Title
A Characterization and Partial Automation of the Multi-revision, Fine-grained Analysis of Code History as an Efficient and Accurate Mechanism to Support Software Development

Permalink
https://escholarship.org/uc/item/8f41604d

Author
Servant, Francisco

Publication Date
2015-01-01

Peer reviewed|Thesis/dissertation
A Characterization and Partial Automation of the Multi-revision, Fine-grained Analysis of Code History as an Efficient and Accurate Mechanism to Support Software Development

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Francisco Javier Servant Cortés

Dissertation Committee:
Associate Professor James A. Jones, Chair
Professor Richard N. Taylor
Professor André van der Hoek
Professor Cristina Videira Lopes

2015
To the mentors in my life, and to those who pursue and promote the expansion of human knowledge.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF ALGORITHMS</td>
<td>ix</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>x</td>
</tr>
<tr>
<td>CURRICULUM VITAE</td>
<td>xi</td>
</tr>
<tr>
<td>ABSTRACT OF THE DISSERTATION</td>
<td>xiv</td>
</tr>
</tbody>
</table>

1 Introduction 1

2 The Process of Multi-revision, Fine-grained Analysis of Source-code History 6

3 History Slicing: The Process of Obtaining the History of a Set of Lines of Code 9
   3.1 History-slicing Criterion 10
   3.2 History Slicing 10
      3.2.1 Navigate the Dimension of History 11
      3.2.2 Navigate the Dimension of the Program 12
   3.3 History Slice 13

4 The Multi-revision, Fine-grained Analysis of Source-code History Can Be Partially Automated in a Way That Is Efficient 15
   4.1 Efficiency Limitations of Current Approaches for History Slicing 16
      4.1.1 Manual, Naïve History Slicing 17
      4.1.2 Conventionally Assisted History Slicing 18
   4.2 Automatic History Slicing 20
      4.2.1 Build History Graph 21
      4.2.2 History-slicing Criterion 24
      4.2.3 Analyze History Graph 25
      4.2.4 History Slice 26
   4.3 Evaluation of the Efficiency Provided by Automatic History Slicing 27
5 The Multi-revision, Fine-grained Analysis of Source-code History Can Be Partially Automated in a Way That Provides Support for Answering Developer Questions

5.1 Motivation and Background

5.1.1 Ad-hoc Developer Questions about Source Code History

5.1.2 Anticipated Developer Question: Who are the most suitable developers to fix a bug?

5.2 Chronos

5.2.1 Select Lines of Code of Interest

5.2.2 Examine History of Lines of Code of Interest

5.3 WhoseFault

5.3.1 Select Lines of Code of Interest

5.3.2 Obtain History of Lines of Code of Interest

5.3.3 Examine History of Lines of Code of Interest

5.4 Evaluation of Chronos

5.4.1 Experimental Design

5.4.2 Experimental Subjects

5.4.3 Independent Variables

5.4.4 Dependent Variables

5.4.5 Results

5.4.6 Threats to Validity

5.5 Evaluation of WhoseFault

5.5.1 Experimental Design

5.5.2 Experimental Subjects

5.5.3 Subject Preparation

5.5.4 Ground Truth

5.5.5 Independent Variable: Bug-Assignment Technique

5.5.6 Dependent Variables

5.5.7 Results

5.5.8 Threats to Validity

6 The Multi-revision, Fine-grained Analysis of Source-code History Can Be Partially Automated in a Way That Is Accurate

6.1 Accuracy Limitations of Current Approaches for History Slicing

6.1.1 Manual History Slicing

6.1.2 All-to-All History Slicing

6.1.3 One-to-One History Slicing

6.2 Automatic Fuzzy History Slicing
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The process of multi-revision, fine-grained analysis of source-code history</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>Characterization of the operation of obtaining the history of a set of lines of code</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Characterization of <em>history slicing</em> with a real-world example</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>Characterization of a <em>history slice</em> with a real-world example</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>My automatic approach to history slicing</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>History Graph</td>
<td>21</td>
</tr>
<tr>
<td>4.3</td>
<td>Mean total number of lines to which a developer would be exposed using each history-slicing approach. Automatic History Slicing (AHS) drastically reduces the expense for a developer. Note that the columns representing the Naïve approach far exceed the bounds of the chart</td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>Conceptual depiction of the bug-assignment process. This process matches the expertise needed to fix the bug with developer expertise to recommend developers for the bug-fixing task</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>My new execution-based bug-assignment approach versus existing natural-language-based approaches</td>
<td>43</td>
</tr>
<tr>
<td>5.3</td>
<td>CHRONOS allows interactive selection of the lines of code of interest</td>
<td>48</td>
</tr>
<tr>
<td>5.4</td>
<td>CHRONOS visualizing a history slice. CHRONOS allows zooming at any level over any area of the visualization</td>
<td>49</td>
</tr>
<tr>
<td>5.5</td>
<td>A global, multi-selection timeline visualization supports pattern recognition and discovery</td>
<td>49</td>
</tr>
<tr>
<td>5.6</td>
<td>The individual history of a code selection provides semantically rich detail of code changes and supporting meta-data</td>
<td>51</td>
</tr>
<tr>
<td>5.7</td>
<td>Distribution of NDCG$_5$ scores obtained by each technique for each separate subject program. Notice that WHOSEFAULT (the right-most box plot for each subject) not only provided as much median effectiveness as natural-language-based techniques, but it was actually more effective than most of the studied techniques for each subject.</td>
<td>74</td>
</tr>
<tr>
<td>5.8</td>
<td>Distribution of NDCG$_5$ scores obtained by each technique across all subject programs. Notice that, for each natural-language-based technique, WHOSEFAULT (the right-most boxplot cluster) provided at least equal effectiveness for any subject, and it provided higher effectiveness for at least one subject.</td>
<td>76</td>
</tr>
</tbody>
</table>
5.9 Time (seconds) spent by each technique to provide a recommendation for a bug on each subject. Notice that WhoseFault (the right-most box plot for each subject) took longer time to execute than most natural-language-based techniques for all subjects. Nevertheless, WhoseFault took a median and mean execution time under one minute.

6.1 Code evolution example and history slices produced by different approaches.

6.2 Manually-assessed history graph (a) and slice (b). On the slice (b), solid edges and solid nodes denote changes contributing to slicing criterion; dashed edges and hollow nodes denote corresponding but unchanged lines.

6.3 Existing all-to-all history graph and slice. The resulting history slice has high recall but low precision.

6.4 Existing one-to-one history graph and slice. The resulting history slice has high precision but low recall.

6.5 My automatic, fuzzy approach to history slicing.

6.6 Example of a fuzzy history graph and fuzzy history slice. Fuzzy history slices allow for tuning precision and recall in a way that is more accurate than existing, absolute techniques.

6.7 Mean precision, recall, and f-measure provided by the different code-evolution models for the different change-hunk types. Higher scores are better.

6.8 Accuracy provided by history slicing for the different code-evolution models. Higher NDCG_{50} scores are better.
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Mechanisms provided by current approaches for history slicing</td>
<td>16</td>
</tr>
<tr>
<td>4.2</td>
<td>Example of a history-slicing criterion</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>Amount of information to which developers would be exposed for different</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>history-slicing approaches. Each row contains the mean value for 1,000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>studied history slices. Lower values mean higher efficiency.</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Number of subjects with ( n ) years of experience with IDEs and revision-</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>control systems.</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Time to answer question and correct answer rate for each technique and</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>question.</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Empirical subjects</td>
<td>68</td>
</tr>
<tr>
<td>5.4</td>
<td>Distribution of NDCG(_5) scores obtained by all techniques for all</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>subject programs</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>Time (seconds) spent by each technique to provide a recommendation for a</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>bug on each subject</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Example of a fuzzy history-slicing criterion</td>
<td>93</td>
</tr>
</tbody>
</table>
## LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Build Fuzzy History Graph</td>
<td>94</td>
</tr>
<tr>
<td>6.2 Analyze Fuzzy History Graph</td>
<td>97</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I would like to thank my parents, Francisco Servant López and Angustias Cortés Higueruelo, for their continued support of my academic endeavors. They have always encouraged me to pursue my goals, even when it meant living half-way across the world from them for many years.

I would also like to thank my wife, Irene López Armenteros, who gave me strength to follow my interest in research all the way to obtaining the doctoral degree. Irene invested great effort by moving to California to be my partner in my mission for obtaining a Ph.D. and by aligning her goals with mine. I am eternally grateful for that.

I am extremely grateful for the guidance and friendship with which my advisor James A. Jones provided me. Jim has always been greatly invested in my success, and he has provided me with great professional skills. He has played a fundamental part in how much I have grown and learned during my graduate student years, as a researcher and as a person. I am very proud of having had Jim as my academic advisor, and I aim to become as excellent of a mentor as him.

I would also like to thank the other members of my committee: Richard N. Taylor, André van der Hoek, and Cristina Videira Lopes. They have also provided me with great support and advice to identify and seek my goals during my graduate student years. I would like to thank André van der Hoek for believing in my potential when I knocked on his door as a M.S. student who wanted to try out research by working on a project with him. I would also like to thank Richard Taylor, whose Software Engineering graduate course built up my interest for research in this field. Also, I would like to thank Crista Lopes for showing her support for my research work, which helped me keep pushing forward.

I would like to thank all the friends and colleagues that I have made at the University of California, Irvine. I would like to thank my lab colleagues Fang Deng, Nicholas DiGiuseppe, Vijay Palepu and Ethan Wessel, with whom I have shared many great research and life experiences. I would also like to thank my other friends in Irvine, who have been my family away from home, in no particular order: Pablo Díaz, Sherry Wang, Alegría Baquero, Ruy Cervantes, Nicolas Mangano, Kah Liu, Kristina Winbladhi, Maria Pau, John Garman, Sergi Pérez, Tiago Proença, Max Moura, Fernan Saiz, Guillermo Moreno, Ixone Díaz, Pere Margalef, Einar Mykletun, Núria Royo, Shane Stephens, Lídia Robert, and Rabih Zaouk.

Finally, I am also thankful for the financial support that multiple sources provided for my research. The work in this dissertation has been funded by the following sources: National Science Foundation CAREER CCF-1350837, National Science Foundation CCF-1116943, Google Faculty Research Award, and the University of California, Irvine, School of Information and Computer Sciences.
CURRICULUM VITAE

Francisco Javier Servant Cortés

EDUCATION

Ph.D. in Software Engineering 2009–2015
University of California, Irvine

M.S. in Information and Computer Sciences, Software track 2007–2009
University of California, Irvine

B.S. in Computer Science 2000–2005
University of Granada

RESEARCH EXPERIENCE

Graduate Research Assistant 2008–2015
University of California, Irvine

Research Intern 2011
Microsoft Research

Research & Development Intern 2008
DreamWorks Animation

PROFESSIONAL EXPERIENCE

Microsoft Corporation

Software Engineering Intern 2004
Valeo Lighting Systems

TEACHING EXPERIENCE

Teaching Assistant and Guest Lecturer 2011–2015
University of California, Irvine
REFEREED CONFERENCE PUBLICATIONS

Supporting Bug Investigation using History Analysis  
Nov 2013

Chronos: Visualizing Slices of Source-Code History  
Proceedings of the 1st IEEE Working Conference on Software Visualization, Tool Track (VISSOFT 2013)  
Sept 2013

History Slicing: Assisting Code-Evolution Tasks  
Nov 2012

WhoseFault: Automatic Developer-to-Fault Assignment Through Fault-Localization  
Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)  
June 2012

History Slicing  
Nov 2011

REFEREED WORKSHOP PUBLICATIONS

CASI: Preventing Indirect Conflicts through a Live Visualization  
Proceedings of the Workshop on Cooperative and Human Aspects of Software Engineering (CHASE 2010)  
May 2010

SELECTED AWARDS

Dean’s Fellowship, Donald Bren School of Information and Computer Sciences  
University of California, Irvine  
2009–2013

Caja Madrid Foundation Fellowship for Graduate Studies  
Caja Madrid Foundation  
2007–2009

PROFESSIONAL SERVICE

Reviewer  
Journal of Internet Services and Applications (JISA)  
2015

Reviewer  
Journal of Systems and Software (JSS)  
2014
External Reviewer 2014
International Symposium on the Foundations of Software Engineering (FSE)

External Reviewer 2014
International Conference on Software Engineering (ICSE)

Reviewer 2013
Central European Journal of Computer Science (CEJCS)

External Reviewer 2013
International Conference on Software Engineering (ICSE)

External Reviewer 2013
Working Conference on Software Visualization, Tool Track (VISSOFT)

External Reviewer 2013
Working Conference on Software Visualization, NIER Track (VISSOFT)

Program Committee 2012
International Conference on Program Comprehension (ICPC), Tool-Demonstration Track

Program Committee 2012
International Working Conference on Mining Software Repositories (MSR), Mining Challenge Track

External Reviewer 2012
International Conference on Software Engineering (ICSE)
ABSTRACT OF THE DISSERTATION

A Characterization and Partial Automation of the Multi-revision, Fine-grained Analysis of Code History as an Efficient and Accurate Mechanism to Support Software Development

By

Francisco Javier Servant Cortés

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2015

Associate Professor James A. Jones, Chair

Multiple studies found that developer questions about the history of code were among the hardest and most time-consuming to answer. In fact, the study of multi-revision, fine-grained code history with current approaches is a laborious, repetitive, and as such, error-prone process. In this dissertation, I posit the thesis that the multi-revision, fine-grained analysis of source-code history can be partially automated in a way that is efficient, that provides support to answer software development questions, and that accurately models source-code evolution. I present a series of techniques, tools and experiments that I developed and performed in order to evaluate this thesis. In the first step towards evaluating my thesis, I observe and conceptualize the process of multi-revision, fine-grained analysis of source-code history, as it is performed with the assistance of current revision-control tools. This conceptualization reveals the limitations in terms of efficiency of such process. I address the efficiency limitations of the multi-revision, fine-grained analysis of source-code history by creating Automatic History Slicing, a novel technique that enables developers to automatically obtain the subset of the history of a program that corresponds to any set of lines of code. Then, I also provide automatic support for answering developer questions by extending Automatic History Slicing into two other techniques and tools. The first one of these techniques is Chronos, which provides support for developers to answer ad-hoc questions...
about source code history by facilitating the visualization and investigation of the history of any set of lines of code. I also create a technique called WHOSEFAULT, which provides support for developers to answer a prevalent anticipated developer question: *who are the most suitable developers to fix a bug?* WHOSEFAULT automates all the steps of the multi-revision, fine-grained analysis of code history to provide a recommendation of the most suitable developers to fix a bug. Finally, I improve the accuracy of the multi-revision, fine-grained analysis of source-code history by creating *Fuzzy Automatic History Slicing*, a technique that allows the modeling and analysis of fine-grained code evolution with a novel fuzzy approach that recognizes the non-discrete nature of code evolution. The findings in this dissertation motivate future research in three directions: the empirical study of code evolution, the usage of code-history analysis for new applications, and the analysis of additional historical artifacts to support software development.
Chapter 1

Introduction

Developer productivity is affected by multiple factors [85]. One factor that largely impacts software development productivity is the time that software developers spend seeking information that they need in order to correctly perform software development tasks. Ko et al. [56] studied industrial software developers and found that developers could take up to 49 minutes to resolve a single information need. Furthermore, in many cases, resolving such information needs impacts the productivity of multiple people, since developers often ask other coworkers for help to resolve information needs [44]. In fact, Perry et al. [71] reported that developers spent half of their time interacting with colleagues.

Multiple studies identified the questions that developers ask during software development, e.g., Ko et al. [56] identified developer questions in collocated teams, Sillito et al. [81] identified questions that developers ask during a change task, and LaToza and Myers [59] identified hard-to-answer questions about code. Some examples of questions that developers ask are: How do I use this data structure or function?, Who, when, how, and why was this code changed over time?, and Why was this code implemented this way?. Particularly, Ko et al. [56] found that questions about why code had been implemented in a certain way were
among the most time-consuming to answer. Similarly, LaToza and Myers [59] found that questions that seek knowledge about the history of code — such as when, how, why, and by whom code was changed — were among the hardest ones to answer. Given the frequency and difficulty of such questions, mechanisms that efficiently support answering them can strongly and positively affect software-development productivity.

The main insight that motivates this work is the premise that the answer to many of the questions that developers ask may be found within the history of the source code of the program. In fact, many developer questions ask about the history of code itself [59]. For example, *Why was this code implemented this way?* may be answered by inspecting the evolution of that code over time. Furthermore, the analysis of code history may also be used to answer developer questions that do not ask directly about the history of source code. For example, *How do I use this data structure or function?* may also be answered by analyzing how the structure or function was used in past revisions of the program. Thus, we may be able to answer many of the questions that developers ask by analyzing the history of source code.

Multiple studies have found that developers require information about the history of code for multiple revisions, *e.g.*, a series of changes within a time frame [59, 46], and at a fine granularity, *e.g.*, code snippets [59, 56]. However, the current software-configuration-management (SCM) systems that store the history of source code — such as CVS [32], Subversion [5] and Git [82], lack functionality for an efficient analysis of multi-revision, fine-grained code history. Current SCM systems allow the query and exploration of source code history for a single revision at a time, and at a coarse granularity — file-level granularity. As a result, the analysis of the multi-revision history of source code at a fine granularity with current SCM systems requires substantial manual effort by the developer to: identify a line of code of interest, track and query each of its past revisions, and then to repeat and synthesize the results for all lines of interest. An additional consequence of such a manual and tedious
The process of studying fine-grained code history is that mistakes can be made at any step and, therefore, the studied code history may be inaccurate.

In this dissertation, I present the following thesis statement:

**Thesis:** The multi-revision, fine-grained analysis of source-code history can be partially automated in a way that is efficient, that provides support for answering developer questions, and that accurately models source-code evolution.

To address this thesis statement, I develop a series of techniques and tools, and I perform a series of experiments that I describe in the following chapters. Each chapter addresses one of the three hypotheses into which I divide my thesis:

- Chapter 4 addresses Hypothesis $H_1$: The multi-revision, fine-grained analysis of source-code history can be partially automated in a way that is efficient.
- Chapter 5 addresses Hypothesis $H_2$: The multi-revision, fine-grained analysis of source-code history can be partially automated in a way that provides support for answering developer questions.
- Chapter 6 addresses Hypothesis $H_3$: The multi-revision, fine-grained analysis of source-code history can be partially automated in a way that is accurate.

First, I conceptualize the process of multi-revision, fine-grained analysis of source-code history into a series of operations and products. I particularly define the concept of history slicing as a fundamental operation of the multi-revision, fine-grained analysis of source-code
history. I define **history slicing** as *the operation of obtaining the history of a given set of lines of code of interest.*

I address the efficiency limitations of performing history slicing with current revision-control systems by creating a novel technique called **Automatic History Slicing**. By providing an automatic process, automatic history slicing enables the efficient analysis of the history of any flexible set of lines of code, across any set of files and revisions of the program. I perform an experiment to evaluate the efficiency provided by automatic history slicing when compared to current approaches for obtaining the history of a set of lines of code. In my experiment, automatic history slicing improves the efficiency provided by existing approaches, since it requires developers to inspect up to three orders of magnitude less information to obtain the history of a set of lines of code.

I provide support for developers to answer software development questions by creating two automatic techniques and tools. I first create **Chronos** to provide developers with automatic support for answering questions about source code history. **Chronos** uses automatic history slicing to efficiently obtain the history of a set of lines of code, and it provides automated support for developers to interactively select the lines of code of interest and investigate their history. I perform an experiment to evaluate the support provided by **Chronos** for developers to answer questions about source code history. In this experiment, **Chronos** allows developers to answer a set of common questions about code history correctly in more than triple the cases and in around half the time than when they used current revision-control systems.

Next, I create **WhoseFault**, a technique that further automates the process of multi-revision, fine-grained analysis of source-code history to answer a prevalent developer question: *who are the most suitable developers to fix a bug?* **WhoseFault** analyzes the execution of a bug and analyzes the history of its executed lines of code to provide an automatic recommendation of the most suitable developers to fix it. In my experiments for evaluating
the support provided by WhoseFault, it provides at least the same — and in many cases better — effectiveness as existing techniques to identify the most suitable developers to fix a bug, and it removes the requirement that existing techniques impose of having a human previously write a description of the bug.

Finally, I address the accuracy limitations of existing approaches for history slicing by presenting a novel technique called Automatic Fuzzy History Slicing. Automatic fuzzy history slicing improves the accuracy with which fine-grained source-code history is modeled and analyzed by providing a fuzzy approach. Automatic fuzzy history slicing recognizes the non-discrete nature of code evolution, and it models the evolution of lines of code into others to different extents. In my experiments to evaluate the accuracy of automatic fuzzy history slicing, it provides a higher f-measure — *i.e.*, balance between precision and recall — of up to 0.29 — on a scale from 0.0 to 1.0 — than current discrete approaches for modeling fine-grained code history.
Chapter 2

The Process of Multi-revision, Fine-grained Analysis of Source-code History

The first contribution of this dissertation lies in the characterization of the process of multi-revision, fine-grained analysis of source-code history. I characterize the process of multi-revision, fine-grained analysis of source-code history into three main operations, as is depicted in Figure 2.1. The multi-revision, fine-grained analysis of source-code history may assist developers with answering software-development questions, therefore improving software development productivity.

Figure 2.1: The process of multi-revision, fine-grained analysis of source-code history
Let’s consider a scenario in which a developer has formulated a question that can be answered by the multi-revision, fine-grained analysis of source-code history, e.g., when, how, why, and by whom code was changed? This developer would first have to select the lines of code of interest for which she wants to study their history, then she would need to obtain the history of those lines of code of interest, and finally she would need to examine the obtained history of the lines of code of interest to answer her question.

These operations may be performed a different number of times, and in different orders. In some scenarios, executing these three operations once in the mentioned order will be enough to answer the question that the developer formulated. In other scenarios, after examining the history of the selected lines of code, the developer may want to go back to the first operation, refine her selection of lines of code of interest, and continue her analysis. Finally, developers may also iterate through the operations of obtaining the history of code and examining it, one revision at a time.

