Assume that $f$ is a subset of edit operations from $U$ to $U'$. $Freq(f)$ denotes the frequency of $f$, i.e. the number of sets of change operations in $E$ contain $f$. $NUsage(f)$ is the number of API usages of the nodes affected by $f$, i.e. the number of all iGROUMs containing $U(f)$. Then, the relative frequency of $f$ is defined as $Freq(f)/NUsage(f)$.

**Reference model.** $U_o(f)$ is defined as the set of mapped nodes in $U$ that are affected by $f$ and their dependent nodes via control and data dependencies. $U'_o(f)$ is similarly defined. $U_o(f)$ and $U'_o(f)$ provide the contextual information on the change $f$. Thus, they are called the reference models of $f$. Because if another usage $V$ contains $U_o(f)$, one could consider that $V$ also has a context that could be adapted by the frequent adaptation $f$. Thus, $U_o(f)$ and $U'_o(f)$ are used to model the usage skeletons corresponding to the adaptation pattern $f$.

Figure 7.10 shows an adaptation pattern and its reference models found in JasperReport with respect to jFreechart library migration. The pattern includes the addition of two method calls `StandardChartTheme.createLegacyTheme` and `ChartFactory.SetChartTheme`, which lead to the addition of two new action nodes and one data node, along with the associated edges, and the addition of an edge from the data node `ChartFactory`. Since `setChartTheme` and `createAreaChart` use the same data node `ChartFactory`, SAM derives the reference models of this change as $U_0$ and $U'_0$ as in Figure 7.10. $U'_0$ contains not only the added sub-graph but also the nodes having dependencies with the changed nodes.

The use of reference model is also useful in the cases of newly added API elements. Suppose that $m$ is a newly added method in the new version of a library and a call to $m$ is added in $U'$. In this case, no node in $U$ can be mapped to the data node $m$. However, there might have some other currently existing nodes that are dependent to $m$ and they could be mapped back to $U$. Thus, SAM could use those nodes as referenced nodes for mapping between $U$ and $U'$ in the case of newly added nodes. In such cases, SAM will also add those dependent nodes into the reference model for later mapping.
Figure 7.10: API Usage Change Patterns and Reference Models

Improvements. To improve the accuracy of the mined patterns of usage changes, $\Delta P_i$ could contain the code taken from different sources: client code on different systems, or test code and demo code provided inside the API’s source code. The threshold $\sigma$ will be slightly different. For example, on test code and demo code, a usage pattern might be tested or demonstrated for only once. Therefore, we could choose small $\sigma$. Test code might contain the initialization of test data and the assertion of test results, which might not be parts of API usage specifications. To improve the quality of mined protocols, SAM discards such initializations and assertions when building the iGROUMs on the test code.

7.5 Recommending Adaptations

Sections 7.5.1 and 7.5.2 discuss how LibSync suggests the code locations to be adapted and edit operations required for those API usage adaptations.

7.5.1 API i-Usage Adaptation Recommendation

After detecting API changes via OAT and mining usage adaptation patterns on relevant codebases via SAM, LibSync has a knowledge base of API usage skeletons and corresponding adaptation patterns for an API $L$ of interest. For each version $i$ of the library $L$, the knowledge base contains the set of usage adaptation patterns $F$ at that version. Each pattern $f$ has a set of reference usage models
Ref(f) = (U_0, U'_0). It also contains ΔL_i, the set of changed entities of L from any two consecutive versions. With this knowledge, LibSync provides API usage adaptation recommendations on any given client code Q that needs to be adapted to a version i of L.

### 7.5.1.1 Location Recommendation

First, LibSync determines the code locations in the client system Q that potentially need adaptation to L_i. Using ΔL_i and xGROUM model of Q at that version, LibSync computes two change sets of methods XU(Q, ΔL_i) and IU(Q, ΔL_i) that are potentially affected by the changed entities in ΔL_i. Details of the method to derive those two change sets will be explained in Section 7.5.2.2. IU(Q, ΔL_i) is the set of methods and classes in L and Q that are affected by changed entities in ΔL_i (including overridden and inherited methods). XU(Q, ΔL_i) is the set of methods and classes in Q that are affected by the changed entities in ΔL_i via method overriding and inheritance. Every code location that uses an entity in IU(Q, ΔL_i) will be considered for adaptation to the changes ΔL_i of L. We use AU(Q, ΔL_i) to denote the set of the corresponding iGROUMs of such code locations.

To improve the performance, LibSync uses some pre-processing techniques. Based on text-based filtering, it finds the source files that could involve the usages of L. Each source file is tokenized. If a file does not contain any token similar to the names of classes/methods in IU(Q, ΔL_i), it will be discarded from further processing. In the next step, the remaining source files are parsed and extracted to build API i-usage models. For each model V, LibSync checks whether V contains some nodes representing the usages of any entity in IU(Q, ΔL_i). If that is the case, it will report V as a location for consideration of adaptation, i.e. V will be added to AU(Q, ΔL_i).

Let us revisit the example in Figures 7.1 and 7.5: L = jFreeChart, i = 0.9.17, Q = jBoss 3.2.7. Assume that jBoss is currently using jFreeChart 0.9.15. Using OAT, LibSync could detect IU(Q, ΔL_i) = \{A, B, x, x’, y, y’\} with the following information:

<table>
<thead>
<tr>
<th>Id</th>
<th>Label</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>XYSeries</td>
<td>modified class</td>
</tr>
<tr>
<td>B</td>
<td>DefaultTableXYDataset</td>
<td>modified class</td>
</tr>
<tr>
<td>x</td>
<td>XYSeries.&lt;init&gt;(String, boolean)</td>
<td>deprecated</td>
</tr>
<tr>
<td>x’</td>
<td>XYSeries.&lt;init&gt;(String, boolean, boolean)</td>
<td>added</td>
</tr>
<tr>
<td>y</td>
<td>DefaultTableXYDataset.&lt;init&gt;(XYSeries)</td>
<td>deprecated</td>
</tr>
<tr>
<td>y’</td>
<td>DefaultTableXYDataset.&lt;init&gt;(boolean)</td>
<td>added</td>
</tr>
</tbody>
</table>