One fundamental insight of my research is that the operation of obtaining the history of a set of lines of code of interest can be made more efficient through automation. The performance of this operation manually by developers is highly inefficient and inaccurate — since developers can make mistakes at any step of the way.

In this dissertation, I present a partial automation of the process of multi-revision, fine-grained analysis of source-code history through a novel technique that automates the operation of obtaining the history of a set of lines of code. This automatic technique lets machines perform the repetitive, time-consuming and error-prone task of obtaining the history of the selected lines of code, and it lets developers focus on the tasks that involve complex reasoning — such as selecting the relevant lines of code for their analysis, and examining and interpreting their history.

Furthermore, for a specific developer question, I also present a novel technique that auto-
mates every step of the multi-revision, fine-grained analysis of source code history, \textit{i.e.}, the operation of selecting the lines of code of interest, the operation of obtaining the history of the lines of code of interest, and the operation of examining the history of such lines in order to finally provide a fully automatic answer to that developer question (see Chapter 5.3). This last technique opens the door for further research on the full automation of the multi-revision, fine-grained analysis of source code history to answer specific developer questions.
Chapter 3

History Slicing: The Process of Obtaining the History of a Set of Lines of Code

As I discussed in Chapter 2, improving the efficiency and accuracy of the operation of obtaining the history of a set of lines of code may have a strong impact on the overall process of multi-revision, fine-grained analysis of source-code history. In order to provide a better description of this operation, I characterize the different steps involved in it.

I characterize the operation of obtaining the history of a set of lines of code under the novel

Figure 3.1: Characterization of the operation of obtaining the history of a set of lines of code
concept of history slicing [75, 76]. This characterization is depicted in Figure 3.1. The history slicing operation takes as its input a history-slicing criterion, and it produces a history slice. I describe these concepts with a real-world example in the following sections.

In this dissertation, I also demonstrate how the analysis of the contents of history slices may enable developers to answer multiple questions, both about the history of code and about other aspects of software engineering. Additionally, the study of history slices may enable developers to answer both anticipated questions — by automatically analyzing the history slice — and ad-hoc questions — by manually studying the contents of the history slice.

3.1 History-slicing Criterion

I define history-slicing criterion as a set of lines of code of interest that have been selected to study their history. This selection may be performed by a developer, but also by an automatic algorithm. The history-slicing criterion is akin to the “slicing criterion” concept in program slicing.

The history-slicing criterion may contain any set of lines of code, contiguous or fragmented, from any file and revision or combination of files and revisions. To use a real-world example, let’s consider a history-slicing criterion that contains lines 999–1011 in revision 1.162 of file AjBuildManager.java from the AspectJ [1] open-source project. Figure 3.2 describes the history-slicing process for this real-world example.

3.2 History Slicing

I define history slicing as the process of obtaining the subset of the history of a program that corresponds to a set of lines of interest. Following the parlance of program slicing —
the process of selecting the subset of the program that is relevant for a given set of lines of code, history slicing is the process of selecting the subset of the program’s history that is relevant for a given set of lines of code.

The history slicing process involves two main operations: navigate the dimension of history, and navigate the dimension of the program. I represent these two operations in Figure 3.2, and I describe them in the following subsections.

### 3.2.1 Navigate the Dimension of History

One of the two main operations involved in history slicing is the navigation of the dimension of the history of the program to identify the revisions of interest for the selected history-slicing criterion. Developers continuously make changes to software projects, creating multiple revisions. In the general case, each revision will contain changes for only a small subset of the lines of code of the program. As a result, in general, only a subset of the revisions of the program will actually contain changes to any given set of lines of code. The process of navigating the dimension of history involves, for each line of code specified in the history-slicing criterion, the identification of those revisions that modified that line, between the starting and ending revision specified for that line.
Let’s consider the history of the only source-code file included in my example history-slicing criterion: `AjBuildManager.java`. Figure 3.2 shows the history of `AjBuildManager.java`, and it represents the history-slicing criterion as a gray highlight of the selected lines of code in revision 1.162. The history of the `AjBuildManager.java` file contains 162 revisions. However, out of those 162 revisions, only five of them (represented in Figure 3.2 as lifted from the stack of files) actually modified the lines of code selected in the history-slicing criterion (lines 999–1011): revisions 1.156, 1.134, 1.60, 1.14 and 1.1. Therefore, in order to study the history of this history-slicing criterion, only 5 out of the 162 revisions need to be studied.

3.2.2 Navigate the Dimension of the Program

The other main operation involved in history slicing is the navigation of the dimension of the program. This operation involves the inspection of the contents of each one of the revisions identified in the navigation of history to identify the lines of code of interest within that revision. In the same way that normally only a subset of the revisions of the program actually modified the lines of code specified in the history-slicing criterion, also only a small subset of the program corresponds to the lines specified in the history-slicing criterion on each identified revision. The process of navigating the dimension of the program involves, for each revision that modified the lines of code specified in the history-slicing criterion, the identification of the specific lines that represent their previous state within that revision.

In my example, for each one of revisions 1.156, 1.134, 1.60, 1.14 and 1.1 of file `AjBuildManager.java`, only a subset of the lines of code of that revision will correspond to a previous state of the lines of code specified in the history-slicing criterion *i.e.*, lines 999–1011 in revision 1.162. In order to study the history of this history-slicing criterion, only a subset of the lines of the file need to be studied in each identified revision.
### 3.3 History Slice

I define **history slice** as the *subset of the history of the program that corresponds to a set of lines of interest*. A history slice is the product of history slicing, and it represents the *complete and minimal* history of a given set of lines of code. The history slice for a history-slicing criterion contains every revision of the program in which any of its lines was modified, as well as their corresponding history-slicing snapshot in each one of those revisions.

I define **history-slicing snapshot** as the *set of lines of code in a particular revision that correspond, either directly or transitively, to the original lines in the history-slicing criterion*. In other words, a history-slicing snapshot represents a previous state of the lines in the history-slicing criterion.

Figure 3.3 shows the history slice that results from the history-slicing criterion in Table 4.2. The bottom part of Figure 3.3 shows the history-slicing snapshots inside the revisions that modified the lines of code in the history-slicing criterion. For example, given this history-slicing criterion, its history-slicing snapshot for revision 1.134 is represented by lines 940–948 and 950. The line numbers in a history-slicing snapshot may be different for each revision,
since an undetermined number of lines may be added and/or removed before and/or after it.
Chapter 4

The Multi-revision, Fine-grained Analysis of Source-code History Can Be Partially Automated in a Way That Is Efficient

The process of multi-revision, fine-grained analysis of source code history, which I characterized in Chapter 2, is enabled by the ability to obtain the subset of the history of the program that corresponds to a given set of lines of code, i.e., history slicing, which I characterized in Chapter 3. Unfortunately, the mechanisms that current SCM systems provide for history slicing are limited in terms of efficiency.

In this chapter, I first describe the efficiency limitations of current approaches for performing history slicing. Then, I create and describe a novel approach for automating the history slicing operation. Finally, I evaluate the efficiency benefits that automatic history slicing provides over existing history slicing approaches.
Table 4.1: Mechanisms provided by current approaches for history slicing

<table>
<thead>
<tr>
<th>Approach</th>
<th>Manual</th>
<th>Conventionally Assisted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate History Dimension</td>
<td>cvs/svn/git log</td>
<td>cvs annotate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>svn/git blame</td>
</tr>
<tr>
<td>Navigate Program Dimension</td>
<td>Scrolling</td>
<td>Ctrl + F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All-to-All diff approaches</td>
</tr>
<tr>
<td></td>
<td></td>
<td>One-to-One diff approaches</td>
</tr>
</tbody>
</table>

4.1 Efficiency Limitations of Current Approaches for History Slicing

Within the process of multi-revision, fine-grained analysis of source-code history, the history slicing operation may be performed with different levels of automation: manually, assisted by tools, or fully automatically. Today’s SCM systems provide some mechanisms that may assist the process of performing history slicing. I list these mechanisms in Table 4.1, and I classify them into mechanisms to navigate the history dimension and mechanisms to navigate the program dimension, as well as into fully manual and conventionally assisted mechanisms.

Despite such mechanisms, current SCM systems do not provide an automatic way of obtaining the complete history of an arbitrary set of lines of code. As a result, history slicing is not a trivial task to perform either manually or assisted by current SCM systems. Current SCM systems impose a time-consuming navigation of the dimension of history and the dimension of the program that iterates through the following four steps:

1. Navigate the history dimension, *i.e.*, retrieve the previous revision \( r \) of a file.
2. Navigate the program dimension, *i.e.*, find inside revision \( r \) the lines that correspond to the lines of interest.
3. Check the contents of those lines and identify whether they were modified.
4. If they were modified, save them. Return to Step 1 until all history is explored.
Depending on a developer’s level of expertise with today’s SCM systems, she may perform history slicing by using a fully manual approach or a more advanced, conventionally assisted approach, as well as a spectrum of approaches in between. Still, no matter the experience of a developer with SCM systems, the performance of an undetermined number of repetitions of multiple steps is likely to be much less efficient than a fully automatic technique for computing history slices.

4.1.1 Manual, Naïve History Slicing

I present this approach, (1) because it is the most straightforward solution and one that is likely to be employed by more novice developers who are unaware of advanced features of SCM systems and (2) because it clearly demonstrates the challenges brought by each of the four steps.

To determine the previous revision to the revision of interest, i.e., navigate the dimension of history (Step 1), a developer could manually retrieve every individual revision of the file(s) of interest from the SCM system. Of course, this requires an exorbitant amount of unnecessary work because only a subset of those revisions contains changes for the lines of interest. This inefficiency affects the search in the dimension of history (as indicated by the “History dimension” arrow in Figure 3.2).

To determine the history-slicing snapshot in a prior revision, i.e., navigate the dimension of the program (Step 2), the developer would, in the worst case, manually inspect the full contents of the files in order to find the position of the history-slicing snapshot. If the files are large, this step also involves a high amount of unnecessary work. This inefficiency affects the search in the dimension of the program (as indicated by the “Program dimension” arrow in Figure 3.2).
To determine whether there are changes in the history-slicing snapshot (Step 3), the developer would compare each pair of history-slicing snapshots. This comparison may not be straightforward when the snapshots are large, disjoint, or contain subtle and hard-to-notice changes.

Finally, to keep track of the entire history of the lines of interest (Step 4), the developer would need to keep a log of all the history-slicing snapshots for all the involved files. This log could be kept in a text editor, requiring application and context switching, and thereby imposing an additional overhead on the process.

For all of these reasons, following such a naïve approach can be extremely time consuming. I want to point out that, while this manual approach may often be an unrealistic scenario, developers are likely to follow it in cases when they don’t have much expertise with SCM systems or in cases where the task only involves a few changes in a few lines in a single file. Nonetheless, I present the manual approach mostly for illustrative purposes.

4.1.2 Conventionally Assisted History Slicing

Developers may use capabilities of SCM systems to support the previously-mentioned four steps for slicing history.

In order to navigate the dimension of history, i.e., determine the previous, modified revision (Step 1), the developer can utilize the annotate/blame feature on the revision of interest. Annotate will return, for every line in the file, the latest revision that modified it. Then, the developer would need to manually find the history-slicing snapshot inside annotate’s output. Since annotate may return different last revisions for different lines, the developer would select the most recent (highest) revision \( r \) in which any of the lines in the history-slicing snapshot was modified. This automation would ameliorate the inefficiency in the dimension
of time of the manual approach.

To navigate the dimension of the program, i.e., to find the history-slicing snapshot inside revision $r$ (Step 2), the developer may use the $Ctrl+F$ functionality in a text editor with the contents of the revision in order to navigate to the area of the file that contains the history-slicing snapshot. However, this option might still involve multiple attempts. Since developers would be searching in an older revision, they might be searching for words that were different then. Another shortcut for this step would be to run a line-mapping technique — such as $diff$ — over $r$ and $r - 1$, but the developer would still have to manually inspect the contents of its output. Additionally, for both $Ctrl+F$ and $diff$, in the case of fragmented history-slicing snapshots, each line of the history-slicing snapshot would have to be found individually and manually. While this automation mitigates the inefficiency in the dimension of the program of the manual approach, it is hard to predict by how much. In a worst-case scenario, the developer would still need to inspect the full contents of the file.

Determining whether the history-slicing snapshots are different (Step 3), will be straightforward, since $annotate$ will directly point developers to only those revisions that do contain changes to the history-slicing snapshot. However, some degree of inefficiency would be introduced in this step if the specific implementation of $annotate$ detects false modifications, such as modifications in white space. In such case, developers would still have to inspect some unnecessary revisions.

Keeping track of the history slice (Step 4), is still performed exactly as in the manual approach.

In summary, this approach assists the process of finding the history of a set of lines of code. However, despite some automation, it is still a highly manual process of performing multiple commands and correlating and interpreting their output. In the extreme case where the lines of interest are fragmented and scattered among multiple files, this process can be extremely
time consuming, even with such assistance.

4.2 Automatic History Slicing

In order to address the efficiency limitations provided by current existing approaches for history slicing (see Section 4.1), I present in this section a fully automatic history slicing technique [76].

My history slicing technique automatically tracks the lineage of every line of code in a program and enables developers to efficiently query and analyze such evolution. Automatic history slicing addresses the limitations of traditional SCM systems by providing the ability to (1) query at the line level, any arbitrary set of lines, across any set of files, and (2) automatically obtain the evolution of the program that corresponds to those lines. Automatic history slicing makes the history slicing process more efficient than current approaches because, unlike them, it automates the navigation of the dimension of the program, the navigation of the dimension of history, and the iteration between them.

The fine granularity and flexibility of automatic history slicing enables the analysis of the history of sections of code that are not automatically supported by current SCM systems.
or by current automated code-history analysis techniques. Some examples of code sections
for which automatic history slicing enables developers to study their history are: a code
snippet within a method, the lines of code that were executed by a bug, or the lines of code
that belong to an aspect. Additionally, automatic history slicing supports any other coarser
granularity as well, i.e., selecting all the lines of code within a method or file allows the
analysis of the history of that method or file.

I describe below my proposed approach for the automation of history slicing. My approach
to automating history slicing involves multiple steps, each of which can be parameterized in a
number of ways. The decision about how to implement each of these steps will be influenced
by the intended use of the history slice. This decision may also affect the contents of the
computed history slice. In any case, the history slice will contain the subset of the history
of the program that corresponds to a given set of lines of code. My approach to automatic
history slicing is depicted in Figure 4.1.

4.2.1 Build History Graph

The first step of my approach involves the creation of a history graph. This step retrieves and
processes every revision of the program from the revision-control system, and it produces a
history graph that models the complete history of every line of code of the program.

![History Graph](image)

Figure 4.2: History Graph.
A history graph is a multipartite graph where each part represents a revision of the code. Inside a part, each node represents a line of code in that revision. Each node is linked to only one node in the previous part and/or only one node in the following part. Additionally, each node can be labeled to store additional meta-data, such as authorship, time stamps, and log messages. Figure 4.2 shows a simple example of a history graph. In this figure, each node contains a label, which describes the operation that produced each line in each revision. In general, history graphs are similar to annotation graphs [92]. History graphs improve the precision of the “modification hunk”-granularity of annotation graphs, since they require a one-to-one node mapping between revisions.

The links between nodes are assigned by applying a line-mapping (i.e., fine-grained program-differencing) technique. Because existing line-mapping techniques vary in their power and flexibility, my approach allows the choice of line mapping to be customized according to the tasks at hand. For example, a simple plain-text differencing technique may be desirable if following the history of non-executable code, such as developer comments or XML metadata. Alternatively, a line-mapping technique based on the abstract syntax tree may be desirable to track changes that change the structure of the executable code.

Once the history graph is built, it can be used for the computation of any history slice. Building the history graph is a one-time expense and, as the code history grows, it can be extended with minimal effort. The history graph is stored in a relational database that can be queried both directly and through a supporting API.

**Line-mapping Technique**

In this section, I describe the line-mapping technique that I used to perform the experiments that evaluate my automatic history slicing technique. For every pair of consecutive revisions of the program, my line-mapping technique estimates which lines of code evolved into which
others and it uses that information to connect the edges that correspond to such code evolutions in the history graph. This line-mapping technique uses insights from two different line-mapping techniques from the research literature.

Similarly to the line-mapping technique proposed by Chen et al. [21], my line-mapping technique first performs a step to identify those lines of code that did not change at all between revisions, and connect them in the history graph. The goal of performing this step is to simplify the line-mapping problem to smaller blocks of code. In this step, I use the SCM system’s `diff` functionality to determine individual added, deleted, changed and unchanged lines, and label their nodes as such. Then, for those lines that `diff` identified to belong to a region of change, i.e., a change hunk, I apply a second step.

In the second step of my line-mapping technique, I estimate which lines of code evolved into which others within a change hunk. Similarly to the technique proposed by Williams and Spacco [89], I treat this problem as an assignment problem, which I solve by using the Kuhn-Munkres [57] combinatorial optimization algorithm. The reason behind this choice is that the line-mapping problem matches really well the definition of an assignment problem. Also, in my initial experiments, I observed that the Levenshtein distance better represented textual difference in source code than other possible approaches, such as longest common subsequence (LCS).

Thus, I first estimate the cost of assigning each line in the older revision to each line in the newer revision, and then estimate the assignment configuration that provides the minimum total cost. In order to solve this assignment problem, I apply the Kuhn-Munkres [57] combinatorial optimization algorithm. As optimization function, I use the sum of the cost of all the selected line assignments. I represent the cost of assigning one line of code to another as the Levenshtein distance between them. The Levenshtein distance [61] measures the number of changes that would have to be made to one sequence of characters to transform it into another.
Finally, I discard the identified line matches that have a high difference score. Previous work (e.g., [20, 89]) has determined the adequacy of a standard threshold of 0.4 for this purpose, which I also use. If the difference score of a pair is lower than the threshold, the pair is classified as a changed line. If the difference score is higher than the threshold, a pair is classified as the old line being deleted and the new one being added. I then use these classifications to decide which edges to draw between consecutive revisions in the history graph. I use this line-mapping technique to build the history graph for my experiments.

### 4.2.2 History-slicing Criterion

My automatic history slicing approach takes a history-slicing criterion as its input to define the lines of code of interest for which to obtain their history.

The lines in a history-slicing criterion may be few (e.g., a single line or a basic block of code) or numerous (e.g., a method or class); contiguous (e.g., a method) or fragmented (e.g., a dynamic slice); within a single file (e.g., a method) or across multiple files (e.g., a test case's...
statement-coverage set); and from any set of revisions (e.g., code selections from different releases of the program). This freedom is provided as a result of the fine grain in which the history graph stores the evolution of the source code.

A history-slicing criterion is specified as a set of (file name, line number, starting revision, ending revision) tuples. Each tuple represents a line of code of interest that is characterized by the two revisions of the program between which its history will be obtained — and likely later examined. The relationship between the starting revision and ending revision will determine the direction in which the history slicing process is performed, and the contents of the resulting history slice (see Section 4.2.3).

Table 4.2 shows a real-world example of a history-slicing criterion for the ASPECTJ [1] open-source project.

### 4.2.3 Analyze History Graph

Given a history-slicing criterion and a history graph, my approach analyzes the history graph to obtain the subset of the history of the program that corresponds to the given history-slicing criterion.

For every line of code specified in the history-slicing criterion, my approach traverses the history graph, starting from the nodes that were specified in the starting revision of the history-slicing criterion, following the edges that connect them to other nodes in consecutive revisions, until the ending revision is reached or the history of that line ends. During the traversal of the history graph, every visited node in the history graph in any revision that modified at least one line of code is marked for inclusion into the history slice. However, depending on the kind of history slice that is requested, this approach may vary (see Section 4.2.4).
The direction of this traversal will be determined by the relation between the starting revision and the ending revision: if the starting revision is higher than the ending revision, the traversal of the graph will be performed backwards, and if the starting revision is lower than the ending revision, the traversal of the graph will be performed forwards. In the backwards direction, the history of a line of code ends when the line was added, and in the forwards direction the history of the line ends when it was deleted. The analysis of the history graph is performed by interacting with the relational database that contains the history graph through SQL queries.

4.2.4 History Slice

The output of my automatic history slicing technique is a history slice, which contains the subset of the history of the program that corresponds to a given history-slicing criterion.

My technique allows for the computing of three different kinds of history slices, according to the revisions and lines of code that they contain for each revision:

History slice without context: It contains only those revisions that modified at least one of the lines of code in the history-slicing criterion, and only the modified lines in each history-slicing snapshot. This kind of history slice would be very useful when only the modifications to the selected code needs to be considered.

History slice with context: It contains only those revisions that modified at least one of the lines of code in the history-slicing criterion, and all the lines that correspond to the history-slicing criterion in each history-slicing snapshot. This is the kind of history slice that would be useful to analyze the modifications to the selected code of interest in the context of the remaining selected lines. I anticipate this kind of history slice to be the one computed in most cases.

Extended history slice: It contains every revision of the program, but only the lines that
correspond to the history-slicing criterion in each history-slicing snapshot. This kind of history slice may be useful for some visualizations, in which the history of code would need to be studied in a specific set of revisions, whether it was modified or not.

In the end, the resulting history slice is specified as a set of nodes and edges in the shape of a multipartite graph, which is a subset of the history graph. Nodes in a history slice are specified as \((file\ name, line\ number, revision, meta-data)\) tuples. The meta-data item in the history slice contains meta-data information about the status of the line of code in a revision, such as the authorship information, the change date, or the message with which it was committed to the revision-control system. The tuples inside a history slice have variable size, \(i.e.,\) may contain an undefined number of meta-data items.

History slices can be represented in different ways, depending on the task being performed. Some scenarios will require minimal information, such as statistics on the number of changes made on a set of lines, while other scenarios will require more context and therefore will need to display the history slice together with the lines that surround the snapshots. An example representation of a real-world history slice can be seen in Figure 3.3.