Using text-based filtering, LibSync detects that the source file ManageSnapshotServlet.java in Q = jBoss 3.2.7 has some tokens XYSeries and DefaultTableXYDataset. Extracting iGROUMs from this file
1 function Adapt(V, F) //adapt API usage based on change patterns
2    for each U_o ∈ Ref(F): //for each change pattern f: calculate similarity
3       Relevant(V, U_o) = sim(GroumDiff(V, U_o)) //to reference models
4          (f*, U_o*) = Max(Relevant) //find the most suitable
5     Ed = GroumDiff(U_o*, U'_o) //derive referenced change operations
6    Recommend(Ed, V) //and recommend

Figure 7.11: Usage Adaptation Recommending Algorithm

for further analyzing, it finds that the iGROUM V of method doit has the nodes whose labels appear
in IU(Q, ∆L_i) (Figure 7.5). Thus, it reports V as a code location that may need the adaptation, and
adds V to AU(Q, ∆L_i) for further operation recommendation.

7.5.1.2 Operation Recommendation

LibSync uses the API i-usage change patterns in its knowledge base to derive the recommended op-
erations for each iGROUM V in the set AU(Q, ∆L_i) of usage models that are considered for adaptation.
Figure 7.11 illustrates the algorithm for this task. First, LibSync determines the change pattern f* that
is most suitable for V (lines 2-3). For each pair of reference models (U_o, U'_o) in the set of all reference
models in the knowledge base Ref(F), LibSync maps U_o and V using the GroumDiff algorithm (Fig-
ure 7.7), and computes the relevant degree between V and U_o based on the number of matched nodes
over the size of U_o (line 4). Next, it ranks them to find the reference model U_o* that is best matched to
V (i.e. with highest relevance) (line 4). At last, LibSync finds the changes of the best matched reference
model U* (line 5) and recommends such changes as edit operations on iGROUM V (line 6).

Notes. Since a usage model could use many usage protocols, LibSync may find more than one
usage change patterns f that could be mapped against V. Thus, it ranks them based on their similarity
with V and their frequencies (the higher the frequency is, the more correct the recommendation would
be). If no change pattern is suitable (e.g. the similarity is too little), V will be considered as an API
usage irrelevant to API changes in ∆L_i.

After processing all usage models, for each model V in recommended list AU(Q, ∆L_i), LibSync
reports its location, its ranked usage adaptation patterns f_s (with similarity levels and frequencies).
It also provides with each pattern a code skeleton that was collected during the usage pattern mining
process. If users choose a code location and a usage change pattern for adaptation, LibSync provides
the recommendation for adaptation at that location.

Let us revisit the example in Figures 7.8 and 7.10 for L = jFreeChart, i = 1.0.12, Q = jasperReport
3.5.0. First, LibSync detects the changed set ∆L_i of jFreeChart at that version as:
<table>
<thead>
<tr>
<th>Id</th>
<th>Label</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>StandardChartTheme</td>
<td>added class</td>
</tr>
<tr>
<td>B</td>
<td>ChartFactory</td>
<td>added class</td>
</tr>
<tr>
<td>a</td>
<td>StandardChartTheme#createLegacyTheme</td>
<td>added</td>
</tr>
<tr>
<td>b</td>
<td>ChartFactory#createLegacyTheme</td>
<td>added</td>
</tr>
</tbody>
</table>

Mining on the code base \( P = \text{JasperReport} \), \textit{LibSync} recovers the change pattern \( f = [\text{Add } a, \text{Add } b] \) with 53 pairs of reference models (one pair is the iGROUMs \((U, U')\) for code fragments in Figure 7.8). In \( Q \), \textit{LibSync} determines that iGROUM \( V \) uses a method of class \texttt{ChartFactory}. Since \texttt{ChartFactory} is in \( \Delta_L \), it is put into \( IU(P, \Delta_L) \), and thus, \( V \) is put into \( AU(P, \Delta_L) \), meaning that it should be considered for adaptation.

Matching \( V \) with the change patterns and reference models, \textit{LibSync} finds \( U \) as the best match for \( V \) with the change pattern \( f \). In the matching, it also finds the maps between the action nodes for two method calls \( c \) and \( d \) with the label \texttt{ChartFactory#createXYLineChart} in \( U \) and \( V \). Differenting \( U \) and \( U' \) gives the operations \( Ed = [\text{Add } a, \text{Add } b] \). Thus, \textit{LibSync} recommends to add those two method calls, \( a \) and \( b \), into \( V \), along with their dependencies: \( a \) is called before \( b \) and the output of \( a \) is the input of \( b \); \( b \) is called before \( c \) due to such dependencies in \( U \). To help developers make the adaptation easier, \textit{LibSync} provides the reference code in \textit{JasperReport} (see Figure 7.8).

### 7.5.2 API x-Usage Adaptation Recommendation

#### 7.5.2.1 Location Recommendation

To find the changes of xGROUM and recommend relevant adaptation, \textit{LibSync} starts with the change set \( \Delta_L \) of API and the change set \( \Delta_P \) of classes and methods in the client code. Those two change sets are obtained from the execution of \textit{OAT} on two versions of both API and client sides.

The outputs of this location recommendation algorithm are two change sets \( XU(P, \Delta_L) \) and \( IU(P, \Delta_L) \) of classes and methods that would be affected by the changes in \( \Delta_L \) in the API, taking into account x-usages and i-usages respectively. Therefore, they are also classes and methods that could need the adaptation.