4.3 Evaluation of the Efficiency Provided by Automatic History Slicing

I performed an experiment to evaluate the efficiency provided by automatic history slicing. In this experiment, I measure the efficiency provided by a given history-slicing approach as a reverse metric of the effort that it requires. I measure the effort that a history-slicing approach requires in terms of the number of lines of code to which developers would be exposed during its performance. Thus, the goal of this experiment is to determine the degree to which extraneous information can be minimized by the use of history-slicing automation.
To evaluate the effectiveness provided by automatic history slicing, I define the following research questions:

**RQ1:** *How much does the automation of history slicing reduce the problem space in terms of the total number of lines of code needing to be examined?*

**RQ2:** *How much does the size of the history-slicing criterion impact the problem-space reduction (in both the history and program dimensions)?*

I describe the details of this experiment in the following subsections.

### 4.3.1 Experimental Design

To answer my research questions, I produced 1,000 random history-slicing criteria at each of four sizes and computed their history slices using four approaches to history slicing (discussed in Sections 4.1 and 3). In total, I studied a total of 4,000 different history-slicing criteria and 16,000 computed history slices. The goal of computing history slices with four approaches to history slicing is to approximate the benefits brought by the range of conventional approaches and my automatic history-slicing approach (Research Question RQ1). The goal of computing history slices with four, differently sized history-slicing criteria is to determine how the size of the history-slicing criteria affects the degree of benefit from automation (Research Question RQ2).

For each history-slicing criterion and treatment technique, I generated the resulting history slice. I then used the resulting history slice to compute the resulting dependent-variable metrics to determine the problem-space costs.
4.3.2 Experimental Subject

In this experiment, I used the ASPECTJ open-source project [1] as the code base over which to perform history slicing. ASPECTJ is an aspect-oriented extension to the Java programming language that consists of over 510,000 lines of code and has been in active development for more than 10 years.

4.3.3 Independent Variables

I experimented using two independent variables: history-slicing-criterion size and technique treatment. Each variable will be defined and motivated in turn.

Independent Variable 1: History-slicing-criterion Size

In order to determine the degree to which the initial history-slicing criterion size affects the problem space reduction, I varied the sizes of the randomly generated history-slicing criteria as such:

- **Size 10.** A set of 10 randomly selected contiguous lines of code. This size was chosen to approximate the size of a block of code (*e.g.*, an *if* block).
- **Size 20.** A set of 20 randomly selected contiguous lines of code. This size was chosen to approximate the size of a small method or function.
- **Size 50.** A set of 50 randomly selected contiguous lines of code. This size was chosen to approximate the size of a large method or a small class.
- **Fragmented.** A potentially fragmented set of varying numbers of lines of code. In order to select a fragmented — non-contiguous— set of lines of code, I selected the lines executed by a test case inside a file. I randomly chose a test case, generated its statement coverage, randomly chose one of the files that it executed, and selected the
lines executed in it as the fragmented slicing criterion.

History slicing criteria may span multiple files. However, for this experiment, each individual criterion is contained in a single file for the continuity requirement of criterion-size 10–50, and to limit the size of the fragmented criterion. Also, I discarded and replaced randomly generated criteria that resulted in no history, i.e., the code in the history-slicing criteria had no previous revisions. In such cases, I viewed the prospect of exploring history unnecessary and fruitless.

**Independent Variable 2: Technique Treatment**

In order to determine the degree to which the automation of history slicing can reduce the amount of information to which developers would be exposed, I parameterized its computation in the following four ways:

**Naïve.** Every revision of the file in the history-slicing criterion is fully examined to determine where and if any changes were made to the history-slicing criterion. This parameterization approximates the effort that would be necessary without any automation at all, and is presented as a baseline.

**Conventionally Assisted.** Only the revisions of the file that modified the history-slicing criterion are examined, but they are fully examined. This parameterization approximates history slicing with assistance of SCM tools, like `annotate`. For fragmented slicing criteria, such as a dynamic program slice, each line would be found individually and manually. Then, I consider that all lines in each identified revision are manually inspected.

**Automatic History Slicing with Context.** Only the revisions identified by the automatic history-slicing approach are examined, and only the lines inside the history-slicing snapshots are examined. In each such revision, at least one line changed in
the history of the slicing criterion. In addition, all lines that have an evolutionary relationship with the history-slicing criterion are included (as “context”) to provide a full view of each snapshot at each relevant revision.

**Automatic History Slicing without Context.** Only the revisions identified by the automatic history slicing approach are examined, and only the lines that changed in the snapshot are examined. For this treatment, the unchanged-yet-correlated code is omitted. Such a technique may be useful for subsequent automated analyses that utilize the output of automatic history slicing.

### 4.3.4 Dependent Variables

To assess the problem space size for a developer (or an automated client analysis) in interpreting a history slice, I measure the following three dependent variables for each computed history slice:

**Dependent Variable 1: Number of Revisions.** Number of revisions in the history of the history-slicing criterion that need to be examined. This is computed as the number of revisions included in the resulting history slice.

**Dependent Variable 2: Mean Number of Lines per Revision.** Mean number of lines of code that a developer would need to inspect in each revision. This is computed as the mean number of lines for any given revision inside the history slice, aggregated across all relevant revisions, according to the treatment technique.

**Dependent Variable 3: Total Number of Lines.** Total number of lines of code that a developer would need to examine across all relevant revisions. This is computed as the sum of all examined lines across all relevant revisions, according to the treatment technique. This serves as a proxy measure of the total amount of work that a developer would need to expend to fully explore and process the history slice.
Table 4.3: Amount of information to which developers would be exposed for different history-slicing approaches. Each row contains the mean value for 1,000 studied history slices. Lower values mean higher efficiency.

<table>
<thead>
<tr>
<th>History-slicing-criterion Size</th>
<th>Technique Treatment</th>
<th>Number of revisions</th>
<th>Mean number of lines per revision</th>
<th>Total number of lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Naïve</td>
<td>29.16</td>
<td>1,177.44</td>
<td>34,334.29</td>
</tr>
<tr>
<td></td>
<td>Conventionally Assisted</td>
<td>3.17</td>
<td>783.04</td>
<td>2,482.24</td>
</tr>
<tr>
<td></td>
<td>Automatic History Slicing with Context</td>
<td>3.17</td>
<td>10.01</td>
<td>31.74</td>
</tr>
<tr>
<td></td>
<td>Automatic History Slicing without Context</td>
<td>3.17</td>
<td>3.54</td>
<td>11.24</td>
</tr>
<tr>
<td>20</td>
<td>Naïve</td>
<td>30.58</td>
<td>1,204.98</td>
<td>36,848.30</td>
</tr>
<tr>
<td></td>
<td>Conventionally Assisted</td>
<td>4.03</td>
<td>867.58</td>
<td>3,496.36</td>
</tr>
<tr>
<td></td>
<td>Automatic History Slicing with Context</td>
<td>4.03</td>
<td>19.98</td>
<td>80.52</td>
</tr>
<tr>
<td></td>
<td>Automatic History Slicing without Context</td>
<td>4.03</td>
<td>5.58</td>
<td>22.51</td>
</tr>
<tr>
<td>50</td>
<td>Naïve</td>
<td>34.62</td>
<td>1,350.64</td>
<td>46,759.26</td>
</tr>
<tr>
<td></td>
<td>Conventionally Assisted</td>
<td>5.92</td>
<td>978.00</td>
<td>5,789.75</td>
</tr>
<tr>
<td></td>
<td>Automatic History Slicing with Context</td>
<td>5.92</td>
<td>49.96</td>
<td>295.75</td>
</tr>
<tr>
<td></td>
<td>Automatic History Slicing without Context</td>
<td>5.92</td>
<td>9.54</td>
<td>56.47</td>
</tr>
<tr>
<td>Fragmented</td>
<td>Naïve</td>
<td>23.95</td>
<td>840.03</td>
<td>20,118.83</td>
</tr>
<tr>
<td>Fragmented</td>
<td>Conventionally Assisted</td>
<td>4.41</td>
<td>567.61</td>
<td>2,503.18</td>
</tr>
<tr>
<td>Fragmented</td>
<td>Automatic History Slicing with Context</td>
<td>4.41</td>
<td>60.65</td>
<td>267.46</td>
</tr>
<tr>
<td>Fragmented</td>
<td>Automatic History Slicing without Context</td>
<td>4.41</td>
<td>6.40</td>
<td>28.24</td>
</tr>
</tbody>
</table>
4.3.5 Results

After computing the values of the dependent variables for each combination of history-slicing-criterion size and treatment technique, I averaged them and obtained the results displayed in Table 4.3.

In this experiment, my automatic history-slicing approach improved the efficiency provided by existing approaches by \textit{up to three orders of magnitude}. For every history-slicing-criterion size studied, I observed effort savings of an additional order of magnitude for \textit{every} additional level of automation — as measured by the cumulative number of lines of code needed to be examined and processed. Particularly, the \textit{Automatic History Slicing without Context} approach reduced the problem space by \textit{three} orders of magnitude over the \textit{Naïve} approach.

I focus my analysis of the obtained results on the value of the “Total number of lines” variable, since this is a proxy measure of the amount of work required by the developer to process a history slice. The value of this variable represents the size of the problem space when a developer searches for the history of a history-slicing criterion by following a given approach. I plot the values of “Total number of lines” in Figure 4.3.

In Figure 4.3, we can observe that a developer would have to search inside a median problem space of 46,759 lines of code when trying to find the history of 50 contiguous lines of code in a file in the \textsc{AspectJ} project by manually inspecting every complete revision of the file. Such scenarios are likely unrealistic, as any developer would likely give up very quickly after starting the \textit{Naïve} approach. I do not expect developers to actually follow this approach — I present it to demonstrate the size of the original problem space.

The current practice is probably to use tool-supported approaches, like \textit{Conventionally Assisted}, which reduces the problem space by one order of magnitude. For a slicing criterion of 50 lines, developers would face an average problem space of 5,790 lines of code. The reduc-
Figure 4.3: Mean total number of lines to which a developer would be exposed using each history-slicing approach. Automatic History Slicing (AHS) drastically reduces the expense for a developer. Note that the columns representing the Naïve approach far exceed the bounds of the chart.
tion of the problem space provided by the *Conventionally Assisted* approach result from the savings in the dimension of history, *i.e.*, fewer revisions. However, developers still may need to inspect many lines of each revision, *i.e.*, the program dimension, in order to correlate each line of each snapshot with their corresponding earlier and later revisions.

The *Automatic History Slicing with Context* approach also provides an additional reduction of one order of magnitude in the problem space over the *Conventionally Assisted* approach. For a slicing criterion of 50 lines, the average problem space is reduced to 296 lines of code. This approach provides savings in both the dimensions of history and the program, because it reduces the number of revisions that need to be inspected and the number of lines that need to be inspected inside each revision.

Finally, the *Automatic History Slicing without Context* approach also provides yet another reduction of one order of magnitude over the *Automatic History Slicing with Context* approach. For a slicing criterion of 50 lines, the average problem space gets reduced to 56 lines of code. In this case, the full problem space represents the solution that developers are looking for if they are interested only in the lines of the snapshots that actually changed. This approach provides further savings, because it further reduces the number of lines that need to be inspected inside each revision.

In conclusion, I provide below my answer to the research questions posed in this experiment:

**RQ1:** *How much does the automation of history slicing reduce the problem space in terms of the total number of lines of code needing to be examined?*

*Automating the computation of history slices reduced the problem space needed to be examined by developers by up to three orders of magnitude.*
RQ2: How much does the size of the history-slicing criterion impact the problem-space reduction (in both the history and program dimensions)?

Larger history-slicing criteria involved a larger cost in their computation, although such cost increases were minimal, especially when considering the automated approaches.

Overall, I find these results as strong evidence that: (1) the task of computing history slices is a non-trivial task for developers to compute on their own with current approaches, and (2) automatic history slicing enables developers to obtain the history of a set of lines of code in an efficient manner.

4.3.6 Threats to Validity

In this section, I discuss some threats to the validity of the results obtained in this experiment.

An external threat to the validity of this evaluation is that the results may not hold for other codebases. Despite the fact that I studied only one codebase — AspectJ — I do not consider this a significant threat. The main factors that affect history slicing are the age of the codebase and the size of the history-slicing criterion, regardless of the nature or functionality of the codebase. I experimented with the size of the history-slicing criterion and found an effect that easily scales with criteria size. And, given the random selection of 4,000 history-slicing criteria, history slices were computed over files that were brand new, as well as other files that were nearly a decade old. Regardless of the age of the code sliced, the automatic history slicing technique computed quickly (the maximum was 30 seconds in my unoptimized tool). Moreover, older codebases and larger criteria will likely affect the conventional SCM approaches much more than the automated history-slicing technique.

An internal threat to validity is that my implementation of automatic history slicing uses my
textual-line mapping technique, which may not always identify the correct line equivalence in between revisions, *e.g.*, when a method is moved to another file. However, this limitation affects both my automatic approach and the manual approaches to history slicing, since traditional SCM tools do not provide any mechanism for tracking movements of code in between files. As a consequence, I believe that this limitation does not affect the results of the experiment. Moreover, my automatic history slicing approach may address this limitation because it allows other line-mapping techniques, *e.g.*, [47, 27]. Additionally, no matter which line-mapping technique is used by automatic history slicing, the results of this evaluation should not vary significantly, since the efficiency improvement provided by automatic history slicing is produced mainly as a result of the automation of the history slicing process.

A threat to construct validity is that I measured developer expense in terms of the number of lines that need to be examined. While I use the metric of the number of lines to be examined as a proxy measure for developer effort, I posit that it is actually indicative of the amount of effort that would be needed by a developer — more lines will require the developer to inspect those lines, leading to greater effort. Regardless, I intentionally conducted another experiment to determine if the proxy measure of effort manifests for real developers on real code-evolution tasks (see Section 5.4).
Chapter 5

The Multi-revision, Fine-grained Analysis of Source-code History Can Be Partially Automated in a Way That Provides Support for Answering Developer Questions

In Chapter 4, I presented my novel Automatic History Slicing technique to partially automate the process of multi-revision, fine-grained analysis of source-code history in a way that improves the efficiency of current approaches.

In this chapter, I present two novel techniques that extend Automatic History Slicing to provide support for answering software developer questions. In my first technique, CHRONOS, I apply Automatic History Slicing to provide support for developers to answer ad-hoc questions about source code history. In my second technique, WHOSEFAULT, I further automate
the process of multi-revision, fine-grained code-history analysis to provide support for developers to answer a prevalent anticipated developer question: \textit{who are the most suitable developers to fix a bug?} \textsc{WhoseFault} automates all three steps of the multi-revision, fine-grained code-history analysis: selecting the lines of code of interest, obtaining the history of those lines, and examining such history.

I first motivate the need for developers to answer questions about source code history and for identifying the most suitable developers to fix a bug. Then, I describe the two techniques that I created to support developers in answering these questions: \textsc{Chronos} and \textsc{WhoseFault}. Finally, I evaluate the support that \textsc{Chronos} and \textsc{WhoseFault} provide.

\section{Motivation and Background}

Software developers spend a large amount of effort trying to answer a variety of questions during the software development process. In order to answer some of these questions, developers need to perform multi-revision, fine-grained analysis of source-code history, particularly to answer questions about source code history itself.

In the following subsections, I describe some scenarios in which the multi-revision, fine-grained analysis of source-code history may support software developers in answering their questions. I also describe for each scenario the current approaches that developers follow to answer their questions.

First, I describe scenarios in which software developers need to answer ad-hoc questions about source code history. Then, I motivate the need for developers to answer a common anticipated developer question: \textit{who are the most suitable developers to fix a bug?}
5.1.1 Ad-hoc Developer Questions about Source Code History

Multiple researchers observed software developers in the workplace and found developers need to answer many questions while performing software development tasks [56, 59, 14, 16, 39]. A study by LaToza et al. [59] found that developers frequently asked questions about the history of source code. LaToza et al. also found that developers wanted to know the latest changes as well as the entire history of code at the granularity level of a code snippet. For example, two different studies by LaToza et al. [60] and Ko et al. [56] found that developers often asked questions about why a snippet of source code had been implemented in a specific way. They also found that resolving such questions about the rationale of source code was highly time-consuming.

The goal of the research in this dissertation is to improve software developer productivity by providing automated support for answering the questions that developers ask. The insight behind this research is that many of the questions that developers ask can be answered by analyzing the history of the source code of the program. For example, in the previously mentioned question of why code was implemented in a specific way, a mechanism that can automatically provide the code’s creation and evolutionary history may help developers to understand such design-rationale motivations and the context in which they were made.

The ability to explore the entire history of a set of lines of code may benefit developers in a wide variety of scenarios. Consider an example scenario in which a developer wants to find an earlier implementation of a specific piece of code, for example, a loop body. In this example situation, our developer implemented multiple versions of the functionality performed by that loop throughout the history of the project. At some point, she realizes that a specific earlier implementation better suits the needs of the program, but she can’t remember which revision contains the desired implementation. In this situation, finding which revisions contain changes for that specific loop body and the corresponding lines to
that piece of code in each of those revisions can be a tedious task if performed with today’s SCM tools, unless log messages were quite explicit and detailed (and, even still, there would likely be an overabundance of log messages to read to find the specific revision).

In another example scenario, a developer may want to explore the parallel history of multiple segments of source code in order to find out whether and when they were modified together. Zimmermann et al. found that multiple software artifacts being committed together to the source code repository is a signal of such entities being dependent on each other. This dependency is the definition for evolutionary coupling [93]. Another motivation for tracking commits across multiple arbitrary segments of code is to help assess code clone risks. Bakota [9] found that segments of code that are identified as code clones often were modified together, and that violations of this pattern can indicate possible problems. In each of these cases, extracting and presenting a detailed exploration of the history of two pieces of code (for example, two small methods in two large files) would be a highly repetitive and time-consuming task if performed with today’s SCM tools.

In a final example scenario, a project manager may need to know which developers ever modified a specific segment of the source code, for example, multiple methods that interact to form a specific functionality. LaToza et al. found that developers asked questions about the authorship of code and their teammates [59], and a number of researchers proposed approaches for mapping “expert” developers to components of source code, e.g., [66, 68, 52, 34, 74, 36]. However, if a developer wanted to perform a detailed exploration through the history of when, how and why each developer made changes to that segment of the source code, she would again need to perform a tedious task with today’s SCM tools.

A common characteristic of each of these three scenarios is that they are difficult to answer with today’s SCM tools, for three reasons:

1. they require the exploration of multiple, non-contiguous, fine-grained sets of lines of
code — potentially across multiple files;

2. they require deep exploration of multiple revisions of code in the history — *i.e.*, not simply a pairwise *diff*; and

3. they require the recognition of patterns and characteristics across potentially long epochs of the projects’ life.

In fact, I conjecture that developers often give up on answering such code-evolution questions. That is, because these questions are exorbitantly time consuming to answer with today’s SCM tools, the possibility for successfully and accurately answering them is considered infeasible.

Some researchers proposed automatic approaches to answering specific developer questions at specific, medium-to-high, source-code granularities, *e.g.*, Fritz *et al.* [34] identified the most knowledgeable people for source code classes and methods. However, software developers:

1. need to answer a variety of ad-hoc questions, not only anticipated questions; and

2. need to answer questions about flexibly-defined, fine-grained source code selections, *e.g.*, multiple code snippets that interact for the execution of a feature.

In this dissertation, I present automatic techniques to assist developers in efficiently and accurately answering both ad-hoc and anticipated questions about any flexibly-defined set of lines of code.
Figure 5.1: Conceptual depiction of the bug-assignment process. This process matches the expertise needed to fix the bug with developer expertise to recommend developers for the bug-fixing task.

Figure 5.2: My new execution-based bug-assignment approach versus existing natural-language-based approaches
5.1.2 Anticipated Developer Question: *Who are the most suitable developers to fix a bug?*

A common anticipated software developer question in the majority of software projects is: *Who are the most suitable developers to fix a bug?* Software projects continuously receive reports of new bugs that have to be assigned to the most suitable person to investigate, diagnose, and fix them. I refer to the process of recommending developer assignments to bug-fixing tasks as *bug assignment*.

Bug assignment — regardless of whether it is performed manually or automatically — works by assessing developer expertise and matching the developers with the expertise required for the bugs being treated. Knowledge of the current bug may be expressed in a number of different ways: *e.g.*, in a bug report description, in an execution trace, in a stack trace, or simply in the mind of the bug’s witness. Similarly, the expertise of developers can be expressed in a number of ways, such as self-reported skills on a resumé, a set of bugs that were fixed in the past, or contributions to a source-code repository.

In all such cases, knowledge of the current bug is used to find developers with the expertise to investigate and fix it. Figure 5.1 depicts this process. In the fictitious scenario depicted in the figure, a bug causes a crash in *Subsystem XYZ*. The developers working on the software project, Alice (represented by “A”) and Bob (represented by “B”), each have knowledge, past experiences, and past contributions to the project, which I will abstractly refer to as *expertise*. The assignment of developers to the bug is performed by finding the best matches between the knowledge of the bug, which can be considered as the *expertise needed to fix the bug*, and the expertise that is offered by the developers, which can be considered as the *expertise available to fix the bug*. As such, the recommended bug assignment contains the best match between the expertise needed and the expertise available. In the figure, Alice is recommended because she has expertise with *Subsystem XYZ*, which is where the program
crashed, and Bob does not. With the available information, Alice is determined to have the best matching expertise to fix the bug.

In current practice, developers are typically assigned manually to bugs. Knowledge of the bug is usually personally experienced, revealed by a failed test case, or described by a bug report. Similarly, the knowledge of “which developer knows what” is tacit — learned through interactions with the developers. Unfortunately, such manual assignments are inefficient and error-prone. As a result, sometimes a bug is assigned to a person who does not have the appropriate expertise to diagnose or fix it. In such case, the bug is re-assigned — and re-investigated — until a suitable person is found [38, 49], which incurs an additional delay in the bug-fixing process.