This algorithm is carried out as follows:

- If \( A.m \in \Delta_L \), any method \( C.m \) overriding \( A.m \) is considered to be adapted. Thus, as \( A.m \) changes, \( C.m \) is added into \( XU(P, \Delta_L) \). \( C.m \) is also added into \( IU(P, \Delta_L) \) for the consideration of usage adaptation later.
• If $A.m \in \Delta L$, any method $D.m$ inheriting $A.m$ is also considered for adaptation for API usages via invocation, i.e. $D.m$ is added into $IU(P, \Delta L)$, because a method call to $D.m$ could be actually a call to $A.m$.

• If $A.m \in \Delta L$, and if $C.m \in \Delta P$ and $C.m$ overrides $A.m$, then $A.m$ and any ancestor method $A_o.m$ of $A.m$ (i.e. overridden or inherited) is also considered to be adapted (i.e. $A.m$ and $A_o.m$ are added to $IU(P, \Delta L)$), because a call to $A_o.m$ or $A.m$ might be dynamically dispatched as a call to $C.m$.

Let us take an example with $P = jBoss$, $L = Axis$. The changes are in Figures 7.3, 7.4, and 7.6. The set $\Delta L$ contains the following classes and methods:

<table>
<thead>
<tr>
<th>Id</th>
<th>Label</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>EJBProvider</td>
<td>modified class in Axis</td>
</tr>
<tr>
<td>B</td>
<td>Serializer</td>
<td>modified class in Axis</td>
</tr>
<tr>
<td>A.n</td>
<td>EJBProvider.getNewServiceObject</td>
<td>renamed</td>
</tr>
<tr>
<td>A.p</td>
<td>EJBProvider.getContext</td>
<td>added</td>
</tr>
<tr>
<td>A.q</td>
<td>EJBProvider.getEJBHome</td>
<td>changed in parameter</td>
</tr>
<tr>
<td>B.m</td>
<td>Serializer.writeSchema</td>
<td>changed in parameters, RetType</td>
</tr>
</tbody>
</table>
|    | ... | ...

Then, based on the xGROUM, two methods $D.m$ and $E.m$ (overriding $B.m$) and the method $C.n$ (overriding $A.n$) are considered to be adapted, i.e. added to $XU(P, \Delta L)$ (see the Table below for the ids). Their corresponding classes are also added to $XU(P, \Delta L)$. Thus, the set $XU(P, \Delta L)$ contains the following classes/methods:

<table>
<thead>
<tr>
<th>Id</th>
<th>Label</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>EJBProvider</td>
<td>extend modified class in jBoss</td>
</tr>
<tr>
<td>D</td>
<td>AttributeSerializer</td>
<td>extend modified class in jBoss</td>
</tr>
<tr>
<td>E</td>
<td>ObjectNameSerializer</td>
<td>extend modified class in jBoss</td>
</tr>
<tr>
<td>C.n</td>
<td>EJBProvider.getNewServiceObject</td>
<td>should be renamed</td>
</tr>
<tr>
<td>C.q</td>
<td>EJBProvider.getEJBHome</td>
<td>should be changed in paras</td>
</tr>
<tr>
<td>D.m</td>
<td>AttributeSerializer.writeSchema</td>
<td>change in paras, RetType</td>
</tr>
<tr>
<td>E.m</td>
<td>ObjectNameSerializer.writeSchema</td>
<td>change in paras, RetType</td>
</tr>
</tbody>
</table>
|    | ... | ...

They are also added to $IU(P, \Delta L)$, along with $A.p$ (newly added method) and other $i$-nodes, i.e. the placeholders such as $C.p$. The $IU(P, \Delta L)$ set contains the followings:
<table>
<thead>
<tr>
<th>Id</th>
<th>Label</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>EJBProvider</td>
<td>modified class in Axis</td>
</tr>
<tr>
<td>B</td>
<td>Serializer</td>
<td>modified class</td>
</tr>
<tr>
<td>C</td>
<td>EJBProvider</td>
<td>modified class in jBoss</td>
</tr>
<tr>
<td>D</td>
<td>AttributeSerializer</td>
<td>modified class in jBoss</td>
</tr>
<tr>
<td>E</td>
<td>ObjectNameSerializer</td>
<td>modified class in jBoss</td>
</tr>
<tr>
<td>A.n</td>
<td>EJBProvider.getNewServiceObject</td>
<td>renamed</td>
</tr>
<tr>
<td>C.n</td>
<td>EJBProvider.getNewServiceObject</td>
<td>should be renamed</td>
</tr>
<tr>
<td>A.p</td>
<td>EJBProvider.getContext</td>
<td>added</td>
</tr>
<tr>
<td>C.p</td>
<td>EJBProvider.getEJBHome</td>
<td>inherited from added method</td>
</tr>
<tr>
<td>C.q</td>
<td>EJBProvider.getEJBHome</td>
<td>should be changed in paras</td>
</tr>
<tr>
<td>D.m</td>
<td>AttributeSerializer.writeSchema</td>
<td>change in paras, RetType</td>
</tr>
<tr>
<td>E.m</td>
<td>ObjectNameSerializer.writeSchema</td>
<td>change in paras, RetType</td>
</tr>
</tbody>
</table>

The outputs \(XU(P, \Delta L)\) and \(IU(P, \Delta L)\) are used in mining algorithm (Figure 7.9), in location/operation recommendation for API i-usages (Section 7.5.1.1), and operation recommendation for x-usages (Section 7.5.2.2).

### 7.5.2.2 Operation Recommendation

After detecting \(XU(P, \Delta L)\), LibSync will recommend for adaptation of API x-usages for the methods in \(XU(P, \Delta L)\). Currently, the recommendation for x-usages is as follows:

- Pointing out the classes/methods that need API x-usage adaptation. For example, two methods `AttributeSerializer.writeSchema` and `ObjectNameSerializer.writeSchema` in Figure 7.3.
- Showing the changes to the API classes and methods in use. For example, it shows the changes to `Serializer.writeSchema` with two operations: Add a new parameter and Replace the return type.
- Suggesting the operation of classes and methods in client code that need adaptation. For example, it suggests to Add a parameter of type `Class`, and to Replace return type into `org.w3c.dom.Element`. It recommends fully qualified names to help the developers to use correct packages.