To assist with these manual assignments, researchers created automatic bug-assignment recommendation systems that typically use natural-language representations of the current bug (such as a bug-report description) and natural-language representations of developer expertise (such as past fixed bug reports or the natural language that is included in source-code contributions) to automatically recommend developers. The intuition that informs these automatic recommendation systems is that the words, or terms, used to describe the current bug are likely to be similar to the terms used in some past bug reports or source code, and the developers who demonstrated enough expertise to fix those similarly worded bug reports or source code in the past likely have appropriate expertise to fix this current bug.

The intuition behind the approach that I present in this chapter for automatic bug assignment is that the history of the changes that developers performed over the source code is descriptive of their development expertise. My approach also follows the idea that other evidence of bugs — namely captured execution meta-data — can also be used to recommend bug assignments. Consequently, my approach recommends developers to fix a bug according to their performed changes over the source code that was exercised by the bug-revealing executions. Figure 5.2 depicts both my execution-based approach (above) and the existing natural-language-based
approaches (below) at a conceptual level.

In my execution-based approach, execution information is recorded at runtime while observing the bug’s behavior. This recorded execution information is input to the bug-assignment technique. The technique, itself, then accesses the history of the code in the revision-control system from all developers (Step 1), performs an analysis of the code history to model developer expertise (Step 2), localizes the potentially buggy code locations to model the expertise needed for the bug (Step 3), and compares the available-developer-expertise models with the needed-bug-expertise model (Step 4) to generate the bug-assignment recommendation.

Similarly, the existing natural-language-based approaches follow the same four abstract steps — depicted in the lower portion of Figure 5.2. The primary difference of these two approaches is the form of the artifacts and expertise models: natural-language terms versus code locations and history. Moreover, the natural-language-based approach requires the bug’s witness to write a natural-language description of the observed buggy behavior. In contrast, the execution-based approach requires some form of automatic runtime monitoring, such as coverage instrumentation, logging, or crash reporting.

These differences inform the potential applicability of these different assignment recommendation systems. Natural-language-based approaches may be a preferred choice when customers submit a bug report to the bug-reporting system — particularly if the failure is difficult to recreate. In contrast, execution-based approaches may be a preferred choice when failures are observed during automated testing: the entire process can be automated without the need for a human-written bug report. Moreover, even if a bug report eventually needs to be written, the automatic execution-based recommendation can inform who would be the best suited developer to write that bug report.

With these scenarios in mind, my goal is to present an automatic bug-assignment approach that can provide bug-assignment recommendations in situations when a natural-language
bug report is unavailable, and further, to determine if those recommendations are sufficiently similar in terms of accuracy (i.e., effectiveness) and cost (i.e., efficiency) as state-of-the-art natural-language-based approaches. As such, if both approaches produce useful recommendations, it may be appropriate to simply use the approach that applies to the available information, and when both forms of information are available, it may be appropriate to use them both to inform the final assignment.

5.2 **Chronos**

In this chapter, I present **Chronos** [76, 78], a novel technique and tool that partially automates the multi-revision, fine-grained analysis of source-code history to support developers in answering questions about source code history. The multi-revision, fine-grained analysis of source-code history is composed of three steps, as I presented in Figure 2.1. **Chronos** provides full automation for the history slicing step with the technique that I presented in Section 4.1. **Chronos** also facilitates the performance of the two other steps: selecting the lines of code of interest, and examining the history of the selected lines of code. I describe the steps involved in **Chronos** in the following sections.

5.2.1 **Select Lines of Code of Interest**

**Chronos** includes functionality to support the process of selecting the lines of code of interest in code-editing windows within Eclipse. I implemented **Chronos** in Java as an Eclipse plug-in and it currently supports the CVS, Subversion and Git revision-control systems.

A developer can open any revision of any file, select any set of lines, and right-click to reveal a contextual menu that includes an option to **Add to history-slicing criterion**. As such, any set of lines, contiguous or fragmented, across any number of files or revisions, can
be added to the history-slicing criterion.

5.2.2 Examine History of Lines of Code of Interest

Once the lines of interest have been selected, CHRONOS allows developers to run the Display history slice command, which will automatically compute the history slice for the selected lines of code and visualize their corresponding history slice. This visualization supports developers in the process of examining history slices.

My goal with CHRONOS is to allow general-purpose, ad-hoc explorations of the history of source code. CHRONOS can display the history of any set of lines of code, as well as multiple selections of groups of code lines, i.e., history-slicing criteria. Then, CHRONOS allows developers to interact with this visualization for examining the computed history of
Figure 5.4: CHRONOS visualizing a history slice. CHRONOS allows zooming at any level over any area of the visualization.

Figure 5.5: A global, multi-selection timeline visualization supports pattern recognition and discovery.

code. Figure 5.4 depicts an example of how CHRONOS displays the history of two code selections. Below, I describe the components of the visualization and the interaction options in CHRONOS.

**Components**

Figure 5.4 is a screen-shot of the history-slice visualization in CHRONOS, which is composed of two main parts. At the top, the two horizontal gray bars represent the global timeline. Below the global timeline, are two individual areas, each of which displays the history of one code selection. If more snippets of code had been selected, there would be more individual areas with the history of the corresponding snippets, and the global timeline would contain as many horizontal gray bars as the number of snippets of code selected.
Global, Multi-selection Timeline Visualization. The top area of the visualization displays a set of timelines — one for each one of the code selections. A zoomed-in view of the global timeline is depicted in Figure 5.5. The goal of the global timeline is to allow developers to discover patterns in the evolution of the selected snippets of code. In the global timeline, a developer could quickly answer questions such as: “Was a code snippet modified at a particular point in time but not in others?”, or “When were two code snippets changed together, and when were they changed separately?”

The top part of the global timeline is annotated with a ruler of equidistant ticks that represent months in time. Time is represented in increasing order from left to right, having the latest dates on the rightmost side of the timeline. For example, Figure 5.5 shows information from July 2005 until December 2005.

The upper gray bar represents the history of the first code selection and the lower gray bar represents the history of the next code selection. A blue mark inside a gray bar denotes a change to its code selection at that time. Also, users can highlight these blue marks to show meta-data about a change — particularly its date, author and commit message. Figure 5.5 shows a change highlighted in orange and its meta-data also in orange text.

A developer could quickly refer to the meta-data of specific changes for a better understanding of an observed change pattern. Additionally, if the developer wants to perform a more in-depth investigation of the code’s evolution, she could also explore the actual contents of a change by looking at the individual histories that are depicted below the global timeline.

Individual Histories. The second component of the visualization of CHRONOS contains the individual histories of the code selections. I show in Figure 5.6 a zoomed-in view from Figure 5.4 into the individual history of a code selection. Each individual history is in turn composed of two parts: the individual timeline at the top and the history-slicing snapshots at the bottom.
Figure 5.6: The individual history of a code selection provides semantically rich detail of code changes and supporting meta-data

CHRONOS also displays the history-slicing snapshots side by side to allow users to easily compare corresponding lines of code between revisions. Then, to allow users to get a sense of the time that passed between changes, CHRONOS also includes individual timelines at the top of the visualization. Since the spacing between changes in the timeline and in the history-slicing snapshots areas is different, CHRONOS connects changes in the timeline with their corresponding history-slicing snapshot using a blue line. This connection is the only difference between the timeline for a code selection inside the individual history and inside the global timeline.

Similarly to the global timeline, changes can be highlighted in the individual timeline for a quick view of their meta-data and for an easier identification of their corresponding history-slicing snapshot. A highlighted change and its connection to its corresponding history-slicing snapshot can be seen in Figure 5.6 in orange. The individual timeline and the global timeline are synchronized, so that if one of them is highlighted, the other one is also highlighted, which can be seen in Figure 5.4.

The individual history of a set of lines of code contains as many history-slicing snapshots as the number of revisions that contain changes for them. If a file has a revision in which the selected lines of code were not changed, that revision is not displayed by CHRONOS.
visualizes *history slices with context* (see Section 4.2.4). Each history-slicing snapshot is represented with gray text for those lines of code that were selected but not modified in that history-slicing snapshot, and with blue text for the lines of code that were selected and were modified in that history-slicing snapshot. Consecutive history-slicing snapshots are represented side by side and aligned to each other for an easier comparison. For example, in Figure 5.6, lines 1131–1135 were added in August 8th, 2005. We know that these lines were added because there are no lines that correspond to them in the previous history-slicing snapshot of July 20th, 2005.

Finally, each history-slicing snapshot is also annotated with meta-data of the change in green: the date, author, and commit message. The goal of including the meta-data of a change on top of the history-slicing snapshot is to aid in the understanding of the rationale of changes. By looking at a history-slicing snapshot, developers can see at a glance not only the changes that were performed in that revision, but also additional information that may help them understand why the changes were performed.

**Interaction**

**Chronos** allows three mechanisms for interaction: zooming, panning, and highlighting.

**Zooming and Panning.** In order to provide users with both a summarized view of the history of source code and a detailed view of the specific changes performed as well as their meta-data, **Chronos** supports panning and zooming on a scalable vector graphic visualization. When **Chronos** is executed, it provides a complete view of the individual histories that correspond to all the sets of lines of code selected, and all the snapshots that exist for them. This summarized view allows developers to have a first quick view of the timing of changes for each individual history and how many lines were changed in each snapshot. An example of this view is depicted in Figure 5.4. Then, **Chronos** allows
zooming without loss of quality. In the same manner, CHRONOS allows users to move to any part of the visualization to focus on and explore different aspects of it. Examples of detailed views are Figures 5.5 and 5.6.

**Highlighting.** As mentioned before, CHRONOS allows the user to highlight a change when they hover the mouse over it. When a change is highlighted, CHRONOS shows the meta-data corresponding to it and changes the color of the link to its corresponding snapshot to orange. With this feature, users can see properties of changes quickly from the timeline without having to pan and zoom to re-focus to the snapshot. In addition, highlighting changes makes it easier to follow the connection and identify their corresponding snapshot.

### 5.3 WhoseFault

In this section, I present WhoseFault [77, 79], a technique that provides a fully automatic answer to a prevalent anticipated developer question — *Who are the most suitable people to fix a bug?* — by automating all the steps of the multi-revision, fine-grained code-history analysis. I depicted all such steps in Figure 2.1. In order to provide its automatic answer, WhoseFault *selects the set of lines of interest* by analyzing the lines of code that were exercised by the bug, *obtains the history of the lines of code of interest* by using automatic history slicing, and it *examines the history of the lines of code of interest* with an automatic algorithm that analyzes the history-slicing criterion and history slice.

Current automated bug-assignment techniques model the expertise required to fix a bug by analyzing a human-written description of the bug. Unlike existing techniques, WhoseFault models the expertise required to fix a bug by analyzing the code exercised in its execution. As a result, WhoseFault can be applied in situations when a bug report is not yet available, and as soon as the bug is witnessed. Erroneous executions — such as testing failures — often
provide the first evidence of a bug in software, irrespective of whether a bug report exists for it or not. In addition, WhoseFault includes a fault-localization step, and thus provides the assignee with diagnostic information to aid the investigation. I describe the components of WhoseFault in more detail in the following subsections.

5.3.1 Select Lines of Code of Interest

WhoseFault automates the process of selecting the lines of interest. For the purpose of recommending the most suitable developers to fix a bug, WhoseFault considers that the lines of code of interest are those that were exercised by the bug. By selecting such lines, WhoseFault is also performing the process of modeling the expertise required to fix the bug, as described in Section 5.1.2 and depicted in Figure 5.2.

The reasoning for this step is that, in order to fix a bug, developers require higher expertise in the lines of code that more likely contain the bug, and lower expertise in the lines of code that less likely contain the bug. Therefore, WhoseFault selects a fuzzy history-slicing criterion with all the lines of code of the program, each of which is annotated with a weight that represents its suspiciousness score, i.e., its relative likelihood of containing the bug. Suspiciousness scores range from 0 — for the lowest likelihood — to 1 — for the highest likelihood. Then, in order to give some relevance to expertise over the whole code base, WhoseFault includes a minimum relevance of infinitesimal value $\epsilon$ for every line of code. In the implementation of WhoseFault that I used for my experiments, I used $\epsilon = 0.01$.

WhoseFault computes the relative likelihood of every line of code to contain the bug by using a spectra-based fault localization technique. Since spectra-based fault localization techniques only require executions of the program, this approach can provide developer-to-bug recommendations when a bug report is not available.
The intuition behind spectra-based fault localization techniques is that the lines of code that are mostly exercised by bug-revealing executions more likely contain the bug. By using an instrumented version of the program, I capture the coverage of its executions. The coverage of an execution contains a list of the specific lines of code that were exercised by it. I capture both bug-revealing executions and executions that produced the expected output, *e.g.*, all the passing test cases of the test suite. Finally, I assign a *suspiciousness score* to every line of code, according to a fault localization formula. In the implementation of WHOSEFAULT that I used for my experiments, I used TARANTULA’s fault-localization formula, proposed by Jones *et al.* [50]. However, other fault-localization techniques can be utilized in my approach to assess the suspiciousness of different code locations to contain the bug.

5.3.2 Obtain History of Lines of Code of Interest

Given the fuzzy history-slicing criterion automatically selected in the previous step, WHOSEFAULT automatically obtains its corresponding history slice by using my automatic history slicing technique that I described in Section 4.2.

5.3.3 Examine History of Lines of Code of Interest

Once the fuzzy history-slicing criterion and its corresponding history slice have been automatically computed, WHOSEFAULT also automatically analyzes them to provide a recommendation of the most suitable developers to fix the bug. For this goal, WHOSEFAULT performs two steps. First, it models the expertise offered by developers according to the changes that they performed to the code in its history. Then, it assesses the expertise that developers have to fix the bug by assessing the expertise that they provide for the specific lines of code that were exercised by it. I describe these steps in the next subsections.
Modeling the Expertise Offered by Developers

First, WhoseFault estimates the relative expertise that developers offer for each line of code by analyzing the history slice obtained in the previous step. I presented this step in Section 5.1.2 and depicted it in Figure 5.2. I postulate that developers increase their expertise in each line of code as they make more changes to it, and that developer expertise in a line of code decreases with time. Similar ideas have been proposed by other researchers, e.g., Fritz et al. proposed that a developer’s expertise for a given software artifact decreases as other developers make changes to that artifact [34].

For each developer, WhoseFault creates the model of expertise offered as a set of all the lines of code of the program, each of which is annotated with an expertise score that represents the developer’s relative expertise in that line of code. WhoseFault estimates the relative expertise of a developer for a line of code by applying Formula 5.1 to all lines of code and developers. WhoseFault measures the recency of each change by applying Formula 5.2. Formula 5.2 assigns a score to each change made by each developer that ranges from 0 to 1, according to how recently they occurred — 0 for changes made on the project’s start date and 1 for the last change made to the program. This way, WhoseFault assigns a higher expertise score to changes that happened more recently. Then, Formula 5.1 sums up the recency scores obtained for each change that was performed to a given line of code by a given developer. This way, WhoseFault assigns a higher expertise score to developers that performed more changes.

\[
\text{expertise}_{d,l} = \sum_{c_{d,l}=1}^{N} \text{recency}_{c_{d,l}} \quad (5.1)
\]

\[
\text{recency}_{c_{d,l}} = \frac{\text{date}_{c_{d,l}} - \text{project start date}}{\text{today} - \text{project start date}} \quad (5.2)
\]

In Formula 5.1, \(\text{expertise}_{d,l}\) represents the expertise that developer \(d\) offers for line of code
Assessing Expertise

In its final step, **WhoseFault** assesses the relative expertise that developers offer for fixing a bug. I also presented this step in Section 5.1.2 and depicted it in Figure 5.2. In this step, **WhoseFault** compares the expertise offered by developers to the expertise required to fix the bug. The intuition behind this step is that the expertise offered by a developer is better suited for fixing a bug when it is offered for the code locations that contain the bug. More specifically, I consider the suitability of developers to fix a bug to be directly proportional to: the suspiciousness of the lines of code over which they performed changes, the number of changes that they performed over the lines of code, and the recency of such changes.

I adjust developer expertise to the expertise required to fix the bug by applying Formula 5.3. My goal with Formula 5.3 is to adjust the expertise offered by each developer for a given line of code with a multiplier that represents the extent to which that line of code is relevant for fixing the bug. Formula 5.3 also aggregates each developer’s expertise by summing up the adjusted expertise scores obtained by that developer for all lines of code. This approach returns a recommendation of developers to fix the bug as a ranked list of developers, sorted by their aggregated expertise score, from highest to lowest.

$$\text{expertise}_d = \sum_{l=1}^{N} (\text{expertise}_{d,l} \times \text{suspiciousness}_l)$$  \hspace{1cm} (5.3)

In Formula 5.3, expertise\textsubscript{d} represents the relative expertise that developer \textit{d} offers for fixing the bug. The expertise score assigned to a developer \textit{d} for fixing a bug is calculated as the sum of the expertise offered by developer \textit{d} for each line of code \textit{l}, multiplied by the suspiciousness of that line of code \textit{l}.
5.4 Evaluation of Chronos

I performed an experiment to evaluate the support that Chronos provides to developers for answering questions about the history of source code. In this experiment, I measured the efficiency and effectiveness with which developers answered multiple questions about source-code history when using Chronos and when using existing techniques for exploring the history of source code. The goal of this experiment is to evaluate to whether the efficiency savings provided by automatic history slicing and the further assistance that Chronos provides for the selection of the history-slicing criterion and the visualization of history slices translate into actual benefits for real developers seeking to answer real code-history questions. In this sense, this experiment complements the experiment that I performed in Section 4.3.

5.4.1 Experimental Design

To evaluate the support provided by Chronos for developers to answer questions about code history, I performed a user study. In this user study, I asked 24 software developers to answer 3 common questions about code history by using both Chronos and a state-of-the-practice tool to explore the history of source code. I measured how efficient and effective developers were when answering these questions by using each of the two studied tools and, finally, I compared their results.

The experiment was structured as such for each human subject:

1. I trained the subject on how to use the chosen treatment technique.
2. I presented the subject with the randomly chosen question and the source code about which it is asked.
3. I asked the subject to answer the chosen question and measured his/her effectiveness
and efficiency in the question.

4. I repeated Steps 1–3 for the other treatment technique and another randomly chosen question.

### 5.4.2 Experimental Subjects

I recruited 24 human subjects for this user study. To encourage subject participation, I offered a small base monetary compensation. I required subjects to have knowledge about both Java and revision-control systems. To ensure such knowledge, I asked subjects to fill in an online questionnaire in order to sign up for the study. I used the information in this questionnaire to screen subjects according to the requirements and to capture the demographics of the population. 22 subjects were male and 2 were female. 23 subjects fell in the age group 18–29 and 1 fell in the age group 30–39. All the subjects were graduate students at UCI, two of them were also professional software developers and one was also a professional software tester. 13 subjects were majoring in Informatics, 10 were majoring in Computer Science and 1 was majoring in Computer Engineering. Table 5.1 contains the years of experience that subjects reported to have with IDEs and revision control systems.

I trained subjects for both tools for 10 minutes. For the existing state-of-the-practice system, I trained developers on how to: (1) request all the revisions of a file, (2) open the contents of a revision of a file, (3) compare two revisions of a file, and (4) request an annotation/blame for a revision of a file. For CHRONOS, I trained developers on how to: (1) select the lines of Table 5.1: Number of subjects with $n$ years of experience with IDEs and revision-control systems.

<table>
<thead>
<tr>
<th>Tool</th>
<th>&lt;1</th>
<th>1–2</th>
<th>3–4</th>
<th>5–6</th>
<th>6–7</th>
<th>8–9</th>
<th>&gt;10</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDEs</td>
<td>0</td>
<td>6</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Revision control</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
code of interest, (2) request the computation and visualization of their corresponding history slice, and (3) navigate through the visualization of the history slice. I also allowed subjects to ask questions about the material covered in the training and about keyboard shortcuts while they were working with the tool.

I asked subjects questions about the history of code selections from the AspectJ open-source project [1]. AspectJ is an aspect-oriented extension to the Java programming language that consists of over 510,000 lines of code and has been in active development for more than 10 years.

5.4.3 Independent Variables

I experimented using two independent variables: treatment technique and developer question. Each variable will be defined and motivated in turn.

Independent Variable 1: Technique Treatment

In order to evaluate the support provided by CHRONOS for developers for answering questions about code history over existing techniques, I studied two technique treatments:

Conventionally Assisted. I selected this technique as the state-of-the-practice technique. For the conventionally assisted approach, I used Eclipse’s default CVS plug-in, which is a widely used graphical user interface for CVS, an industry-standard revision-control system. The plug-in enables an easy and automated use of the annotate feature to navigate the dimension of history: the user can select a file, and through a context menu, choose to see the last revision in which each of its lines was changed. It also provides implementations of the Ctrl+F and diff features to navigate the dimension of the program.
**Chronos.** Chronos, implemented as an Eclipse plug-in, along with its visualization (presented in Section 5.2). Chronos visualizes history slices with context, given that it expects them to be consumed by humans, as opposed to history slices without context, which I envision being more often used as input for other analysis techniques.

**Independent Variable 2: Developer Question**

I varied the question about code history that subjects were asked:

**Authorship.** Subjects were asked to identify the complete set of developers who had ever contributed changes to a segment of code. This question reflects the real task demand of determining authorship and expertise (discussed in Section 5.1.1).

**Original Revision.** Subjects were asked to identify the original revisions in which a segment of code was originally created. This question reflects the real task demand of determining an earlier implementation of a given functionality (discussed in Section 5.1.1).

**Co-evolution.** Subjects were asked to identify the revisions in which two segments of code in two different files were changed within a day of each other. This question reflects the real task demand of determining “evolutionary coupling” for identifying related code or code-clone risk (discussed in Section 5.1.1).

**Controlled Variables**

To control for the effects of outside influences, I vary the following three controlled variables: user subject skill, technique order, and technique-question combination.

**Subject Skill.** I studied 24 human subjects of varying levels of experience with programming in integrated development environments (IDEs) and revision-control systems. The number of years of experience with each of these is presented in Table 5.1.
Technique Order. I varied the order in which the subjects used each technique treatment. Half of the subjects used the techniques in one order, and the other half used them in the opposite order.

Technique-Question Combination. Every combination of treatment technique and question was performed in equal numbers. Each human subject was asked to answer 2 questions with 2 different techniques. Each of the three questions was assigned to an equal number (16) of subjects, and each technique was used for each question in equal numbers — i.e., each question was answered with each tool 8 times.

5.4.4 Dependent Variables

To assess the benefits of CHRONOS over existing techniques, I measure two dependent variables: correct answer and time to answer question.

Correct Answer. Correctness of the answer given by the subject. I precomputed the correct answer for each task in advance, and was able to determine whether the answer supplied by the subject was correct. In order to encourage correctness, I offered a small additional compensation for providing a correct answer, on top of the base compensation that subjects obtained simply by performing the study.

Time to Answer Question. Time that the subject took to answer the question. The time is determined by when the subject decided that she was confident and satisfied with her result. In order to encourage speed, the earlier that the answer was given determined the size of the small additional compensation offered for correct answers. For example, a correct answer given at minute 5 was rewarded more greatly than a correct answer given at minute 10. The subjects were given a maximum of 10 minutes to answer each task that they were given, regardless of the treatment technique. This compensation
structure was implemented equally for both techniques to encourage the subjects to answer quickly and accurately.

5.4.5 Results

For each technique-question combination, I averaged the time that all subjects took to answer each question, and I calculated the percentage of subjects that answered each question correctly. Table 5.2 contains the results that I obtained for all technique-question combinations.

In the results of this experiment, I observed that CHRONOS allowed developers to answer three common questions about code history in around half the time and correctly in more than triple the cases than when using a state-of-the-practice technique to explore code history for all the studied questions.

The subjects who used CHRONOS could answer any of the questions in around half the time that was needed by the subjects who used the Conventionally Assisted technique. I performed a paired t-test with 7 degrees of freedom over the timings that I measured for subjects answering the questions. The difference in time needed to answer the questions for the two different techniques was statistically significant, with p-values of 0.02, 0.01, and

Table 5.2: Time to answer question and correct answer rate for each technique and question.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Question</th>
<th>Mean time</th>
<th>% Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventionally Assisted</td>
<td>Authorship</td>
<td>6:04</td>
<td>37.50%</td>
</tr>
<tr>
<td>CHRONOS</td>
<td>Authorship</td>
<td>3:21</td>
<td>100.00%</td>
</tr>
<tr>
<td>Conventionally Assisted</td>
<td>Original Revision</td>
<td>7:34</td>
<td>37.50%</td>
</tr>
<tr>
<td>CHRONOS</td>
<td>Original Revision</td>
<td>3:15</td>
<td>100.00%</td>
</tr>
<tr>
<td>Conventionally Assisted</td>
<td>Co-evolution</td>
<td>9:57</td>
<td>0.00%</td>
</tr>
<tr>
<td>CHRONOS</td>
<td>Co-evolution</td>
<td>5:19</td>
<td>62.50%</td>
</tr>
</tbody>
</table>
0.00006 for the authorship, original revision, and co-evolution questions, respectively.

In the specific case of the co-evolution question, the differences in time are so high because most subjects using the Conventionally Assisted technique were unable to answer this question in the maximum time allotted (10 minutes). Only two subjects provided an incomplete, incorrect answer when they had 15 seconds and 5 seconds left, respectively. In a post-experiment discussion, some of the subjects mentioned that, in real life, they would have just given up on answering the co-evolution question before spending ten minutes on it by using the Conventionally Assisted technique.

Regarding the answer-correctness rate, the subjects who used CHRONOS were in general 62.5% more likely to provide correct solutions to any of the questions than those subjects who used the Conventionally Assisted technique. I also performed a paired t-test with 7 degrees of freedom over the correctness of the answers provided by the subjects. The difference in correctness of the answers for the two different techniques was also statistically significant, with a p-value of 0.01 for each question.

I observed multiple reasons why the subjects did not provide correct answers when using the Conventionally Assisted technique. For the authorship question, some subjects only reported information about the last change to each line of interest, instead of their whole history; some other subjects only reported information about the header of the method of interest, instead of all its lines; and another subject checked all revisions of the file one by one, manually checking which revisions affected the lines of interest, and ending up providing a partial answer. For the original revision question, some subjects interpreted changes in lines as additions; and other subjects ran out of time. For the co-evolution question, all subjects ran out of time long before even composing a small portion of the correct answer. I also observed a common phenomenon to all tasks when using the Conventionally Assisted technique: subjects very often lost track of where in the file they were, and they needed to backstep to remind and re-focus their search.
When using CHRONOS, all the subjects provided the correct answer for the authorship and co-evolution questions, and a majority of the subjects provided correct answers to the co-evolution question (despite zero success for the other technique). For the 37.5% subjects that incorrectly answered the co-evolution question with CHRONOS, I observed the following reasons for their failure: One subject did not explore the beginning of the timeline, missing the first revision. Another subject rushed and explored only the latter half of the timeline. Another subject only explored a small subset of the timeline because she thought that her partial view was complete — the zooming function did not refresh the image until the mouse button was released, causing her to think that her zoomed-in view contained the whole timeline.

5.4.6 Threats to Validity

An external threat to validity is that this experiment studied people who were not already familiar with the code base, and thus, its results may not generalize to more experienced developers. However, I believe this factor to be insignificant because the same experience level was brought to both technique treatments. I expect that greater experience would assist the more automated approach because its results require more interpretation and application of experience, whereas the manual approach requires more tedium. Nevertheless, in the future, I plan to evaluate with in-vivo field studies.

5.5 Evaluation of WhoseFault

In this section, I evaluate my hypothesis that the multi-revision, fine-grained analysis of source-code history can be automated to support developers in answering a prevalent developer question: who are the most suitable developers to fix a bug? To that goal, I created
WhoseFault, an automatic technique that analyzes multi-revision, fine-grained source code history to provide an automatic recommendation of the most suitable developers to fix a bug. In order to evaluate the support that WhoseFault provides for answering this question, I conducted an experiment that compare the recommendations provided by WhoseFault to those provided by eight prominent bug-assignment techniques.

Unlike existing bug-assignment techniques, WhoseFault allows developers to perform automatic bug assignment in situations when a bug report is not available. The goal of this evaluation is to study whether WhoseFault can provide developer-to-bug recommendations with similar effectiveness and efficiency as when a bug report is available. In the following subsections, I describe in more detail the design and results of my experiment.

5.5.1 Experimental Design

In this evaluation, I implemented may execution-based bug-assignment approach in my technique WhoseFault and 3 state-of-the-art natural-language-based bug-assignment techniques. I evaluated both kinds of techniques over 107 real bugs from 3 real-word software projects. For each studied bug, I identified its corresponding execution and bug report, as well as the ground truth of which developer was most suitable to fix it. I used the bug execution to obtain a developer recommendation with WhoseFault, and I used the bug report to obtain a developer recommendation with the natural-language-based techniques. Finally, I measured the effectiveness and efficiency of the developer recommendation provided by each technique.
5.5.2 Experimental Subjects

I performed my experiment over three different software projects of varying sizes. The first subject is Apache Commons IO [6], which is a library of utilities for the development of input and output functionality. Apache Commons IO contains 26,000 lines of code. The second subject is Mozilla Rhino [69], which is an open-source JavaScript compiler and interpreter. Mozilla Rhino contains 185,000 lines of code. The final subject is AspectJ [1], which is an aspect-oriented development framework for Java. AspectJ is composed of around 510,000 lines of code. Table 5.3 contains statistics about the studied subjects and the bugs that I studied from them. Since I studied bugs at different points in time, the size of the development team varied for each bug.

I studied a total of 107 real bugs from the subject programs: 72 bugs for AspectJ, 15 bugs for Mozilla Rhino, and 20 bugs for Apache Commons IO. For each subject, I studied all the bugs from the subject’s history for which I could obtain: (1) a bug report that represents the bug — to evaluate natural-language-based techniques, (2) an execution of the bug — to evaluate WhoseFault, and (3) the ground-truth for the most suitable developer to fix the bug. In other words, the studied bugs were complete and comprehensive for those that were feasible for the study. I describe the process that I followed to obtain a bug report and an execution of a bug in Section 5.5.3. I also describe how I established the most suitable developer to fix a bug in Section 5.5.4.

5.5.3 Subject Preparation

Since I evaluated many techniques with different resource requirements, this experiment involved an intensive process of subject preparation. In order to perform the experiment, I had to obtain from all the subject programs the resources that each kind of bug-assignment technique requires: a bug report that describes the bug, an execution of the bug, the history
Table 5.3: Empirical subjects

<table>
<thead>
<tr>
<th></th>
<th>Apache Commons IO</th>
<th>Mozilla Rhino</th>
<th>AspectJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>26,000</td>
<td>185,000</td>
<td>510,000</td>
</tr>
<tr>
<td>Age (years)</td>
<td>11</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>Median Dev. Team Size</td>
<td>20</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>Number of Bugs</td>
<td>20</td>
<td>15</td>
<td>72</td>
</tr>
</tbody>
</table>

of the bug reports of the project, and the history of the source code of the project. I obtained these resources from the official repositories of the studied subject programs and from the iBugs data set [23].

First, I obtained a set of bug reports that natural-language-based techniques could use to model the expertise required to fix the bug or to model the expertise offered by developers. To achieve this, I parsed the complete bug-reporting system of all the subjects and extracted the bug descriptions inside the bug reports that had already been fixed.

Second, I obtained a set of bug executions that execution-based techniques could use to model the expertise required to fix the bug. In order to obtain this set of bug executions, I identified test cases that failed as a consequence of a bug. To do that, I had to identify the revisions in the history of the source code that fixed a bug and to check out the revisions before (pre-fix) and after (post-fix) the fix was applied. Once I identified the pre-fix and post-fix revisions that corresponded to every fixed bug, I had to compile such revisions, which often required manual assistance. After compiling all the pre-fix and post-fix revisions, I had to build scripts to enumerate the test cases in the test suite and automate their individual execution. I also had to identify all the test cases that existed only in the post-fix revision and copy them to the pre-fix revision, since such test cases were more likely to fail in the pre-fix revision as a consequence of the fixed bug. Then, I instrumented every pre-fix revision,
individually executed every test case in the test suite both in the pre-fix revision and the post-fix revision, and captured the outcome of every test case. At this point, I could compare the output of test cases and identify those that failed in the pre-fix revision and passed in the post-fix revision. From that set of test cases, I selected the ones that failed as a consequence of the fixed bug — I selected the test cases that had executed at least one of the lines of code that were fixed in the post-fix revision.

Third, I obtained the complete source-code history of the subject programs so that both natural-language-based techniques and my execution-based technique could model the expertise offered by developers. To this goal, I mined the source-code repository of the subject programs. For every change that was ever performed over the source code, I extracted its meta-data as well as the contents of the lines of code that it changed. I used this information to build a history graph for every subject program, by following the steps described in Section 4.2.1.

5.5.4 Ground Truth

I identified the ground truth of who was the most suitable developer to fix a bug for all the studied bugs. The common way to assess ground truth in the bug-assignment field is that the most suitable developer to fix a bug is the person who actually fixed it. The person who fixed a bug can be obtained from the bug-tracking system or from the source-code repository. In this experiment, I studied bugs that could be labeled both with a developer who fixed them according to the bug-tracking system and with a developer who fixed them according to the source code repository — whether the two repositories pointed to the same developer or not.
5.5.5 Independent Variable: Bug-Assignment Technique

I used one independent variable in this experiment: the bug-assignment technique. In total, I studied nine bug-assignment techniques: eight representative state-of-the-art natural-language-based techniques, and my execution-based technique — WHOSefault. Below, I include the nine values of the independent variable:

**NaïveBayes**, proposed by Cubranic et al. [22]. This technique assesses expertise by measuring the similarity between the vocabulary that developers used to describe the bug and the vocabulary used to describe the bugs that they fixed in the past. To assess such similarity, this technique uses a Naïve Bayes classifier without normalization of word frequencies.

**Sibyl-NB**, proposed by Anvik et al. [4]. This technique assesses expertise by measuring the similarity between the vocabulary that developers used to describe the bug and the vocabulary used to describe the bugs that they fixed in the past. To assess such similarity, this technique uses a Naïve Bayes classifier.

**Sibyl-SVM**, proposed by Anvik et al. [4]. This technique assesses expertise by measuring the similarity between the vocabulary that developers used to describe the bug and the vocabulary used to describe the bugs that they fixed in the past. To assess such similarity, this technique uses a Support Vector Machines classifier.

**Sibyl-C45**, proposed by Anvik et al. [4]. This technique assesses expertise by measuring the similarity between the vocabulary that developers used to describe the bug and the vocabulary used to describe the bugs that they fixed in the past. To assess such similarity, this technique uses a C4.5 decision-tree classifier.

**Sibyl-EM**, proposed by Anvik et al. [4]. This technique assesses expertise by measuring the similarity between the vocabulary that developers used to describe the bug and the vocabulary used to describe the bugs that they fixed in the past. To assess such similarity, this technique uses an Expectation Maximization classifier.
**Sibyl-CR**, proposed by Anvik *et al.* [4]. This technique assesses expertise by measuring the similarity between the vocabulary that developers used to describe the bug and the vocabulary used to describe the bugs that they fixed in the past. To assess such similarity, this technique uses a Conjunctive Rules classifier.

**Sibyl-NN**, proposed by Anvik *et al.* [4]. This technique assesses expertise by measuring the similarity between the vocabulary that developers used to describe the bug and the vocabulary used to describe the bugs that they fixed in the past. To assess such similarity, this technique uses a Nearest Neighbor classifier.

**Develect**, proposed by Matter *et al.* [65]. This technique assesses expertise by measuring the similarity between the vocabulary that developers used to describe the bug and the vocabulary that they used to write the source code. To assess such similarity, this technique uses a cosine-distance-similarity metric.

**WhoseFault**, my novel bug-assignment technique that I described in Section 5.3. My technique assesses expertise by scoring developers according to the suspiciousness level of the lines of code that they modified, the number of changes that they performed over such lines of code, and the recency of such changes.

### 5.5.6 Dependent Variables

I used two dependent variables in my experiment: the effectiveness and the efficiency with which each bug-assignment technique produced a developer recommendation for a bug.

**Effectiveness.** I measure the effectiveness with which a bug-assignment technique provided a developer recommendation for a bug by using the Normalized Discounted Cumulative Gain (NDCG) metric [48]. NDCG scores range from 0 — for the worst recommendation possible — to 1 — for the best recommendation possible. All the techniques that I studied output their recommendation in the format of a ranked list of developers, from most to least suitable to fix the bug. NDCG is commonly used in information retrieval
to assess the quality of ranked recommendations. In this experiment, I used the NDCG formula proposed by Burges et al. [15], as in Equation 5.4.

\[
\text{NDCG}_p = \frac{\text{DCG}_p(\text{recommendation})}{\text{DCG}_p(\text{ideal recommendation})} \quad (5.4)
\]

\[
\text{DCG}_p = \sum_{i=1}^{p} \frac{2^{\text{rel}_i} - 1}{\log_2(i + 1)} \quad (5.5)
\]

NDCG is calculated as the Discounted Cumulative Gain (DCG) score for a recommendation, divided by the DCG score of an ideal recommendation — a recommendation in which the correct answers are contained in its top positions. In the DCG formula (Equation 6.7), \(i\) represents the top \(i^{th}\) position inside a recommendation, and \(\text{rel}_i\) represents the actual, ground-truth relevance of the item recommended in position \(i\).

I attributed a relevance of 1 to every developer that I considered as having actually fixed the bug according to my collected ground truth (see Section 5.5.4). The \(p\) suffix in Equations 6.6 and 6.7 determine the top \(p\) positions that are considered inside a recommendation. To account for recommendations as large as five developers, I used the NDCG\(_5\) score in this experiment.

**Efficiency.** I measured the efficiency of a bug-assignment technique for providing a developer recommendation for a bug by measuring its execution time. For each one of the studied bug-assignment techniques, I measured the amount of time that it took to provide a developer recommendation. More specifically, I measured the time that each technique took to provide a developer recommendation once its pre-requisites had already been pre-computed. The requirement-preparation process (see Section 5.5.3) can be executed once and its output can then be incrementally updated. Since the requirement-preparation process does not need to be re-executed for every developer recommendation, I did not account for its execution measuring the efficiency of a technique.
For NaïveBayes and Sibyl, I measured the time that they took to tokenize the words inside bug reports, normalize term frequencies, train the machine-learning classifier, and return a prediction from the classifier. For Develect, I measured the time that it took to tokenize the terms inside the history of the source code, normalize term frequencies, create the term-author matrix, create the term vector for the bug report and calculate the expertise scores. For WhoseFault, I measured the time that it took to localize the bug and to apply its expertise formula. I executed all the evaluated bug-assignment techniques in a machine with an Intel i7 950 CPU at 3.07GHz, 12 gigabytes of memory, and Ubuntu 12.10 64 bit.

5.5.7 Results

I present in this section the results of my evaluation. I report here the effectiveness and efficiency of the developer recommendations that WhoseFault provided from bug executions in comparison with the developer recommendations that natural-language-based techniques provided from bug reports.

Effectiveness

I report the effectiveness provided by each bug-assignment technique in Table 5.4. Table 5.4 shows the median value, mean value, and standard deviation of the NDCG₅ scores provided by each technique for each subject. A higher NDCG₅ score means higher effectiveness. I analyze the effectiveness provided by both kinds of bug-assignment techniques from two perspectives. I first study how WhoseFault compares to the most effective natural-language-based technique for each specific subject. Since I observed that different natural-language-based techniques were most effective for different subjects, I also study how WhoseFault compares to the natural-language-based techniques across all subjects.
Table 5.4: Distribution of NDCG<sub>5</sub> scores obtained by all techniques for all subject programs

<table>
<thead>
<tr>
<th>Technique</th>
<th>Apache Commons IO</th>
<th>Mozilla Rhino</th>
<th>AspectJ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \bar{x} )</td>
<td>( \bar{x} )</td>
<td>( \sigma_x )</td>
</tr>
<tr>
<td>NaïveBayes</td>
<td>0.82</td>
<td>0.75</td>
<td>0.30</td>
</tr>
<tr>
<td>Sibyl-C45</td>
<td>0.63</td>
<td>0.52</td>
<td>0.46</td>
</tr>
<tr>
<td>Sibyl-CR</td>
<td>0.63</td>
<td>0.71</td>
<td>0.28</td>
</tr>
<tr>
<td>Sibyl-EM</td>
<td>0.00</td>
<td>0.35</td>
<td>0.49</td>
</tr>
<tr>
<td>Sibyl-NB</td>
<td>0.82</td>
<td>0.75</td>
<td>0.29</td>
</tr>
<tr>
<td>Sibyl-NN</td>
<td>0.50</td>
<td>0.68</td>
<td>0.29</td>
</tr>
<tr>
<td>Sibyl-SVM</td>
<td>0.82</td>
<td>0.65</td>
<td>0.42</td>
</tr>
<tr>
<td>Develect</td>
<td>1.00</td>
<td>0.79</td>
<td>0.30</td>
</tr>
</tbody>
</table>

**WhoseFault**** | 1.00 | 0.77 | 0.30 | 1.00 | 0.83 | 0.19 | 0.63 | 0.62 | 0.36 |

Figure 5.7: Distribution of NDCG<sub>5</sub> scores obtained by each technique for each separate subject program. Notice that WhoseFault (the right-most box plot for each subject) not only provided as much median effectiveness as natural-language-based techniques, but it was actually more effective than most of the studied techniques for each subject.

**Effectiveness per Separate Subject Program.** In order to compare the effectiveness provided by WhoseFault with the effectiveness provided by natural-language-based techniques for each subject, I visually represent Table 5.4 into Figure 5.7. Figure 5.7 contains three groups of box plots in its horizontal axis, one per subject program. For each subject program, Figure 5.7 contains nine individual box plots, one for each technique. The box plot corresponding to each technique and subject represents the distribution of NDCG<sub>5</sub> scores obtained by that technique for all the bugs for that subject.

These results show that not only WhoseFault provided as much effectiveness
as natural-language-based techniques, but it was actually more effective than most of the studied techniques for each subject. In fact, WhoseFault provided very close effectiveness to the most effective natural-language-based technique for each subject. WhoseFault obtained the same median effectiveness as the natural-language-based technique with the highest median effectiveness for each subject. WhoseFault also obtained a roughly equivalent mean effectiveness to the natural-language-based technique with the highest mean effectiveness for Apache Commons IO and Mozilla Rhino, and only moderately lower mean effectiveness than the natural-language-based technique with the highest mean effectiveness for AspectJ.

In terms of median effectiveness, the natural-language-based techniques that obtained the highest median NDCG$_5$ score were Develect for Apache Commons IO; NaïveBayes, Sibyl-C45, Sibyl-NB, and Sibyl-SVM for Mozilla Rhino; and Sibyl-CR and Sibyl-SVM for AspectJ. For every one of the three subjects, WhoseFault obtained the same median NDCG$_5$ score as the natural-language-based technique with the highest median NDCG$_5$ score. For Apache Commons IO and Mozilla Rhino, WhoseFault obtained a median NDCG$_5$ score of 1.0, which means that, in the median case, WhoseFault recommended the most suitable developer to fix the bug in the first position of its developer recommendation. For AspectJ, WhoseFault obtained a median NDCG$_5$ score of 0.63, which means that, in the median case, WhoseFault recommended the most suitable developer to fix the bug in the second position of its developer recommendation.