### 7.6 Evaluation

This section presents the evaluation of our framework. For OAT, the parameter setting of similarity thresholds \(\delta = 0.75\) and \(\mu = 0.625\) is used. For SAM, the parameter setting of coefficients \(\alpha = 0.5\),
Table 7.1: Subject Systems

<table>
<thead>
<tr>
<th>Client</th>
<th>Life Cycle</th>
<th>Releases</th>
<th>Methods</th>
<th>Used APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>jBoss (JB)</td>
<td>10/2003 - 05/2009</td>
<td>47</td>
<td>10-40K</td>
<td>45-262</td>
</tr>
<tr>
<td>Spring (SP)</td>
<td>12/2005 - 06/2008</td>
<td>29</td>
<td>10-18K</td>
<td>45-262</td>
</tr>
</tbody>
</table>

Table 7.2: Precision of API Usage Change Detection

<table>
<thead>
<tr>
<th>Client</th>
<th>Changes</th>
<th>Libs</th>
<th>Operations</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>JasperReports</td>
<td>30</td>
<td>5</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>JBoss</td>
<td>40</td>
<td>17</td>
<td>38</td>
<td>2</td>
</tr>
<tr>
<td>Spring</td>
<td>30</td>
<td>15</td>
<td>30</td>
<td>0</td>
</tr>
</tbody>
</table>

\[\beta = 0.5\] and matching threshold \[\lambda = 0.5\] is used in \textit{GroumDiff} algorithm, and confidence threshold \[\sigma = 0.5\] is used in \textit{ChangePattern}.

7.6.1 Adaptation of i-Usage

We evaluated the quality of \textit{LibSync} in recommending API i-usage adaptations. In order to recommend API i-usage adaptation, \textit{LibSync} needs to detect API i-usage changes and derive adaptation patterns.

The experiments were carried out on large-scale, real-world systems in different application domains with long histories of development. Table 7.1 shows the details about those subject systems. For example, jBoss is a middle-ware framework that has been developed for more than 6 years with more than 40 releases. It has about 40 thousand methods and uses hundreds of different libraries.

7.6.1.1 Detection of i-Usage Changes

In this experiment, our evaluation questions were (1) can \textit{CUE} detect the API usage changes correctly? and (2) are the client-side, API usage changes detected by \textit{CUE} and \textit{SAM} indeed caused by the evolution of libraries used by the client system?

We ran our tool on those three subject client systems to report all API usage changes along with edit operations. For each client, we randomly picked 30 to 40 of the API usage changes. We manually checked the correctness of detected edit operations in API usage skeletons. In addition, we also examined whether the identified API usage changes are indeed caused by the changes to APIs.

Table 7.2 shows the result of this investigation. Column \textit{Changes} shows the number of checked cases in detected API usage changes. Column \textit{Libs} shows the number of libraries involved in those reported usage changes. The next two columns (\textit{Operations}) display the numbers of correctly (see column √)
Figure 7.12: Create NumberAxis in jFreeChart

and incorrectly detected API i-usage changes (column X) respectively. Similarly, the last two columns (column API) show how correctly our tool relates an API usage change to the changes to API(s).

In most cases, our tool correctly detected the edit operations and correctly related the API usage changes on the client-side and the library-side changes (see two columns √). In 93 cases out of 100 checked cases, our tool correctly detected API usage changes and related them to library-side API declaration changes.

**Example 1.** Let us discuss an interesting case in Figure 7.12. This usage of jFreeChart creates a NumberAxis object and sets up its range and ticking unit. In the versions before 0.9.12 of jFreeChart, setting up the range of a NumberAxis object is carried out by invoking two methods setMinimumAxisValue and setMaximumAxisValue. However, from version 0.9.12, those two methods are deprecated, a new method setRange is added and should be used instead. SAM correctly identified API usage skeletons but did make some mistakes in deriving edit operations for adaptation. Instead of reporting two deletions and one addition, it reported one replacement and one addition. Importantly, however, SAM is able to recognize and correlate that the API usage change is due to the change in jFreeChart API specification.

In some other cases, our tool wrongly related client-side updates with library-side updates even though the library-side updates did not affect the corresponding usage in the client code such as a method’s access visibility modification. Another case is when the API method changes the types of exceptions that could be thrown, but the client code always catches the general exception type, Exception. Another one is when the API method changes the type of one parameter into its super-type (e.g. from String to Comparable). In those cases, there were some changes to those API usages but these changes were irrelevant to changes in the declaration of the API. Our tool mistakenly related them. Let us explain another interesting case of API usage changes due to the evolution of a library.

**Example 2.** Ruby, a scripting language/framework for Web applications, provides a new method parse in the version 0.8.0. This method accepts two string inputs: one referring to the piece of code required to compile and one referring to the compiling configuration. It returns a Node as the root node of the parse-tree. Using this newly added feature of Ruby, developers of Spring changed their
IRubyObject rubyObject = ruby.evalScript(scriptSource); // direct evaluation
if (rubyObject instanceof RubyNil) {
    throw new ScriptCompilationException(...);
}

Node scriptRootNode = ruby.parse(scriptSource, ""); // parse the script
IRubyObject rubyObject = ruby.eval(scriptRootNode); // eval the parse−tree
if (rubyObject instanceof RubyNil) { // if cannot eval the whole script
    String className = findClassName(scriptRootNode); // just find class name
    rubyObject = ruby.evalScript("n"+className+".new"); // to create an object
}
if (rubyObject instanceof RubyNil) {
    throw new ScriptCompilationException(...);
}

Figure 7.13: API usage changes in Spring with respect to the evolution of Ruby

implementation of the method createJRubyObject, which receives a string scriptSource as the input script, and returns an Object created by that script. In the old version of this method, it calls the evalScript method directly on scriptSource. This direct evaluation could have a disadvantage in which the script is not well-formed, or more severely, is crafted as malicious code that exploits some vulnerabilities of the system. In the new version, Spring code first calls parse to parse the scriptSource into a tree, and then calls the eval method to execute this parsed code. If the script is ill-formed or maliciously crafted, the parsing will not return a well-formed parse tree and the eval method simply does not execute, thus, resolving the above vulnerability issue. LibSync was able to mine this API usage adaptation pattern based on the API usage changes in the client code of Spring at 2.0 (JRubyScriptUtils.java).