In terms of mean effectiveness, WhoseFault obtained a roughly equivalent mean NDCG$_5$ score to the natural-language-based technique with the highest mean NDCG$_5$ score for Apache Commons IO and Mozilla Rhino, and moderately lower mean NDCG$_5$ score for AspectJ. In Apache Commons IO, the difference between the mean NDCG$_5$ score obtained by WhoseFault and Develect was 0.02. This difference is equivalent to WhoseFault recommending the most suitable developer in the second position when Develect
Figure 5.8: Distribution of NDCG$_5$ scores obtained by each technique across all subject programs. Notice that, for each natural-language-based technique, WhoseFault (the rightmost boxplot cluster) provided at least equal effectiveness for any subject, and it provided higher effectiveness for at least one subject.

recommended her in the first position for only one of the bugs studied for Apache Commons IO. In Mozilla Rhino, the difference between the mean NDCG$_5$ score obtained by WhoseFault and NaïveBayes was 0.03. This difference is equivalent to WhoseFault recommending the most suitable developer in the third position when NaïveBayes recommended her in the first position for only one of the bugs studied for Mozilla Rhino. In AspectJ, the difference between the mean NDCG$_5$ score obtained by WhoseFault and Sibyl-CR and Sibyl-SVM was 0.1. This difference is equivalent to WhoseFault recommending the most suitable developer in the third position when Sibyl-CR and Sibyl-SVM recommended her in the first position for 20% of the bugs studied for AspectJ.

**Effectiveness Across Subject Programs.** While studying the effectiveness that bug-assignment techniques provided for each separate subject, I also observed that all the techniques provided different effectiveness for different subjects. Therefore, I study in this subsection the overall effectiveness that WhoseFault provided across all subjects in comparison with natural-language-based techniques. This time, I visually represent Table 5.4 in a different layout into Figure 5.8. Figure 5.8 shows nine sections in its horizontal axis, one for each studied technique. Within each technique, Figure 5.8 contains three individual box plots that display the distribution of NDCG$_5$ scores obtained by that technique on each one of the three subject programs.
Figure 5.9: Time (seconds) spent by each technique to provide a recommendation for a bug on each subject. Notice that WHOSÉFAULT (the right-most box plot for each subject) took longer time to execute than most natural-language-based techniques for all subjects. Nevertheless, WHOSÉFAULT took a median and mean execution time under one minute.

These results show that, for every natural-language-based technique, WHOSÉFAULT provided at least equal effectiveness for any subject, and it provided higher effectiveness for at least one subject. WHOSÉFAULT provided higher median effectiveness than any natural-language-based technique for at least one subject and higher mean effectiveness for at least two subjects. There were only four cases in which a natural-language-based technique provided slightly or moderately higher mean effectiveness than WHOSÉFAULT for only one subject: SIBYL-NB for MOZILLA RHINO, DEVELECT for APACHE COMMONS IO, and SIBYL-CR and SIBYL-SVM for ASPECTJ.

**Efficiency**

In this subsection, I study whether WHOSÉFAULT can provide developer recommendations for a bug as efficiently as natural-language-based techniques. I report in Table 5.5 the median, mean, standard deviation, minimum and maximum values of the time (in seconds) that each technique took to provide a developer recommendation for a bug on each subject. I also visually represent Table 5.5 in Figure 5.9. Figure 5.9 contains three groups of box plots in the horizontal axis, one for each subject program. Within each subject, Figure 5.9 displays an individual box plot for each studied technique. The box plot for each technique and
Table 5.5: Time (seconds) spent by each technique to provide a recommendation for a bug on each subject

<table>
<thead>
<tr>
<th>Technique</th>
<th>Apache Commons IO</th>
<th>Mozilla Rhino</th>
<th>AspectJ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\bar{x}$</td>
<td>$\bar{x}$</td>
<td>$\sigma_x$</td>
</tr>
<tr>
<td>NaïveBayes</td>
<td>0.06</td>
<td>0.07</td>
<td>0.02</td>
</tr>
<tr>
<td>Sibyl-C45</td>
<td>0.18</td>
<td>0.21</td>
<td>0.17</td>
</tr>
<tr>
<td>Sibyl-CR</td>
<td>0.06</td>
<td>0.06</td>
<td>0.03</td>
</tr>
<tr>
<td>Sibyl-EM</td>
<td>5.21</td>
<td>7.31</td>
<td>8.32</td>
</tr>
<tr>
<td>Sibyl-NB</td>
<td>0.06</td>
<td>0.06</td>
<td>0.02</td>
</tr>
<tr>
<td>Sibyl-NN</td>
<td>0.05</td>
<td>0.04</td>
<td>0.01</td>
</tr>
<tr>
<td>Sibyl-SVM</td>
<td>0.07</td>
<td>0.06</td>
<td>0.01</td>
</tr>
<tr>
<td>Develect</td>
<td>20.19</td>
<td>20.93</td>
<td>6.79</td>
</tr>
<tr>
<td>WhoseFault</td>
<td>1.75</td>
<td>1.98</td>
<td>0.63</td>
</tr>
<tr>
<td>WhoseFault</td>
<td>16.77</td>
<td>17.66</td>
<td>13.78</td>
</tr>
</tbody>
</table>
subject represents the distribution of times (in seconds) that the technique took to provide a recommendation for the studied bugs for that subject. Lower values in the vertical axis mean higher efficiency. I limited the vertical axis to 60 seconds in order to display execution times within a range that represents practical use.

These results show that WhoseFault took longer time to execute than most natural-language-based techniques for all subjects, providing less efficiency. Nevertheless, WhoseFault took substantially less time to execute than the remaining techniques. In general for all subjects, WhoseFault took a median and mean execution time of only one minute, and a maximum execution time of only two minutes for all subjects.

For Apache Commons IO, most natural-language-based techniques took a median execution time under 1 second, except for Sibyl-EM and Develect, which took a median of 5 seconds and 20 seconds, respectively. WhoseFault took a median execution time under 2 seconds, which is slightly longer than most natural-language-based techniques, but much shorter than Sibyl-EM and Develect. For Mozilla Rhino, most natural-language-based techniques took a median execution time under 2 seconds, again except for Sibyl-EM and Develect, which took a median of 85 seconds and 26 seconds, respectively. WhoseFault took a median execution time of 52 seconds, which is much longer than most natural-language-based techniques, but shorter than Sibyl-EM. For AspectJ, most natural-language-based techniques took a median execution time under 5 seconds, except for Sibyl-C45, Sibyl-EM and Develect, which took a median of 142 seconds, 2,317 seconds and 711 seconds, respectively. WhoseFault took a median execution time of 17 seconds, which is slightly longer than most natural-language-based techniques, but drastically shorter than Sibyl-C45, Sibyl-EM and Develect.
5.5.8 Threats to Validity

In this section, I mention several threats to validity for this experiment. A threat to internal validity is the possibility that I did not consider some factors that could have impacted the dependent variables without my knowledge. As such, it is possible that mistakes in the implementation could have affected the results of the evaluation. However, I diligently tested the implementation of the studied techniques and checked their results at each step of the process. Additionally, I exercised high care in the preparation of the experimental data to avoid biases caused by its characteristics, as described in Section 5.5.3.

Threats to external validity include the possibility of the results of the experiments to not generalize to other subjects or other conditions. I used three real-world software programs to perform the experiments, all of which had been developed for at least 10 years and which also represented three different program sizes. As such, I believe that my execution-based bug-assignment technique can provide competitive developer-to-bug recommendations for a variety of software projects.

A threat to construct validity is the possibility that the metrics chosen by the experimenters do not accurately capture the concepts that they intended to evaluate. I measured effectiveness and efficiency with metrics that are commonly used for such concepts: the NDCG metric is commonly used to evaluate recommendation systems, and execution time is a common metric to measure efficiency. I defined the most suitable developer to fix a bug as the person who actually ended up fixing it. It is possible that in some cases the person who fixed the bug was chosen because it was the person with lower workload, or because that person wanted to learn about the functionality involving the bug. Nonetheless, the actual person chosen by the people who know the most about the code-base (the actual development team) is likely to be among the most experienced, most of the time. I also increased the correctness with which I identified the ground-truth of the person who fixed a bug by using two different
sources of information: the bug report, and the source-code repository.
Chapter 6

The Multi-revision, Fine-grained Analysis of Source-code History Can Be Partially Automated in a Way That Is Accurate

In Chapters 4 and 5, I presented techniques to partially automate the process of multi-revision, fine-grained analysis of source-code history in a way that improves the efficiency of current approaches and that provides support for answering developer questions.

In this chapter, I investigate the accuracy with which existing models of multi-revision, fine-grained code history may represent actual code evolution. Existing fine-grained code-history models are limited by the fact that they track code changes between contiguous revisions of code absolutely and unambiguously. However, in my personal experience with human-assessed, manual line mappings between consecutive code revisions, are often not binary, i.e., mapped or not. Instead, there are varying degrees to which lines should be mapped.
Moreover, there is sometimes uncertainty by individuals performing manual mapping, or dispute between multiple individuals as to the correct mapping.

The ground truth of line-of-code evolution, as expressed by individuals, has a fundamentally fuzzy nature. This fuzzy nature of code evolution inspired me to propose a fuzzy approach to modeling and analyzing code history. This fuzzy approach follows the intuition that, in truth, lines of code change into other lines to varying degrees. I present the fuzzy history graph as a model of fine-grained code history that represents the varying degrees to which individual lines of code evolve into others. Additionally, I present automatic fuzzy history slicing as an automatic technique that takes advantage of the fuzzy history graph to improve the accuracy of history slicing.

Finally, I evaluate the accuracy provided by automatic fuzzy history slicing when compared to existing approaches for modeling code evolution by performing two separate experiments. First, I evaluate the accuracy with which a fuzzy history graph represents code evolution in comparison with current models. Then, I evaluate the accuracy that automatic history slicing provides with fuzzy history graphs and with existing models. These two experiments complement each other, since they evaluate the accuracy with which a fuzzy history graph represents code evolution for individual code changes and with which automatic fuzzy history slicing represents code evolution for full code histories.

6.1 Accuracy Limitations of Current Approaches for History Slicing

The accuracy of a history slicing approach is strongly impacted by the accuracy with which it navigates the dimension of the program, i.e., the accuracy with which it can identify which lines of code evolved into which others between consecutive revisions. Additionally, the
accuracy of the navigation of the dimension of the program also impacts the accuracy of the navigation of the dimension of history, since the decision of whether to include a revision in the history slice depends on the decision of which lines belong to the history-slicing snapshot for that revision — i.e., whether such lines were modified.

Current SCM tools allow either a completely manual or conventionally assisted approach to navigating the dimension of the program. In the manual approach and the approach assisted by Ctrl+F, humans are highly likely to make mistakes judging the correspondence of lines between revisions — especially after many repetitions. In the other — more automated — conventionally assisted approaches, i.e., all-to-all and one-to-one, history slicing may present limitations in terms of precision and recall. As a result, different approaches to navigating the dimension of the program will return different history slices for a given history-slicing criterion.

In order to illustrate the navigation of the dimension of the program through multiple consecutive revisions (and for creating my automatic history slicing technique described in Chapter 4.2), I created a fine-grained model of the evolution of the code. I call such a fine-grained, multi-revision model of the code history, a history graph. The history graph maps the corresponding lines between each two consecutive revisions, across any epoch of the code history. Once the history graph is constructed (either a priori or on-demand), the history slice can be computed for any history-slicing criterion by traversing the graph (in either time direction).

Running Example. To illustrate the accuracy limitations of history slicing with current SCM systems, consider the source-code history included in Figure 6.1. This example program evolved through five revisions. Lines of code that changed between consecutive revisions are marked with connected rectangles. In this example, I consider a history-slicing criterion that contains only line 4 in revision 5.
6.1.1 Manual History Slicing

As I discussed in Section 4.1.1, in the manual approach to history slicing, developers manually decide which lines of code evolved into which others between consecutive revisions. When developers make this decision by manually inspecting the contents of source code files, or even when they are assisted by mechanisms like \textit{Ctrl+F}, they are highly likely to make mistakes, given the high amount of information that they need to process and the constant context-switching involved in the process. In fact, different people may assess the correspondence of lines of code between consecutive revisions differently — such manual assessments are subject to both human fallibility and differences of opinion. In truth, often such histories can be subject to debate, and no authoritative ground truth is possible. More importantly, manual history slicing can be highly time consuming.

\textbf{Example}. Figure 6.2(a) depicts a history graph for the program history shown in Figure 6.1. This history graph was created without any automation and was assessed by a person from one of my empirical studies (described later in Section 6.3). Each column represents a revision, and in each column the nodes represent lines of code. Between each adjacent pair of revisions, edges are drawn to represent evolution of lines of the incident nodes. Solid edges are drawn to represent changed and corresponding lines, and dotted
Figure 6.2: Manually-assessed history graph (a) and slice (b). On the slice (b), solid edges and solid nodes denote changes contributing to slicing criterion; dashed edges and hollow nodes denote corresponding but unchanged lines.

On this graph, the history-slicing criterion is defined as line 4 in revision 5. The computation of the history slice in depicted in Figure 6.2(b). In this figure, the analysis is performed as a backward (in time) analysis, and the edges are now depicted as directional to convey the traversal of the reachability analysis. Nodes that represent changed lines are depicted as a solid, red dot — these nodes constitute the resulting history slice. Nodes that did not change, but are on the trajectory of the slice, are depicted as open, red nodes. For this small example, the resulting human-assessed history slice was highly accurate (when compared with the lines of code that actually belong to the history of the selected line of code in Figure 6.1). However, as the number of revisions and lines involved in the history-slicing process increases to real-world sizes, it becomes increasingly hard (and time-consuming) for manual history slicing to provide high accuracy.
6.1.2 All-to-All History Slicing

Alternatively to the manual approach for navigating the dimension of the program, developers may use other, more automated approaches. Multiple tools and technique exist to automatically compute the correspondence of lines of code between two different files. Such techniques may be classified into all-to-all approaches and one-to-one approaches. A common example of an all-to-all approach is diff. For the example in Figure 6.1, diff computes the difference between revision 3 and revision 4 as:

```
1,8c1,7
< Array a = getFirstArray();
< Array b = getSecondArray();
<
< a = sortArray(a);         > Array firstArray = getFirstArray();
< b = sortArray(b);         > Array secondArray = getSecondArray();
< int m = 50; // max size   > // Get union with maximum size
< // Get the array union    > int maxSize = 50;
< Array u = getUnion(       > // Get the union of arrays
<     firstArray,secondArray,maxSize);    
< Array u = getUnion(a, b, m);
```

Such continuous blocks of changed code are called change hunks [67].

Using such diff results, researchers created models of history and analyses on them by encoding the potential for all lines in the prior revision to have changed into all lines in the subsequent revision for each change hunk. For example, Zimmermann et al. [92] proposed a model called annotation graphs. I generalize all such history models that map all lines of the prior revision to all lines of the later revision of a change hunk — I refer to these as All-to-All History Graphs.

The all-to-all history graph reduces the number of false negatives (i.e., not mapping lines of code that actually did evolve one into the other) by performing conservative mapping. As a consequence, such all-to-all models can be expected to provide high recall, although they
Figure 6.3: Existing all-to-all history graph and slice. The resulting history slice has high recall but low precision.

may also provide low precision. Moreover, the imprecision introduced by such conservative mapping can cause compounding imprecision when performing history slicing over multiple revisions.

Example. For my running example, Figure 6.3(a) depicts the history graph constructed using an all-to-all approach to map lines in the older revision to lines in the newer revision for each change hunk. Compared to the actual code evolution (Figure 6.1), in Figure 6.3(b) one can observe that the all-to-all analysis produced a result that wholly subsumes the correct lines in each revision. However, as expected, it includes many more lines in each revision, particularly for the oldest revision that contains all lines except one. Additionally, the number of revisions that would be recommended for examination is also an over-approximation. As a result, techniques that analyzed this resulting history slice may provide inaccurate results. For example, if the history slice was used to measure the number of changes ever performed to the selected line and thus assess its risk for bugginess, it would be reported as being changed five times, even though it was changed only three times. On another example, if the resulting history slice were used for the task of identifying the developers who ever modified the selected line and thus find experts for it, Chris would
be included in the list with an equal number of changes as Bob, even though Chris never actually changed the line.

6.1.3 One-to-One History Slicing

To address the problems introduced by the over-approximate nature of all-to-all history models, researchers developed advanced techniques that allow for the disambiguation of lines in change hunks, e.g., Canfora et al. [19]. The methods by which each such technique resolves which lines in the older revision correspond to which lines in the newer revision differ, however a common technique is to use an optimization algorithm to find the most optimal one-to-one mapping. In this work, I generalize history models that are based on such one-to-one line mapping for change hunks — I refer to these as One-to-One History Graphs.

The one-to-one history graph reduces the number of false positives (i.e., not mapping lines of code that are not truly associated). As a consequence, such one-to-one models can be expected to provide high precision, although they may also provide low recall. Moreover, the errors of under-approximation introduced by the one-to-one models can cause compounding false-negative errors when performing history slicing over multiple revisions.

Example. For my running example, Figure 6.4(a) depicts the history graph constructed using a one-to-one strategy to map lines in the older revision to lines in the newer revision for each change hunk. Compared to the actual code evolution (Figure 6.1), in Figure 6.4(b) one can observe that the one-to-one analysis produced a result that contains no spurious lines for any revision. However, as expected, it excludes many lines in each revision that should have been included. Additionally, the set of revisions that would be recommended for examination is also an under-approximation. Thus, techniques that analyzed this resulting
Figure 6.4: Existing one-to-one history graph and slice. The resulting history slice has high precision but low recall.

history slice may also provide inaccurate results. For example, if the history slice was used to measure the number of changes ever performed to the selected line and thus assess its risk for bugginess, it would be reported as being changed only once, even though it was changed three times. On another example, if the resulting history slice were used for the task of identifying the developers who ever modified the selected line and thus find experts for it, only Bob would be included in the list even though Alice also modified it.

For each such type of approaches — all-to-all and one-to-one — errors are felt, and moreover, those errors tend to compound when multiple revisions are involved in history slicing. Given this background, my goal is to enable accurate history slicing.

6.2 Automatic Fuzzy History Slicing

To address the over-approximation errors found with all-to-all analyses and the under-approximation errors found with one-to-one analyses (see Section 6.1), I created a technique that encodes a fuzzy line mapping between code revisions.
As I discussed in Chapter 2 and Section 6.1, the success of the fine-grained analysis of code evolution depends on the accuracy of the underlying history-slicing approach. To more accurately model and analyze code evolution, I propose a fuzzy variation of my automatic history-slicing approach (see Section 4.2) that can account for the indeterminate nature and degree of evolution of lines from one revision to the next. As such, I propose automatic fuzzy history slicing as an automated analysis that is based on a fuzzy model of code evolution, which I naturally call the fuzzy history graph. The abstract approach for performing fuzzy history slicing is depicted in Figure 6.5, and I describe it in the following subsections. Similarly to my automatic history slicing approach (see Section 4.2), my automatic fuzzy history slicing approach can also be parameterized in different ways.

### 6.2.1 Build Fuzzy History Graph

This process creates my novel fine-grained model of code evolution, the fuzzy history graph, by analyzing the revision-control system. Following my intuition that lines of code evolve into other lines to varying degrees, the fuzzy history graph includes a measure of the degree to which each older line became a new one.
Figure 6.6: Example of a fuzzy history graph and fuzzy history slice. Fuzzy history slices allow for tuning precision and recall in a way that is more accurate than existing, absolute techniques.

As we can observe in Figure 6.6, and similarly to my previous definition of a history graph (see Section 4.2.1), the fuzzy history graph has the shape of a multipartite graph: each part represents a revision, each node represents a line of code, and each edge represents a mapping between incident line nodes. In a fuzzy history graph, however, each edge has a weight assigned (or membership function) that estimates the degree to which an older line became a new one. Edge weights can be computed in multiple ways, and different approaches to computing them can inform different analyses of fuzzy code evolution. The fuzzy history graph can be constructed once and then updated for every new revision of the code, and it can also be reused for different history analyses.

Figure 6.6(a) depicts an example of a fuzzy history graph that was constructed from the example in Figure 6.1 that was introduced in Section 6.1. Darker lines represent a stronger evolutionary relationship, and lighter lines represent weaker evolutionary relationship.
Table 6.1: Example of a fuzzy history-slicing criterion

<table>
<thead>
<tr>
<th>File Name</th>
<th>Line Number</th>
<th>Starting Revision</th>
<th>Ending Revision</th>
<th>Degree of interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExampleFile.java</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**Fuzzy Line-mapping Technique**

I created a novel automatic fuzzy line-mapping technique to build the fuzzy history graph. My intention with this technique is to estimate the degree to which lines of code evolved into others, and to be able to capture situations where any number of lines evolve into any other number of lines. In this technique, I use a *branch and bound* optimization algorithm that tries to minimize the textual difference between mapped lines by using the Levenshtein distance [61].

This technique is depicted in Algorithm 6.1, and it can be divided into multiple steps: First, I map the lines that did not change at all between revisions with a weight of 1. Second, I process the remaining change hunks. Third, for each change hunk, I iterate through every line of code in the older revision and try to map it to the most similar concatenation of lines from the newer revision. Whenever a candidate concatenation (*branch*) is less similar to the candidate line than any of its components, I stop adding lines to that concatenation — I *bound* the search. For my experiments, I also bounded the search to a maximum of three lines of code, although this setting is configurable. Fourth, I iterate through every newer line and assign it the maximum weight identified for a concatenation that contains it. Fifth, I perform the same operation in the opposite order, iterating every line of code in the newer revision to compare it to concatenations of lines in the older revision. Sixth, I iterate through every line pair between older and newer line, and assign it the maximum weight found for it by comparing the mappings obtained in both orders.
Algorithm 6.1 Build Fuzzy History Graph

```plaintext
procedure FuzzyMapRevs(RevisionL, RevisionR)
    m ← mapUnchangedLines(RevisionL, RevisionR)
    for <linesL, linesR> ← chHunks(RevisionL, RevisionR) do
        m ← m + fuzzyMap(<linesL, linesR>)
    end for
    return m
end procedure

procedure FuzzyMapLines(<LinesLeft, LinesRight>)
    for lLeft ← LinesLeft do
        mL ← mL + fuzzyMapLine(lLeft, LinesRight)
    end for
    for lRight ← LinesRight do
        mR ← mR + fuzzyMapLine(lRight, LinesLeft)
        for <lR, rR, wR> ← mR do
            if mL contains <rR, lR> then
                mL ← mL + fuzzyMapLine(<lL, rL, max(wR, wL))
            end if
        end for
    end for
    return mL
end procedure

procedure FuzzyMapLine(<l, LinesRight>)
    cs ← {{}, 0}
    for i ← 0, MAX_CS do
        newCs ← {}
        for <c, w> ← cs do
            for r ← LinesRight do
                if c not contains (r) then
                    wR ← 1 - Levenshtein(l, r)
                    newC ← concatenate(c, r)
                    wNewC ← 1 - Levenshtein(l, newC)
                    if (wNewC > wC) & (wNewC > wR) then
                        newCs ← newCs + <newC, wNewC>
                    end if
                end if
            end for
        end for
    end for
    cs ← cs + newCs
end procedure
```

94
6.2.2 Fuzzy History-slicing Criterion

My fuzzy approach to automatic history slicing takes a fuzzy history-slicing criterion as its input. In this case, for each line of code of interest that was selected to study its history, the fuzzy history-slicing criterion allows a user or an automated analysis to specify a degree of interest in that line.

Thus, the fuzzy history slicing criterion is specified as a fuzzy set that is composed of (line, starting revision, ending revision, degree of interest) tuples. Each tuple represents a line of code of interest that is characterized by the two revisions of the program between which the history analysis is requested for that line, as well as a membership value that specifies the degree of interest (between 0 and 1, from lowest to highest) specified for that line. A fuzzy history slicing criterion can include any set of lines of code, contiguous or not, from any set of files and revisions.

In the example of Figure 6.6(b), the fuzzy history-slicing criterion would be specified as in Table 6.1.

6.2.3 Analyze Fuzzy History Graph

This process involves the analysis of the fuzzy history graph to obtain the history of the lines of code that were specified in the fuzzy history-slicing criterion, between the revisions that were specified in it. In the process of analyzing the fuzzy history graph, this process also estimates the degree to which each analyzed line on each analyzed revision belongs to the history of the lines specified in the fuzzy history-slicing criterion. Essentially, this operation answers in a fuzzy manner the question included in my example in Figure 6.1 every time a new revision is visited: *What are the corresponding lines, in other revisions, to a given line of code?* Similarly to the analysis of the history graph in my automatic history slicing
approach (see Section 4.2.3), the analysis of the fuzzy history graph may also be configured
to return different kinds of fuzzy history slices, \textit{i.e.}, history slices with context, history slices
without context, or extended history slices.

Different techniques could be created to analyze history graphs, depending on the task for
which the history slice will be employed. In order to allow experimentation, I created a novel
fuzzy history slicing technique. This technique leverages the fuzzy information stored in the
fuzzy history graph to estimate the degree to which lines of code belong to the history of
the lines selected in the fuzzy history-slicing criterion, according to their textual change over
time. I implemented this technique as a \textit{breadth-first-search} algorithm over the fuzzy history
graph, and I describe it in Algorithm 6.2. I also used this technique for my experiments in
Section 6.4.

This technique works in multiple steps: First, it creates a current set of lines to visit that
includes every line in the fuzzy history slicing criterion. Second, it iterates through the
current set of lines, and it obtains every line of code that belongs to a previous revision of
it. Second, for each previous revision, it updates its weight by multiplying it by the weight
assigned to the line from which it was visited. Third, once it has performed this process for
every line in the current set, it overwrites the current set of lines by including in it every
past revision visited. Fourth, it filters the set of current lines by, when there are duplicate
lines, keeping the line with the highest weight. Fifth, it also adds every line in the set of
current lines to the fuzzy history slice. Sixth, it checks if the visited revision is the last one
specified to visit. If it is not, it starts the process again for the current set of lines. If it
is, it stops and it returns the fuzzy history slice. The resulting fuzzy history slice contains:
(1) the visited revisions, (2) lines that correspond to the visited nodes in such revisions,
(3) computed weight for each visited node, and (4) weighted edges that connect the visited
nodes.
Algorithm 6.2 Analyze Fuzzy History Graph

procedure FuzzyHistorySlice(criterion)
    fhs ← {}
    for ⟨startLine, startRev, endRev⟩ ← criterion do
        startNode ← getNode(startLine, startRev)
        fhs ← startNode
        futureNodes ← startNode
        currentRev ← startNode.Rev
        while currentRev > endRev do
            nFutureNodes ← {}
            for futureNode ← futureNodes do
                for pastNode ← futureNode.getPastNodes() do
                    pastNode.w = pastNode.w × futureNode.w
                    if nFutureNodes contains pastNode then
                        if nFutureNodes.get(pastNode).w < pastNode.w then
                            nFutureNodes.update(pastNode, w)
                        else
                            pastNode ← nFutureNodes.get(pastNode)
                        end if
                    else
                        nFutureNodes ← nFutureNodes + pastNode
                    end if
                end for
                fhs ← fhs + pastNode
                currentRev ← pastNode.Rev
            end for
            futureNodes ← nFutureNodes
        end while
    end for
    return fhs
end procedure
6.2.4 Fuzzy History Slice

The final output of the fuzzy history slicing process is a fuzzy history slice, i.e., a fuzzy set of the lines and revisions that belong to the history of the fuzzy history-slicing criterion. The fuzzy history-slicing approach can be configured to compute different kinds of fuzzy history slices, much like regular history slices (see Section 4.2.4).

The fuzzy history slice for a set of lines of code of interest contains: (1) the revisions of the program that modified those lines, (2) the lines that correspond to them in such revisions, (3) a weight for each included line indicating the degree to which it belongs to the fuzzy history slice, and (4) the weighted edges that connect the included lines between consecutive revisions, indicating the degree to which such lines evolved into each other.

A fuzzy history slice can be represented as a weighted multipartite graph, which is a subset of the fuzzy history graph. However, the edges of the fuzzy history slice may have different values. Additionally, the nodes in a fuzzy history slice are also weighted. Nodes in a fuzzy history slice are specified as tuples with the same values as nodes in a history slice, as well as a new value that specifies the degree to which it belongs to the history slice: \( \langle \text{file name}, \text{line number}, \text{revision}, \text{meta-data}, \text{membership degree} \rangle \).

Figure 6.6(b) depicts a fuzzy history slice that was automatically obtained from the example in Figure 6.1 that was introduced in Section 6.1. The lines in each revision that are included in the fuzzy history slice are colored solid red, and their membership values are represented by the saturation of those nodes (or darker gray for a gray-scale reproduction of this document). We can observe that, among the lines in each revision with the highest membership, these more strongly agree with the history slice assessed by a human in Figure 6.2. Moreover, the revisions that would be recommended for examination also more strongly match the revisions that a human assessed.
6.3 Evaluation of the Accuracy Provided by Automatic Fuzzy History Slicing for a Single Code Change

I evaluate the accuracy provided by my automatic fuzzy history slicing approach both for individual code changes and for full code histories. In this first experiment, I evaluate the accuracy with which a fuzzy history graph represents code evolution for individual code changes. The goal of this experiment is to evaluate whether, and to what extent, fuzzy history graphs improve the accuracy (in terms of precision and recall) of current fine-grained code-evolution models.

6.3.1 Experimental Design

In order to perform this experiment, I implemented my technique to build fuzzy history graphs that I described in Section 6.2.1, and I replicated two state-of-the-art techniques to build: a one-to-one history graph and an all-to-all history graph. Then, I built these three models for 300 code changes that I sampled from three real-world software projects. Finally, I measured the accuracy with which each model represented the actual evolution of the code in each change. I obtained the ground-truth evolution of code from four software developers who manually assessed the mappings for the studied code changes.

6.3.2 Experimental Subjects and Sampling

I used three real-world software projects to sample changes for my evaluation: APACHE COMMONS IO [6], which is a library to perform input and output functionality; MOZILLA RHINO [69], which is a JavaScript parser written in Java; and ASPECTJ [1], which is an aspect-oriented programming framework for Java. APACHE COMMONS IO has a size of
26,000 lines of code and a history of 11 years, Mozilla Rhino is composed of 185,000 lines of code and has a history of 12 years, and AspectJ’s size is around 510,000 lines of code and its history spans 10 years.

I randomly sampled 300 change hunks from my subjects: 100 samples from each subject. I extracted the change hunks produced by all the changes in a subject by using `diff`. I sampled change hunks that had between one and fifteen lines in the older or newer revision, so that they had a manageable size to ask developers to assess the ground truth of their evolution.

### 6.3.3 Ground Truth

To assess the ground truth for the sampled change hunks, four participants independently and manually assessed them and defined their judgment of the correct and ideal mapping. I recruited participants that had between 5 and 12 years of experience programming in Java to ensure that they were capable of assessing the ideal mapping.

Confirming the subjective nature of “truth” that I described in Section 6.1, I observed the fuzzy nature of actual code evolution: 31% of the sampled change hunks obtained different assessments from different developers. For example, consider the following change obtained from Apache Commons IO depicted below:

```diff
31,32c31,42
< /** Singleton instance of false filter */
< public static final IOFileFilter INSTANCE = new FalseFileFilter();
---
> /**
> * Singleton instance of false filter.
> * @since Commons IO 1.3
```
For this change, two participants assessed that line of code 32 in the older revision evolved into line of code 35 in the newer revision. However, one other participant assessed that line of code 32 in the older revision evolved into both lines of code 35 and 42 in the newer revision. We can observe how both kinds of assessments would be reasonable. Some people could consider that line 32 only evolved into line 35, since they are the most similar ones. Other participants may assess that line 32 also evolved into line 42 in the newer revision, since line 42 also contains the declaration of the INSTANCE variable.

For each older line that was assessed to evolve into a newer line, I assigned a ground-truth weight to such evolution equal to the number of developers that assessed it to exist, divided by the total number of developers that assessed the change hunk. For example, if three developers assessed the change hunk from revision 4 to revision 5 of my example in Figure 6.1 and only one developer assessed line 4 from revision 4 as evolving into line 5 in revision 5, that mapping would have a weight of 0.33 in the ground truth for that change hunk. Although I did not time the participants during the process, I estimate that each participant took around 3 hours to assess the ground truth for the 300 sampled changes.
6.3.4 Independent Variables

I used three independent variables in this experiment: the change hunk type, the evolution model, and the similarity threshold for the evolution model. Next, I describe and motivate these independent variables.

Independent Variable 1: Change Hunk Type

As observed in Section 6.1, the accuracy of current models is especially impacted when lines of code evolve into or from multiple other lines. In order to study both cases where current models are expected to have limitations and cases in which they are not, I classify the studied change hunks into four categories. The classification divided the sample into: 145 one-to-one change hunks, 36 one-to-many change hunks, 34 many-to-one change hunks, and 85 many-to-many change hunks.

**One-to-One.** Every line of code in the older revision evolved into at most one line of code in the newer revision.

**One-to-Many.** At least one line of code in the older revision evolved into multiple lines of code in the newer revision.

**Many-to-One.** Multiple lines of code in the older revision evolved into the same line of code in the newer revision.

**Many-to-Many.** At least one line in the older revision evolved into multiple lines in the newer revision, and multiple lines in the older revision evolved into the same line in the newer revision.
Independent Variable 2: Evolution Model

I compare the accuracy of the fuzzy history graph to that of the other two existing evolution models at the line-of-code granularity — one-to-one and all-to-all history graphs. I build the existing models by replicating a corresponding state-of-the-art technique.

One-to-One History Graph. Each line of code in the older revision is modeled as evolving into at most one line of code in the newer revision. I build this evolution model using my one-to-one approach to building the history graph that I described in Section 4.2.1.

All-to-All History Graph. Every line of code in the older revision is modeled as evolving into every line of code in the newer revision. I build this evolution model using the technique proposed by Zimmermann et al. [92].

Fuzzy History Graph. Every line of code in the older revision is mapped as evolving into every line of code in the newer revision to a different extent, which is indicated by a weight. I build this evolution model using the technique described in Section 6.2.1.

Independent Variable 3: Similarity Threshold

Most of the techniques that build a one-to-one model use a similarity threshold to avoid modeling evolutions of lines that are highly dissimilar. In order to apply the same treatment to all the evaluated models, I discard mapped lines with similarity below the similarity threshold — calculated as one minus the Levenshtein distance — after the one-to-one model and the fuzzy history graph are built. Because the all-to-all model does not account for the similarity of the modeled evolutions, it is not affected by the similarity threshold. I use ten different values for the similarity threshold: 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, and 0.0.
6.3.5 Dependent Variables

I measure the accuracy with which a model represents code changes by measuring three dependent variables: Precision (Formula 6.1), Recall (Formula 6.2) and F-measure (Formula 6.3). I used the fuzzy variation of these metrics, since I assessed the ground truth of the evolution of a change hunk as a fuzzy set of line mappings. The fuzzy-set definitions of precision and recall depend on fuzzy cardinality and intersection. The cardinality of a fuzzy set is defined as the sum of its membership degrees, as in Formula 6.4. The intersection of two fuzzy sets is defined as the set of all the elements that are in common, each of which is assigned a membership degree equal to its minimum membership degree from both sets, as in Formula 6.5.

\[
\text{Precision} = \frac{|\text{Modeled mappings} \cap \text{Ground-truth mappings}|}{|\text{Modeled mappings}|} \quad (6.1)
\]

\[
\text{Recall} = \frac{|\text{Modeled mappings} \cap \text{Ground-truth mappings}|}{|\text{Ground-truth mappings}|} \quad (6.2)
\]

\[
\text{F-measure} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (6.3)
\]

\[
|A| = \sum \mu_A(x), \forall x \in X \quad (6.4)
\]

\[
\mu_{A \cap B}(x) = \min\{\mu_A(x), \mu_B(x)\}, \forall x \in X \quad (6.5)
\]

6.3.6 Results

In the results of this experiment, I observed the limitations in terms of precision and recall of existing code-evolution models that I described in Section 6.1. I also observed that the
Figure 6.7: Mean precision, recall, and f-measure provided by the different code-evolution models for the different change-hunk types. Higher scores are better.

Fuzzy history graph mitigated such accuracy limitations, since it provided a higher f-measure — which is a combined metric of precision and recall — than existing models in general for all the evaluated change types.

I depict in Figure 6.7 the mean precision, recall and f-measure provided for each type of change hunk by each model in a matrix of 15 graphs. From top to bottom, each row of graphs shows mean precision, mean recall, and mean f-measure scores, respectively. From left to right, each column of graphs shows the results provided by each model for one-to-one, one-to-many, many-to-one, and many-to-many change hunks, as well as the aggregated mean results for all kinds of change hunks. The vertical axis of each graph represents mean precision, mean recall, and mean f-measure scores, in the first, second, and third row, respectively. Higher mean precision, mean recall, and mean f-measure scores are better, in terms of accuracy. The horizontal axis of all graphs represents the similarity threshold used. Within each graph, red squares represent the scores obtained by the all-to-all model, yellow
crosses represent the one-to-one model, and green triangles represent the fuzzy history graph.

The one-to-one model provided the highest mean precision and the lowest mean recall of all models for all types of change hunks and similarity thresholds. I expected the one-to-one model to provide high precision, because it includes at most one mapping per older line of code, and is therefore less likely to include false positives than other models. For the same reason, I also expected the one-to-one model to provide low recall. As the similarity threshold decreased, the recall provided by the one-to-one model increased, because it included more mappings. However, the recall stabilized below the similarity threshold of 0.3 value, since the one-to-one model did not include more than one mapping per line, regardless of the similarity threshold used. This characteristic also caused the one-to-one model to provide higher recall for the one-to-one change hunks than for the other types of change hunks. These results demonstrate the recall limitations of one-to-one models that I described in Section 6.1.3.

The all-to-all model provided the lowest mean precision and the highest mean recall of all models for all types of change hunks and similarity thresholds. Because the all-to-all model mapped every older line of code to every newer line of code in a change hunk, it always reached a recall of 1.0. However, it modeled many false positives, which caused it to provide low precision for all change-hunk types, particularly for one-to-one change hunks. These results show the limitations of all-to-all models in terms of precision that I described in Section 6.1.2.

The fuzzy history graph provided mean precision and mean recall values between those provided by existing models. As I anticipated, the fuzzy history graph always provided higher precision than the all-to-all model and higher recall than the one-to-one model. Moreover, precision was positively correlated with the similarity threshold, and recall was negatively related with the threshold. As such, the fuzzy history graphs allow a more flexible increase of the recall provided by the one-to-one model without sacrificing as much precision as the all-to-all model. I also observed that the potential for increasing recall is much higher for
one-to-many, many-to-one and many-to-many change hunks, because the one-to-one model already provides a quite high recall for one-to-one change hunks. The fuzzy history graph also showed benefits in terms of the f-measure, which provides a balanced metric between precision and recall. In terms of f-measure, the fuzzy history graph reached a higher value than the existing models for most change-hunk types and most similarity thresholds, specifically for similarity threshold 0.6, which is the recommended value by most techniques that build one-to-one models, \textit{e.g.}, [19, 8]. The exception was the case of one-to-one change hunks, for which the one-to-one model provided higher f-measure than the fuzzy history graph. Still, the fuzzy history graph provided a higher f-measure on average for all change-hunk types than the other models.

### 6.4 Evaluation of the Accuracy Provided by Automatic Fuzzy History Slicing for Full History Slices

In the second experiment, I evaluate the accuracy provided by automatic fuzzy history slicing for full history slices when compared to existing history-slicing approaches. This experiment complements the previous experiment from Section 6.3 by evaluating whether, and how, the accuracy improvements that I observed in the previous experiment accumulate over multiple revisions.

#### 6.4.1 Experimental Design

In this experiment, I built fuzzy, one-to-one, and all-to-all history graphs for the whole history of three real-world software projects. I also implemented the fuzzy history-slicing technique defined in Section 6.2.3. Then, I randomly sampled multiple lines of code from each of the three software projects, and I computed their history slices by using my novel
automatic fuzzy history-slicing approach over a fuzzy history graph, and over two existing models of code-evolution: a one-to-one model and an all-to-all model. Finally, I measured and compared the accuracy of the obtained history slices.

6.4.2 Experimental Subjects, Sampling and Ground Truth

In this experiment, I used the same subjects as my previous experiment: Apache Commons IO, Mozilla Rhino, and AspectJ. For every subject, I randomly sampled history-slicing criteria of a line of code that experienced at least five changes in its prior history. I assigned every line in a sampled history-slicing criterion a relevance weight of 1.0. For each history-slicing criterion, I manually traversed the history of the program and determined the revision in which the selected line was first authored by iteratively using git blame over the history of the program. I also manually identified the ground truth of which lines of code corresponded to each sampled line in its first-authored revision by iteratively comparing each subsequent revision. I attributed a ground-truth relevance of 1.0 to every line of code that I assessed as belonging to the history of the studied lines, and a ground-truth relevance of 0.0 to any other line. Given the arduous nature of producing correct, manually assessed ground-truth assessments, I conducted this experiment with a total of 15 history-slicing criteria, randomly chosen, with five criteria per subject program. From history-slicing criterion to its first authorship, the length of the history slices spanned a median of five years of development, with a maximum length spanning nine years of development. These changed files, from criterion to first authorship, spanned on average over 100 revisions per criterion, which had to be manually inspected and traversed.
6.4.3 Independent Variable: Evolution Model

I compare the accuracy of fuzzy history slicing when using the fuzzy history graph and when using the existing evolution models — one-to-one and all-to-all.

**One-to-One.** I built a one-to-one history graph using the technique described in Section 4.2.1, with a similarity threshold of 0.6. Because the one-to-one history graph produces a discrete mapping, I assigned all the edges of the graph a weight of 1.0.

**All-to-All.** I built an all-to-all history graph using the technique proposed by Zimmermann *et al.* [92]. Because the all-to-all history graph also produces a discrete mapping, I assigned all its edges a weight of 1.0.

**Fuzzy History Graph.** I built a fuzzy history graph using the technique described in Section 6.2.1.

6.4.4 Dependent Variable: Accuracy

I measured the accuracy of a computed fuzzy history slice by evaluating the weighted lines contained in its earliest revision with the Normalized Discounted Cumulative Gain (NDCG) metric [48]. NDCG is commonly used in information retrieval to assess the quality of weighted results. I chose the NDCG metric because it provides a convenient and standard way to evaluate (potentially) weighted results in a way that accounts for positions of multiple correct and incorrect results, and accounts for the size of the result set. NDCG evaluates a weighted set as a recommendation that is sorted in terms of the weights of its elements. NDCG results range from 0 — for the worst recommendation possible — to 1 — for the best recommendation possible. I used the NDCG formula proposed by Burges *et al.* [15], as in Equations 6.6 and 6.7.
\[ \text{NDCG}_p = \frac{\text{DCG}_p(\text{recommendation})}{\text{DCG}_p(\text{ideal recommendation})} \] (6.6)

\[ \text{DCG}_p = \sum_{i=1}^{p} \frac{2^{\text{rel}_i} - 1}{\log_2(i + 1)} \] (6.7)

NDCG is calculated as the Discounted Cumulative Gain (DCG) score for a recommendation, divided by the DCG score of an ideal recommendation — in which the correct lines are contained in its top positions. In the DCG formula (Equation 6.7), \(i\) represents the top \(i^{th}\) position inside a recommendation, and \(\text{rel}_i\) represents the actual, ground-truth relevance of the item recommended in position \(i\). I attributed a ground-truth relevance of 1.0 to every line of code that I assessed as belonging to the history of the studied lines, and a relevance of 0.0 to every other line. Since none of the relevant lines were recommended with any model after position 50, I used NDCG\(_{50}\) scores. In some cases, multiple lines were recommended with the same weight — as was the case with the absolute, discrete models. In such cases, I computed the average-case result for the order of the recommendation, \(i.e.,\) as if the recommendation results were randomly ordered over an infinite number of orderings.