7.6.1.2 Recommendation of Locations for Adaptation

This section describes the evaluation of LibSync in recommending the code locations for adaptation to a target library version. We chose six pairs of a library and its client. For each pair, let $V_C$ and $V_A$ be the versions of the client system and the library respectively. For each $V_A$, we selected another version $V_A'$ of the library such that the client system had been changed in a later version than $V_C$. We ran LibSync on $V_A$ and $V_A'$ to detect library-side changes and client-side API usage updates. LibSync
Table 7.4: Accuracy of i-Usage Operations Recommendation

<table>
<thead>
<tr>
<th>Mine on</th>
<th>Adapt to</th>
<th>Usages</th>
<th>Rec.</th>
<th>√</th>
<th>Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.5-3.2.8</td>
<td>3.2.5-4.0.5</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4.0.5-4.2.3</td>
<td>4.0.5-5.0.1</td>
<td>26</td>
<td>25</td>
<td>25</td>
<td>1</td>
</tr>
</tbody>
</table>

was run to recommend the locations for adaptation. Then, we manually checked in the history of the client code after that version \( V_C \) to see whether the code at those locations have been actually updated to work with the new library version \( V_A' \).

Table 7.3 shows the result. \texttt{jFree} and \texttt{Jasper} are used as abbreviations for \texttt{jFreeChart} and \texttt{Jasper-Reports}, respectively. Column \texttt{Version} shows the pairs of versions of the library and the client system. Column \texttt{Rec.} shows the number of the recommended locations. Columns \texttt{√}, \texttt{X}, \texttt{Miss} show the correctly, incorrectly, and missed detected locations respectively. Column \texttt{Hint} represents the cases in \texttt{JasperReport} on the changes of \texttt{jFreeChart} in which the API methods are \textit{deprecated}, but developers have not updated yet in the code. As we could see, \texttt{LibSync} provides highly correct locations. It missed in only one case in the total of 67 recommendation locations.

### 7.6.1.3 Recommending Edit Operations for Adaptation

In this experiment, we ran \texttt{LibSync} on a development branch in jBoss’ history to mine usage adaptation patterns for all libraries used by jBoss. We then ran \texttt{LibSync} for adaptation recommendation on another branch which derives from the same branching point with the first branch but are independently developed onward. We manually checked the recommended operations against the actual adaptations in the second branch. A recommendation is considered correct if it has at least one correct operation at a correct location.

Table 7.4 shows the result. The first two columns show the development branches on which \texttt{LibSync} mined the adaptation patterns and applied adaptation recommendations respectively. Column \texttt{Usages} shows the number of usage adaptations. Column \texttt{Rec} shows the numbers of recommended adaptations. As we could see, \texttt{LibSync} provides highly correct recommendations. The recommended operations were correct as developers changed all of them except for three missing cases in which old usages were completely abandoned and totally new usages were used.

\texttt{LibSync} was able to correctly recommend the adaptation for all examples in this chapter. For example, \texttt{LibSync} could recommend the correct adaptation for the case of \texttt{jFreeChart} in jBoss in Figure 7.1. This change happened in jBoss 3.2.8 in the branch from version 3.2.5 to 3.2.8 and was learned to adapt from version 4.0.1 to 4.0.2 in the branch from version 3.2.5 to 4.0.5. Those two changes
Table 7.5: Accuracy of x-Usage Recommendation

<table>
<thead>
<tr>
<th></th>
<th>Rec.</th>
<th>√</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Class name</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Package name</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Deprecated</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Change parameter type</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Del parameter</td>
<td>7</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>Change return type</td>
<td>6</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Change exception</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Add parameter-Change Exception</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Add parameter-Change Return type</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

were actually the patches to fix a bug of NullPointerException when using the deprecated constructor of DefaultTableXYDataset.

7.6.2 Adaptation of x-Usage

This section describes our evaluation of LibSync in recommending the code locations for the adaptation of x-usages in jBoss. We used a wide range of versions in jBoss as described in Table 7.1. For each change in jBoss from version $i$ to $j$, we used OAT to collect all changed APIs into the change set $ΔL_i$. We identified a set $XU(P, ΔL_i)$, all methods in jBoss at version $i$ that override some API’s methods.

Each method in $XU(P, ΔL_i)$ is considered for adaptation recommendation with the same operations as those operations that are detected on the overridden method in the API. A recommendation to a method at version $i$ was considered correct if that method was really changed in the same way in the version $j$, otherwise, it was marked incorrect.

The result is shown in Table 7.5. Each row represents one particular type of changes in the external API(s). For example, the row Name is only for the methods with changed names. The row of Add parameter-Change Exception is for the methods changing in both parameter and exception that could be thrown. Therefore, the numbers in a column are exclusive from row to row. Column Rec shows the number of recommended locations. Columns √ and X respectively show the correctly and incorrectly detected locations for x-usage adaptation.

LibSync provides highly correct recommendations. It is incorrect in only two cases out of the total of 33 cases. These two wrong cases have the same nature in which they are both caused by the incorrect mapping results from OAT when detecting the changes of the class PersistenceInfoImpl in javax API that is used in jBoss from version 4.0.3SP1 to 4.0.4GA. Instead of reporting two deleted methods (getPersistenceXmlFileUrl and setPersistenceXmlFileUrl), and two added methods (getPersistenceUnitRootUrl and setPersistenceUnitRootUrl), OAT reported two renaming operations.
Therefore, the recommendation was two renaming operations while the correct adaptation should be two deletions and two additions. For other types of changes, the recommendations are all correct.

7.7 Related Work

7.7.1 Library Evolution and Client Adaptation

There are several existing approaches to support client adaptations to cope with evolving libraries. Chow and Notkin (1996) proposed a method for changing client applications in response to library changes—a library maintainer annotates changed functions with rules that are used to generate tools that will update client applications. Henkel and Diwan’s CatchUp (2005) records and stores refactorings in an XML file that can be replayed to update client code. However, its update support is limited to three refactorings: renaming operations (e.g. types, methods, fields), moving operations (e.g. classes to different packages, static members), or change operations (e.g. types, signatures). The key idea of CatchUp, record-and-replay, assumes that the adaptation changes in client code are exact or similar to the changes in the library side. Thus, it works well for replaying rename or move refactorings or supporting API usage adaptations via inheritance. However, CatchUp cannot suggest programmers how to manipulate the context of API usages in client code such as the surrounding control structure or the ordering between method-calls such as the example shown in Section 7.2. Furthermore, CatchUp requires that library and client application developers use the same development environment to record API-level refactorings, limiting its adoption in practice.

SemDiff (2008) mines API usage changes from other client code or the evolution of library itself, similar to our work. The key difference of LibSync from SemDiff is that with our work uses a graph-based representation to capture the context of an API usage, including the dependencies among method calls and with a surrounding control logic. In our work, an adaptation pattern is captured in term of a frequent set of graph editing operations that are common to multiple API usage skeletons before and after library migration. On the other hand, SemDiff defines an adaptation pattern as a frequent replacement of a method invocation. That is, if a method call to $A.m$ is changed to $B.n$ in several adaptations, $B.n$ is likely to be a correct replacement for the calls to $A.m$. As SemDiff models API usages in terms of method calls, it cannot support complex adaptations that involve multiple objects and method calls and that require the knowledge of the surrounding context of those calls. LibSync’s key departure point is that when a library’s API declarations are modified, such evolution often involves coordinating uses of multiple objects and multiple method calls under certain contexts.
Xing and Stroulia’s Diff-CatchUp \cite{Xing:2007} automatically recognizes API changes of the reused framework and suggests plausible replacements to the obsolete APIs based on working examples of the framework codebase. Dig et al.’s MolhadoRef \cite{Dig2007} uses recorded API-level refactorings to resolve merge conflicts that stem from refactorings; this technique can be used for adapting client applications in case of simple rename and move refactorings occurred in a library.

Tansey and Tilevich’s approach \cite{Tansey:2008} infers generalized transformation rules from given examples so that application developers use the inferred rules to refactor legacy applications. However, this approach focuses on annotation refactorings that replace the type and naming requirements to the annotation requirements of a target framework. Furthermore, this approach does not focus on updating client applications to cope with evolving libraries.

Andersen and Lawall \cite{Andersen:2008} proposed \textit{spdiff} that identifies common changes made in a set of files. API developers could use \textit{spdiff} to extract a generic patch and apply it to other clients. Their approach models the changes at the level of \textit{text-line} changes. On the other hand, LibSync uses a graph-based representation to capture more thorough syntactic and semantic information for adapting API usages. SmPL \cite{Padioleau:2007, Lawall:2009} is a domain-specific source transformation language that captures textual patches with a more semantic description of program changes. However, it does not explicitly distinguish API changes from their usage changes.

### 7.7.2 Program Differencing and Origin Analysis

Existing differencing techniques use similarities in names and structures to match code elements at a particular granularity: lines and tokens \cite{Tichy:1984}, abstract syntax tree nodes \cite{Fluri:2007, Neamtiu:2005}, control flow graph nodes \cite{Apiwattanapong:2004}, and program dependence graph nodes \cite{Binkley:1995}, etc. Our API usage comparison algorithm is similar to program differencing algorithms that it detects changes between two versions of an internal program representation using name-, content- and structure-based similarities. Zou and Godfrey \cite{Zou:2005} first developed an origin analysis technique to support software evolution analyses by mapping corresponding code elements between two program versions. Several other techniques \cite{Dig:2006, Kim:2005b, Weissgerber:2006, Xing:2005, Zou:2005} improved and extended prior origin analysis techniques; some of these derive refactoring transformations—move a method, rename a class, add an input parameter, etc.—based on the matching result between two versions. We developed our own analysis technique, OAT, to map corresponding code API declarations and API usage code fragments by improving these existing analyses.
### 7.7.3 API Usage Specification Extraction

There exist several approaches for extracting API usage specifications. The forms of recovered API usage specifications and patterns include finite state automaton Wasylkowski et al. (2007); Zhong et al. (2009b), pairs of method calls Livshits and Zimmermann (2005); Williams and Hollingsworth (2005), partial orders of calls Acharya et al. (2007); Thummalapenta and Xie (2009), Computation Tree Logic formulas Wasylkowski and Zeller (2009). The API usage representations in those static approaches are still limited, for example, the patterns are without control structures and involve only individual objects belonging to one class. Our graph-based API usage representation captures multi-object API usage patterns with control structures. In contrast to those static approaches, dynamic approaches recover the specifications by investigating the execution traces of programs Gabel and Su (2008); Yang et al. (2006); Shoham et al. (2007); Ramanathan et al. (2007a); Pradel and Gross (2009). These dynamic approaches require a huge amount of execution traces. Our graph-based representation, iGROUM, captures API usage patterns with control and data dependencies among method calls, and surrounding control logic such as while loop and if statement. The API usage representations in this chapter extend our prior work on GRouMiner Nguyen et al. (2009c) to tailor the original multi-object usage representation in order to capture the relevant context surrounding the use of external APIs. In particular, iGROUM explicitly captures API types and methods that appear in action and data nodes, so that program slicing can isolate only a sub-graph that is relevant to the use of a particular library. On the other hand, xGROUM, captures overriding and inheritance relationships between client methods and API methods.