Let’s consider one of the studied history-slicing criteria as an example. This criterion was originated 6 revisions in the past of the selected revision, and it included only line 591 in that revision. For this criterion, the One-to-One approach provided an empty recommendation in the originating revision, since it only reached 4 revisions before the one included in the criterion. As a result, the One-to-One approach obtained an NDCG\(_{50}\) score of 0.0 for this criterion. The All-to-All approach included with a weight of 1.0 lines 592, 593, and 591 in the originating revision for this criterion. Since all the recommended lines obtained the same score, I considered their average random ordering, in which line 591 is recommended in position 2. This way, I calculated the NDCG\(_{50}\) score of this recommendation as in Equation 6.8.
Figure 6.8: Accuracy provided by history slicing for the different code-evolution models. Higher NDCG subscripts 50 scores are better.

\[
\text{NDCG}_{50} = \frac{2^{-1} \log_2(2) + 2^{-1} \log_2(3) + 2^{-1} \log_2(4)}{1.0 + 0.0 + 0.0} = 0.63
\]

Finally, for this criterion, the Fuzzy History Slicing approach recommended in the originating revision line 591 with a weight of 0.79, line 593 with a weight 0.13 and line 592 with a weight of 0.12. After sorting the recommended lines by their score, the Fuzzy History Slicing approach recommended the lines of code that belong to the history of the criterion in the originating revision in this order: 591, 593, 592. I calculated the NDCG subscripts 50 score of this recommendation as in Equation 6.9.

\[
\text{NDCG}_{50} = \frac{2^{-1} \log_2(2) + 2^{-1} \log_2(3) + 2^{-1} \log_2(4)}{1.0 + 0.0 + 0.0} = 1.0
\]

### 6.4.5 Results

In this experiment, I observed that existing models of fine-grained code evolution — one-to-one and all-to-all — provided limited accuracy (in terms of NDCG subscripts 50 score) when considering full history slices, since their limitations discussed in Section 6.1 propagated through multiple
revisions — including too few revisions in the case of the one-to-one model and too many lines in each revision in the case of the all-to-all model. My fuzzy history slicing approach provided higher accuracy (higher NDCG$_{50}$ score) than existing models for full history slices. Figure 6.8 shows, for each subject, three box plots that represent the distribution of NDCG$_{50}$ scores provided by each evolution model.

The one-to-one model provided a median NDCG$_{50}$ score of 0.0 for all subjects, which is displayed as a flat bar at the 0.0 score in Figure 6.8. For most of the studied lines of code, the one-to-one model did not capture their changes through enough revisions to track their history completely back to the selected past revision — such a limitation of the one-to-one model for multi-revision analysis was demonstrated in the example in Section 6.1. the one-to-one model applies a similarity threshold to decide which lines evolved into others. As a consequence of this premature end of a line’s history, the history slicing technique found that no lines in the past revision were part of the history of most of the studied lines and then returned an empty recommendation for them.

The all-to-all model obtained a median NDCG$_{50}$ score of 0.5 in Mozilla Rhino and a median NDCG$_{50}$ score of 0.63 — in Apache Commons IO and AspectJ. Because the all-to-all model often contains multiple mappings for each line of code, the history of a single line of code may include increasing numbers of lines as the traversal length increases over history. However, in the end, I were surprised to find that performing history slicing with the all-to-all history graph produced results that usually included the correct, ground-truth lines. I speculate that, at least for my randomly chosen slicing criteria, the compounding imprecision that I discussed in the example program in Section 6.1 was not found — at least not to that extent.

Nevertheless, the fuzzy history graph obtained a median NDCG$_{50}$ score of 1.0 in all subjects, which is displayed as a flat bar at the 1.0 score in Figure 6.8. For all the studied lines of code (except for one in AspectJ), the technique recommended all their actual correspond-
ing lines of code in the top position of its recommendation. Unlike the one-to-one model, the fuzzy history graph contains all potential mappings, so the history of lines of code did not end prematurely. Unlike the all-to-all model, the fuzzy history graph assigned weights to mappings to represent their strength, so the technique could provide more informed recommendations by using the strength of the changes experienced by the line of code. The weights assigned to mappings in the fuzzy history graph allowed the technique to select the paths with highest overall similarity in the history of the studied lines and thus recommend their actual corresponding lines in the top positions of its recommendation.

6.5 Threats to Validity

A few factors may be mentioned as threats to validity for this evaluation. An external threat to validity is that my proposed technique to build the fuzzy history graph may not capture some complex changes, such as movements of code between files, since it is based on textual differencing. Such complex changes are more accurately captured by semantic differencing techniques, such as those that compare the abstract syntax tree of the program, *e.g.*, [7]. I intend to study in future work the accuracy improvements that may be provided by modeling fine-grained code history with such semantic approaches in a fuzzy manner. In these experiments, however, my goal is to study the accuracy improvements provided by a fuzzy approach to modeling and analyzing code history. The limitation for capturing moved regions of code between files affect both my proposed textual differencing technique and all the other techniques studied in my experiments. In that respect, I believe that this limitation did not affect the results of my experiments. Additionally, the fuzzy history graph model may address this limitation, since it allows its construction with other line mapping techniques, *e.g.*, [27].

Another external threat to validity is that my technique may not scale to large code bases.
The two main factors that affect the efficiency of the automatic fuzzy history slicing process are the size and age of the selected code. In order to account for this factor, I performed my experiments over three different subject programs of up to 510 KLOC in size and up to 12 years of development. I also evaluated fuzzy history slicing over code files that included an average of 100 revisions, and I selected code that spanned an average of five years of development, and a maximum of nine. In the worst case, the fuzzy history graph may only become as large and complex as the all-to-all model, since they include the same number of nodes and edges. Moreover, in the case that improvements were needed in the efficiency of traversing the fuzzy history graph, it allows the use of the weights in its edges for live paths trimming.

A threat to internal validity is that the accuracy improvement provided by the fuzzy history graph may not actually turn into accuracy improvements for some code history analyses. In order to account for this threat, I performed Experiment 2. In Experiment 2, I observed how the balance between precision and recall provided by the fuzzy history graph enabled fuzzy history slicing to automate the analysis operation of obtaining the complete history of a set of lines of code with higher accuracy than when using other existing models. While it is possible that some other code history analyses will not benefit from a balanced precision and recall, the fuzzy history graph will still allow them to configure such balance at different levels.

Another internal threat to validity is that the participants that assessed the ground truth of change hunks were known to me personally. I took extensive care to avoid bias caused by this factor. I informed the participants that no assessment would be wrong, and I asked them to assess the ground truth of code changes according to their own concept of code evolution, connecting any lines of code to any others between consecutive revisions. Additionally, none of the participants knew the behavior of the algorithms that would be evaluated with the ground truth obtained from their assessment.
Another internal threat to validity is that the ground truth was assessed by people who were not familiar with the code base, and who sometimes disagreed. I believe that this factor did not significantly affect the results of my experiments, since the ground truth of code changes was assessed by software developers that had a median of 6 years of development expertise with Java. Additionally, since I used the same ground truth data to study all techniques, any mistakes made in assessing the ground truth affected all of them equally. Finally, I consider that the disagreements that appeared when assessing code changes reflect the actual subjectivity of code history. Moreover, I consider the studied change hunks to be representative of how often complex code changes appear in the field, since I sampled the change hunks randomly from real-world software projects.
Chapter 7

Related Work

Four main bodies of research relate to the work proposed in this dissertation: analyses, models, and visualizations of code evolution, and expertise-assessment techniques.

7.1 Analyses of Code Evolution

Researchers proposed techniques to analyze the multi-revision evolution of code for specific purposes and at different granularities, e.g., Kim et al. [55] and Duala-Ekoko and Robillard [28] track the history of code fragments that contain code clones to study their evolution. Herzig and Zeller [45] analyze multi-revision changes to methods to predict defects. Hassan and Holt [41] analyze the evolution of methods to extract change rationale from them. In contrast, the automatic technique proposed in this dissertation — automatic history slicing — facilitates multi-purpose, multi-revision analyses of code evolution at a finer granularity: a single line of code.
7.2 Models of Code Evolution

A number of researchers proposed line-mapping techniques to model code evolution at the line-of-code granularity. One example of line-mapping technique that produces an all-to-all model is the annotation graph, proposed by Zimmermann et al. [92]. Many approaches have been proposed to model code evolution in a one-to-one fashion. Canfora et al. [20, 19], Chen et al. [21], Williams and Spacco [89] use a line-mapping technique that involves a two-step process, where they first perform an inexact difference of revisions, and then refine it by using an optimization algorithm. Reiss [73] proposed a group of line mapping techniques, some of which considered adjacent lines. Asaduzzaman et al. [8] proposed a language-independent line-mapping technique that also detects lines that evolve into multiple others, although only when they change little and are contiguous. In this dissertation, I describe the first explicit weighted model of code evolution: the fuzzy history graph.

Other models represent code evolution at different granularities. Hassan and Holt model code evolution at the method-level [40, 41]. Hata et al. [42] proposed a model for tracking the history of methods and fields that accounts for renames. Godfrey and Zou [37] and Wu et al. [90] also track the history of methods and fields and detect their splits and merges. Zimmermann et al. [91], Fluri et al. [31] and Spacco and Williams [83] capture differences at the statement level. Girba and Ducasse [35] proposed an evolution meta-model of code evolution at multiple levels of granularity. Other researchers proposed more sophisticated algorithms by performing the mapping over models of the program, e.g., [7, 64] allowing the detection of moved code, e.g., [47, 27] or providing techniques for specific domains, e.g., [29]. Davies et al. [24, 25] proposed “software bertillonage” to track the evolution of releases of code outside the revision-control system. Finally, Kim and Notkin [54] presented a survey of techniques that track program elements between revisions. To the extent of my knowledge, the fuzzy history graph is the first fine-grained code-evolution model to quantify the evolution of code and preserve it as a fuzzy measure to augment the model itself, thus
enabling multi-revision analyses of code evolution at the line-of-code granularity to leverage such fuzzy measure.

### 7.3 Visualizations of Code Evolution

Multiple visualizations have been proposed for visualizing the evolution of source code. In terms of the dimension of the program, many authors propose visualizations that show the evolution of the whole software system with a level of detail of source code files, e.g., [58, 36, 88, 86]. In terms of the dimension of history, such techniques allow the visualization of the whole history of the software system.

In order to provide a more detailed understanding, other authors have proposed techniques that visualize the history of source code at finer levels of granularity. Some techniques display evolution information about source code methods [41, 46, 43]. Hassan and Holt [41] and Holmes and Begel [46] annotate source code methods with additional information taken from commit operations. Hattori et al. [43] list all the changes that affected a source code method. Then, users need to select one change at a time to see the diff that it caused. In terms of the dimension of history, Hassan and Holt’s and Holmes and Begel’s approaches display the complete history of the source code method. Hattori et al., however, offer a finer-grained visualization of the dimension of history, by displaying changes captured in the IDE between commit operations.

Other techniques provide a visualization at the line-level granularity, e.g., [10, 21, 87, 63, 13]. Ball and Eick [10] display aggregated evolution information for every line of code. They annotate a SeeSoft [30] visualization with color codes to indicate historical properties, such as last author, code age, and ratio of bug-fixing changes to feature-addition changes. Chen et al. [21] display those lines in the current revision of code that have ever been changed.
with a commit message that matches a user-specified query. Voinea et al. [87] join multiple SeeSoft views of the source code to represent the evolution of a source-code file with line-level granularity. They also allow users to see the diff between two consecutive revisions. Lommerse et al. [63] extend Voinea et al.’s approach by allowing the inclusion of the evolution of multiple files. Bradley and Murphy [13] display, for each line of code, information about its last change, such as its author, date, and commit message. In terms of the dimension of history, Ball and Eick, Voinea et al., and Lommerse et al. display information about the whole history of a file (Ball and Eick do so in an aggregated form), and Chen et al. and Bradley and Murphy display information only about the last change to each line of code.

In contrast, Chronos is the only tool that allows the visualization of all and only those revisions that affected a specified set of lines of code and that shows the meta-data as well as the contents of the code for said revisions in a single visualization.

7.4 Expertise Assessment

Existing work in the area of automatically determining developer expertise generally falls into two categories: those that leverage the natural-language bug reports in a bug-tracking system to assign a developer; and those that can identify the most knowledgeable developer given a location in the source code.

7.4.1 Bug-Report Expertise

Most of the bug-assignment techniques presented to date model the expertise required to fix a bug by analyzing the contents of the bug report and model developer expertise by analyzing the history of the bug reports of the program. These techniques create both models as a set of weighted keywords that they extract from each source of information. The weight of each
keyword represents its relevance in the model. Then, some of these techniques use a similarity formula to compare the two models and thus recommend the most suitable developers to fix the bug. McDonald et al. [66] use cosine distance. Canfora et al. [17, 18] use their own similarity metric over the intersection of keywords used both in the bug report and in the history of bug reports and the commit messages in the source code repository. Baysal et al. [11] allow different similarity metrics and include additional factors in the expertise profiles, such as developer workload. Other techniques use a machine learning classifier to assess the expertise offered by each developer to fix the bug. Cubranic et al. [22] used a Naïve Bayes classifier. Anvik et al. [3, 4] and Di Lucca et al. [26] compared the performance of various classifiers — such as Naïve Bayes, Support Vector Machines, C4.5, Expectation Maximization, Conjunctive Rules, and Nearest Neighbor. Jeong et al. [49] and Bhattacharya et al. [12] improve the effectiveness of various classifiers by accounting for historical bug reassignment information. Aljarah et al. [2] apply term selection before training a Naïve Bayes classifier. Tamrawi et al. [84] use fuzzy sets to improve the effectiveness of various classifiers. Park et al. [70] use natural-language modeling and include in their algorithm considerations to balance developer workload.

Other bug-assignment techniques model the expertise required to fix the bug by analyzing the contents of the bug report and model developer expertise by analyzing the history of the source code of the program. These techniques use various approaches to match the two models and assess the expertise offered by each developer to fix the bug. Matter et al. [65] use a cosine distance metric to measure the similarity between the sets of weighted keywords that they extracted from the bug report and from the textual contents of every version of the source code. Kagdi et al. [53, 51], Linares-Vasquez et al. [62] and Shokripour et al. [80] first match the keywords in the bug report to the names of source-code artifacts in order to localize the areas of the code where the bug may be located. Then, they analyze the history of the localized code to estimate each developer’s expertise in it.
Finally, WhoseFault is the first bug-assignment approach that models the expertise required to fix the bug by using bug executions. After WhoseFault, one other technique was proposed by Perscheid et al. [72] to recommend developers to fix bugs based on bug executions. Perscheid et al.’s technique recommends developers for a bug based on the similarity between the set of methods that were executed by the bug and the set of methods for which each developer was the last one to modify.

7.4.2 Source-Code Expertise

Another area of existing work automatically selects a developer who has expertise in a specific location of the code. In these techniques, a person chooses a location in the source code, and the technique is able to assess the developer who has the most expertise for that location. This work mines the history of the source code, capturing all changes to the system and the developers who made those changes.

A number of researchers have investigated this area of mapping “expert” developers to components of source code (e.g., [66, 68, 36, 52, 33, 34, 74]). Each such research endeavor has examined different factors in mapping developers to components. For example, McDonald and Ackerman [66] suggest that the developer who most recently changed a file is the expert for the entire file, and Mockus and Herbsleb [68] suggest that the expert is the developer who made most changes to a file. Another example includes Fritz et al. [34], who additionally consider factors such as frequency of reading the code.

If such techniques were to be applied to finding the most appropriate developer to fix a fault, they would require that a person can determine which parts of the program are faulty, which is a task that, itself, requires specific expertise. Moreover, the granularity of such approaches is typically at the file, class, or method level — tracking developer expertise by method or by file instead of tracking it for a set of lines of code, which could be located in different parts of
the system. Also, these approaches can only point to the developers that have expertise in
one particular area of code at a time — that is, they do not take a number of locations that
may be related in some way and determine the developer who has the best expertise across
this group. For example, for a single location, developer A may have the most expertise, and
for another location, developer B may have the most expertise. However, if we are seeking
the developer who has the most expertise in both of those locations, developer C may be
the best choice.
Chapter 8

Conclusions

This dissertation presents a characterization of the multi-revision, fine-grained analysis of source-code history and a series of approaches for its partial automation. In the characterization of this process, I reveal the limitations of its fundamental operation: history slicing, i.e., obtaining the subset of the history of the program that corresponds to a set of lines of code of interest. To address the efficiency limitations, I create Automatic History Slicing: an approach that automates the operation of history slicing by identifying the historical revisions and lines of code inside those revisions that correspond to any given set of lines of code. Then, I create two techniques and tools to provide support for answering developer questions. First, I create CHRONOS, a technique and tool that support developers in answering questions about source code history by allowing them to interactively investigate the history of any set of lines of code. Then, I also create WHOSEFAULT, a technique and tool that provides developers with automatic answers to a prevalent developer question: who are the most suitable developers to fix this bug? Finally, I create Automatic Fuzzy History Slicing: an approach that enables the modeling and analysis of fine-grained code history in a fuzzy manner that accounts for variable degrees of code evolution.
This dissertation also presents multiple experiments that evaluate the techniques and tools presented in it. In these experiments, Automatic History Slicing improves the efficiency of existing techniques for history slicing by allowing developers to inspect up to three orders of magnitude less information in the history-slicing process. CHRONOS allows developers to answer common questions about code history correctly in more than triple the cases and in around half the time than when they used current revision-control systems. WHOSEFAULT allows developers to automatically answer a prevalent developer question — who are the most suitable developers to fix this bug? — with as much effectiveness — and in many cases better — as existing techniques for the same purpose, by also removing their requirement of a human writing a description of the bug. Finally, Automatic Fuzzy History Slicing improves the accuracy of existing techniques for modeling and analyzing fine-grained code history by providing a higher f-measure — i.e., balance between precision and recall — of up to 0.29 — on a scale from 0.0 to 1.0.

In conclusion, the techniques, tools and experiments presented in this dissertation provide enough evidence to support the thesis that: The multi-revision, fine-grained analysis of source-code history can be partially automated in a way that is efficient, that provides support for answering developer questions, and that accurately models source-code evolution.

8.1 Contributions

The research presented in this dissertation provides a number of contributions to the field of software engineering:

1. A set of definitions and characterization of the process of the multi-revision, fine-grained analysis of source-code history.

2. A set of definitions and characterization of the process of obtaining the subset of the
history of the program that corresponds to a set of lines of code: history slicing.


4. A model of multi-revision, fine-grained, source-code history: History Graph.

5. A tool that enables to automatically obtain and interactively visualize the history of code selections to support developers in answering questions about source code history: Chronos.

6. A visualization technique to display multi-revision, fine-grained history of source code.

7. A technique and tool that automate the multi-revision, fine-grained analysis of source-code history to support developers in answering a prevalent developer question (who is the most suitable developer to fix a given bug?): WhoseFault.


9. A model of multi-revision, fine-grained, source-code history that accounts for variable degrees of code evolution: Fuzzy History Graph.

10. A many-to-many line-mapping technique that computes fuzzy correspondences of lines of code between consecutive revisions.

11. An empirical evaluation that provides evidence of the efficiency limitations of current approaches for history slicing and the efficiency improvements of Automatic History Slicing.

12. An empirical evaluation that provides evidence of the improvements in effectiveness and efficiency that Chronos provided for developers to answer questions about source code history.
13. An empirical evaluation that provides evidence for the fact that WhoseFault allowed developers to automatically answer a prevalent developer question (*who is the most suitable developer to fix a given bug?*) with as much effectiveness as current techniques without requiring humans to write bug descriptions.

14. An empirical evaluation that provides evidence of the accuracy limitations of current approaches for history slicing and the accuracy improvements of *Automatic Fuzzy History Slicing*.

### 8.2 Implications for Future Research

The contributions of this dissertation motivate future research efforts in three directions:

**Empirical Study of Code Evolution.** In this dissertation I presented Automatic Fuzzy History Slicing, a fuzzy model and analysis of code history that improved the accuracy of existing approaches for modeling code history by including a measure for varying degrees of code evolution. This technique was motivated by a study in which I observed that different developers provided different answers when asked which lines of code evolved into which others for real-world code evolutions. This work motivates further empirical studies of code evolution that may provide even more knowledge about the ways in which code really evolves. It may be the case that different approaches to software development change the way in which code evolves. For example, more and more developers are using collaborative development environments in which they make changes to the same files at the same time. Developers are also more often using distributed version control, in which branching is extensively used. Empirical studies of code evolution in different approaches to collaborative software development may also uncover different requirements for modeling such code evolution.
New Applications of Code-history Analysis. This dissertation presented evidence for the benefits provided by the multi-revision, fine-grained code-history analysis for supporting the answer to multiple developer questions. This work motivates future research towards developing techniques that can support developers in answering additional questions for other software-engineering tasks. For example, new code-history analysis techniques may support developers in tasks such as: finding the rationale of code, i.e., why was this code implemented in this way?, or identifying bug fixes, i.e., how have bugs like this one been fixed in the past?

Analysis of Additional Historical Artifacts to Support Software Development. This dissertation presented and evaluated the support that the analysis of code history provided to support developers in answering questions. In a similar manner, the analysis of other software artifacts may also provide support for developers in performing development tasks. Software development projects store a wide variety of artifacts in repositories besides source code, e.g., bug reports, build scripts, test suites, or developer conversations in forums. The analysis of such artifacts over time may be useful for applications such as: identification of developer expertise by analyzing the conversations in developer forums, or recommendation of bug fixes by analyzing changes that fixed similar bug reports in the past.
Bibliography