### 7.7.4 Empirical Studies of API Evolution

Dig and Johnson Dig and Johnson (2005) manually investigated API changes using the change logs and release notes to study the types of library-side updates that break compatibility with existing client code, and discovered that 80% of such changes are refactorings. Xing and Stroulia Xing and Stroulia (2006) used UMLDiff to study API evolution in several systems, and found that about 70% of structural changes are refactorings. Kim et al.’s signature change pattern analysis Kim et al. (2006b) categorizes API signature changes in terms of data-flow invariant. Yokomori et al. Yokomori et al. (2009) investigated the impact of library evolution on client code applications using component ranking measurements. Padioleau et al. Padioleau et al. (2006) found that API changes in the Linux kernel lead to subsequent changes on dependent drivers, and such collateral evolution could introduce bugs into previously mature code. These studies motivate the need for supporting complex client adaptations beyond replaying library-side refactorings in client code.
7.8 Conclusions

This chapter presents *LibSync* that guides developers in adapting API usages in client code to cope with evolving libraries. *LibSync* uses several graph-based techniques to recover the changes of API usage skeletons from codebase of other client systems, and recommends the locations and edit operations for adapting API usage code. The evaluation of *LibSync* on real-world software systems shows that it is highly correct and useful. Especially, *LibSync* can recover and recommend on complex API usage adaptations, which current state-of-the-art approaches are hardly able to support. One limitation of our approach is that it requires a set of programs that already migrated to a new library version under focus or adequate amount of API usages within the library itself; as it is not straightforward to identify which version of a library is used by client systems, we are currently in the process of developing a co-evolution analysis framework that can automatically extract the versioning information of libraries used by client systems in order to build a large corpus of API usage skeletons and to build a repository of API usage adaptation patterns.
CHAPTER 8. CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

Software systems inevitably contain a large amount of repeated artifacts at different levels of abstraction. This dissertation focuses on analyzing software repetitiveness at the implementation code level and leveraging the derived knowledge for easing tasks in software maintenance and evolution such as program comprehension, API use, change understanding, API adaptation, and bug fixing.

We have developed different representations for software artifacts at source code level, and the corresponding algorithms for measuring code similarity and mining techniques. Our mining techniques base on the key insight that code that conforms to programming patterns and specifications appears more frequently than code that does not. Thus, correct patterns and specifications can be mined from large code corpora. We also have built program differencing techniques for analyzing changes in software evolution. Our key insight is that similar code is likely changed in similar ways and similar code likely has similar bug(s) which can be fixed similarly. Therefore, learning changes and fixes from the past can help automatically detect and suggest changes and fixes to the repeated code in development.

Our empirical evaluation shows that our techniques can accurately and efficiently detect repeated code, mine useful programming patterns and API specifications, and recommend changes. It can also detect bugs and suggest fixes, and provide actionable insights to ease maintenance tasks. Specifically, our code clone detection tool detects more meaningful clones than other tools. Our mining tools recover high quality programming patterns and API preconditions. The mined results have been used to successfully detect many bugs violating patterns and specifications in mature open-source systems. The mined API preconditions are shown to help API specification writer identify missing preconditions in already-specified APIs and start building preconditions for the not-yet-specified ones. The tools are scalable which analyze large systems in reasonable times. Our study on repeated changes gives useful insights for program auto-repair tools. Our automated change suggestion approach achieves top-1 accuracy of 45%–51% which relatively improves more than 200% over the base approach. For a special type of changes, API adaptation, our tool is highly correct and useful with precision of 100% and recall of 91%.
8.2 Future Work

8.2.1 Mining Postconditions of Behavioral Interface Specifications

Behavioral interface specification contains not only preconditions but also postconditions. We will develop an approach with similar ideas as the approach for mining preconditions in which, after calling a method, programmers commonly check its post-states which are related to the postconditions in the formal specification of the API. For example, after calling method `compare(Integer, Integer)` to compare two `Integer` objects, developers usually check the return value against 0 to determine what to do next. We will mine the conditions that are frequently checked after calling an API. Let us call them post-state conditions. After calling method `compare(Integer, Integer)`, the post-state conditions would be \( \texttt{result} = 0, \texttt{result} > 0 \) and \( \texttt{result} < 0 \), where \( \texttt{result} \) denotes the return value.

The post-state conditions might not be sufficient to derive the postconditions in the formal specification because they do not contain information about the relations between the states before (e.g. inputs) and after (e.g. return value) calling the API. For example, method `compare(Integer, Integer)` returns a positive value if the integer value of the first argument is greater than that of the second argument. We will apply a dynamic symbolic execution (DSE) technique Tillmann and Schulte (2005); Tillmann and Halleux (2008); Xie et al. (2009); Sen et al. (2005); Majumdar and Sen (2007) on the source code of the API to get this information. DSE is a testing technique that combines symbolic execution, path constraints on input values and concrete execution to systematically explore and cover as many as possible paths in the program under testing. After running DSE, we have a set of pairs of a path constraint and a heap. For example, if we run DSE on method `compare(Integer, Integer)` above, we would have a set of three pairs \( \texttt{arg0.value == arg1.value, result = 0}, \) \( \texttt{arg0.value > arg1.value, result = 1} \) and \( \texttt{arg0.value < arg1.value, result = -1} \). They are used together with the set of post-state conditions to infer the postconditions. For example, since \( \texttt{result} = 1 \) (on the heap) satisfies the post-state condition \( \texttt{result} > 0 \), we can infer a postcondition that \( \texttt{result} > 0 \) if \( \texttt{arg0.value > arg1.value} \).

8.2.2 Inferring API Specifications from Similar APIs

The proposed specification mining techniques work well for the APIs which are widely-used. However, since they are driven by data, they cannot be applied to the newly-released APIs which are rarely-used. Our future work will aim at developing similarity-based techniques to infer specifications for rarely-used APIs. Our key insight is that similar code would likely have similar or the same specification.

First, we will build a database of pairs of API and its corresponding specification from as many as possible already-specified APIs. Then, we will enrich this database with the specifications mined for
the widely-used APIs. The APIs’ code in the database will be abstracted by a representation that can capture the semantics and behaviors of the APIs which will be used in comparing APIs’ implementations. We plan to use PDG-like representation. We will use Exas Nguyen et al. (2009a) (Section 2.2) to measure the similarity which reduces the complexity of the graph comparison problem and enable hashing which makes our approach scalable.

For a given new API, we will search its abstracted implementation in our database to find the most similar implementations and derive the most likely specifications. To accommodate the variation of the implementations, we will systematically study program transformations that preserve the specifications such as desugaring, normalization, if and loop optimization and refactoring. The inferred specifications will be used as the starting point for further refinement and verification.

8.2.3 Validating Mined Specifications

When a specification is given to a developer or a checker tool, it is essential to know if the given specification is necessary for the API or not. We will develop automated techniques using different strategies for this validation process to reduce manual effort.

8.2.3.1 Finding counter-examples of specifications from the large codebase to refute the false positives

Given an API and its mined specification, e.g., \texttt{List.add(Object)} with the precondition \texttt{argument \textasciitilde null}, we will automatically extract executable client code from large-scale corpus that uses the API. Then, for each API call location, we will try to execute the calling code to exercise the API call so that the program state either satisfies or violates the API’s specification. If there exists an input to the program leading to a program state that violates the API’s specification, e.g., a \texttt{null} object is passed to method \texttt{List.add()} in this example, and still produces the correct result, a counter-example is found, e.g., \texttt{argument == null} in this example. If there exists an input to the program leading to a program state that satisfies the API’s specification but produces an unexpected result, a counter-example is found. The counter-example could be used to remove the false positives or, in certain cases, correct (weaken or strengthen) the mined ones. For example, in this example with \texttt{List.add()}, the counter-example could be used to refine the specification to infer that no precondition is needed.

We will also adapt prior work on executing arbitrary blocks of code Jiang and Su (2009), and steering execution path towards program points of interest Flanagan and Godefroid (2005); Godefroid (2005); Sen et al. (2005); Majumdar and Sen (2007). In the case we have access to the source code of the API,
we will adapt prior work that uses static program checkers Flanagan and Leino (2001); Flanagan et al. (2002) to verify or refute the mined specifications.

8.2.3.2 Leveraging maturity/stability of the high fidelity code corpus to filter the false positives

It is conventional wisdom that the more mature/stable the code is, the more likely it contains correct usages of the API. Thus, the specifications appearing in many mature/stable code usages could be likely correct and the ones not appearing in mature/stable client code could be likely false positives. We will first develop a measure for the maturity/stability of client methods/projects in the code corpus based on their change history. Our assumption is that client code gets mature/stable as it evolves. Thus, we will adapt the techniques used in our prior study on software changes Nguyen et al. (2012, 2013b) to compute this measure that takes the number of changes and the amount of changed code in the history of the projects into consideration. Then, we will develop a technique to filter specifications, in which those mined from more mature projects are more favored.

8.2.3.3 Using bug-fixing change history to filter false positives

Bug fixes are changes that correct the behaviors of programs. If certain fixes related to the specifications of APIs occur frequently, they can be used to refine the mined specifications. For example, if a guard condition of an API call is removed in many bug-fixing changes and happens to be part of the mined preconditions, it could indicate that the corresponding precondition is not needed and should be filtered. We will leverage the syntactic and semantic changes extracted by the techniques in our prior work Nguyen et al. (2012, 2013b, 2010a,b) to derive changes related to API specifications.

8.2.4 Mining Bug Fixing Change Patterns

Existing studies on code change repetitiveness and patterns are limited to changes at syntax level. On the other hand, approaches for mining adaptations to library API and framework changes focus only on the method calls, thus, miss changes to other kinds of computations in programs such as arithmetic and logical operations. Kim et al. (2013) pre-defined a set of templates for bug fixing, thus, cannot deal with not-yet-seen fixes.

In future work, we will systematically study the repetitiveness of changes for all kinds of expressions in programs augmented with semantic information such as type information. More specifically, the study will focus on fixing changes rather than general changes. The mined change patterns will be
categorized and mapped to high-level program changes. They are also be used to recommend fixes to buggy programs.

We will focus on changes at expression level and divide expressions into two kinds: computation and condition. Computation expressions are identifiers, method invocations, infix/prefix/suffix expressions, etc. Condition expressions are expressions as the conditions of the branching and looping in the program that control the flow of the programs.

The syntactic fine-grained changes to those expressions will be abstracted into higher level changes. For each syntactic unit, there will be a specific set of pre-defined rules for abstraction. For example, a change to a method call could be recognized as deleting/adding/replacing or renaming method name, or adding/deleting/replacing an argument, or reordering the argument list, or any combinations of them. We will build those rules as exhaustive as possible. The changes to condition expressions will be associated with specific computation expressions. For example, adding a null check condition on the receiver object of a method call and adding a range check on the first argument of a method call are composite high level changes.

A fixing change pattern will contain a set of expression changes that occur frequently together in many bug fixes. Therefore, in general, we will employ the idea of frequent item set mining for mining those patterns. One key difference is that in the pattern of this problem, to fix a bug, one change can be applied in multiple locations, making a pattern a bag of single changes rather than a set. The mining algorithm has to take this difference into consideration.
Bibliography


Java Path Finder (JPF). http://babelfish.arc.nasa.gov/trac/jpf.


CCFinderX. [http://www.ccfinder.net](http://www.ccfinder.net)


Deng, X., Robby, and Hatcliff, J. Kiasan: A verification and test-case generation framework for java based on symbolic execution. In Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISOLA ’06.


Jmol. [Jmol](http://sourceforge.net/projects/jmol/).


