UNIVERSITY OF CALIFORNIA
RIVERSIDE

Quantitative Decision-making in Software Engineering

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Pamela Bhattacharya

June 2012

Dissertation Committee:

Professor Iulian Neamtiu, Chairperson
Professor Gianfranco Ciardo
Professor Michalis Faloutsos
Professor Rajiv Gupta

The Dissertation of Pamela Bhattacharya is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I first met my advisor Prof. Iulian Neamtiu on 28th July, 2009 with absolutely no hope that I will be able to continue my PhD. Today, when I write this dissertation, I cannot thank him enough for all his advice and mentoring. In fact, it would be an understatement to say Iulian introduced me to research in software engineering; he taught me everything about research: from choosing the right problems to solve to sharpening my writing and presentation skills. I appreciate his patience, encouragement and support through out these years that helped me to achieve much more than I thought I could. Most importantly, I learned the importance of perfection from Iulian, a quality I think, will continue to help me grow in my career. Thank you Iulian!

I collaborated closely with Prof. Michalis Faloutsos during last couple years of my PhD. Michalis has an innate ability to recognize very quickly a student's aptitude, interests, potential, and then to encourage them in exactly the right way. On the technical side, I learnt from Michalis the importance of abstraction. Michalis can beautifully abstract lower-level details and help focus on high-level "stuff" (the same reason he can write killer abstracts in a paper). Even though he claims he is mostly a network-science researcher, I would never know he is not a "software-engineering guy" (yeah, that's what he calls us) unless he declared so because he can magically abstract software engineering details to understand the "grand vision." Michalis often mentored me on non-technical issues, and his motivation-encouragement has helped me immensely to improve my confidence levels. Thank you Michalis.

My other committee members Prof. Rajiv Gupta and Prof. Gianfranco Ciardo

have been supportive of my work through these years. Prof. Gupta helped me understand the importance of finding a research topic I am passionate about early on in my PhD life and I will be indebted to him forever for that priceless advice. My sincere thanks to Prof. Christian R. Shelton and Prof. Eamonn Keogh for always being available to answer my questions on data mining and machine learning.

I would like to thank my co-authors Prof. Christian Shelton and Dr. Marios Illiofotou for their advice, opinions, and guidance while collaborating on projects that contributed to this dissertation. Many software developers who worked on the several open source projects often helped me in data collection and answering our questions that formed an integral part of our work. All their help are gratefully acknowledged; special thanks to Gerv from Firefox and Chris from Eclipse. My sincere thanks to Prof. Monica Neamtiu, Dr. Arnab Mitra and Rajdeep Sau for all the help with statistical analysis that forms an important part of this dissertation. Thanks to Lucy and Sai for collaborating on the Android-project. Most of the work presented in this dissertation have been published in premier software engineering conference proceedings; I would like to take this opportunity to thank all the anonymous reviewers for their comments on the early versions of the respective chapters.

On an organizational level, I would like to thank the Department of Computer Science, and University of California, Riverside for providing an excellent environment for a student 8154 miles away from home. Thank you Amy, Sandra, Maddie, Alicia, Tatev, Lanisa, Issac, Kara, Sahar and Vanoohi for all the help.

I would also like to take this opportunity to thank my first academic advisor, Prof.

Gerald Baumgartner, who stood by my decision of changing schools in the middle of my PhD under his supervision. Special thanks to Prof. Sukhamay Kundu who guided me in graduate school admissions.

I was lucky to have two inpsiring mentors, Prof. Krishnan Venkatachalam and Dr. Amitabha Chanda during my undergraduate days who encouraged me to pursue graduate school. I was blessed to have been associated with several other teachers and mentors (special mentions: Sanat Roy Choudhury, Bijon Jethu, and Dr. Prasanta Mukherjee) all throughout my life who helped me value education. I would like to thank Computing Research Association for Women for letting me be a part of their mentoring sessions that gave me the opportunity to network with leading female researchers and other fellow graduate students. If not for the generous fellowships by NSF and ACM-SIG grants, attending conferences at luxurious venues, being able to present our work to the broader software engineering community and networking with other researchers would have been impossible. I would also like to sincerely acknowledge the support of several grants that funded my graduate education: CCF-1149632, CNS-1064646, and UCR Senate award.

Riverside is where I spend the longest time of my life after leaving home for the first time. While I am happy that I will be finally graduating, I will dearly miss so many people. To start with, Amy Ricks (graduate students affairs officer) was the first person I met after I arrived at UCR. Since then Amy has been a true friend in all respects. She was kind enough to lend sound advice to me about matters that has nothing to do with her job description. Thank you Amy! I had countless friends and well-wishers at UCR. Changhui, Dennis, Suchismita di and Smruti were the best friends ever. They helped me in thousands

of ways and stood by me in times of great hardships. I will forever cherish the memorable times with my lab mates: Xiaoqing, Masoud, Steve, Reaz, Tony, Dorian, Curtis, Lucy and Amlan. A big shout-out to my classmates with whom I spend hours understanding homework problems and preparing for exams: Reaz, Hassan, Farhan, Keerthi, Busra and Marios. Roger, Mueen, Doruk, Jianxia, Inder, Shail, and Olga were fellow grad students who I usually met on corridors and had the pleasure of having the long enlightening conversations (research or otherwise). Special thanks to Sumit for lending a virtual shoulder to cry on; if not for him, keeping the humor alive in the dark days of grad school would have been impossible.

My family never heard of GRE and neither did they have any clue which part of the world Riverside is located in six years ago. My parents, uncle-aunt, grandparents and my three wonderful sisters deserve much credit for whatever I have achieved in life so far. They provided the most intellectually nurturing environment that a child could ask for and moved heaven and earth to help me pursue my dreams. Most importantly, my grandma, mom and aunt are three women who have been homemakers all their life, lived in a society (or rather part of India) which opposes professional growth of women and yet relentlessly encouraged all the four daughters to pursue higher education and have a professional career. I love you guys!

If not for Sudipto, I would have quit PhD long back. Apart from all the emotional support he has given me over the years, he helped me realize the importance of sincerity and dedication towards one's work without focusing on the end-results - a much needed realization that helped me be the optimistic individual I am today. Moreover, from his

work ethic, I have learned that a task worth pursuing is a task worth doing well; even if that means spending 16 to 18 hours a day on it for months and years.

Lastly, this dissertation would not have been possible without the help and good wishes of several people. It is impossible to acknowledge everyone here; if I have missed you, I apologize for the mistake but want you to know that you are gratefully remembered.

To my family.

Ma, Baba, Mamon, Taun, Shonai, Dadu, Pom-Dona-Tuklu, Sudipto

ABSTRACT OF THE DISSERTATION

Quantitative Decision-making in Software Engineering

by

Pamela Bhattacharya

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2012
Professor Iulian Neamtiu, Chairperson

Our thesis is that *software repositories contain latent information that can be mined to enable quantitative decision making.* The decision-making process in software development and maintenance is mostly dependent on software practitioner's experience and intuition. For example, developers use their experiences when prioritizing bug fixes, managers allocate development and testing resources based on their intuition and so on. Often these human driven decisions lead to wasted resources and increased cost of building and maintaining large complex software systems. The fundamental problem that motivates this dissertation is the lack of techniques that can automate decision-making process in software engineering. As data mining techniques became more mature, mining software repositories has emerged as a novel methodology to analyze the massive amounts of data created during software development process. Significant, repeatable patterns and behaviours in software development can be identified as a result of this mining which are often used for predicting various aspects of software development and maintenance, such as predicting defect-prone releases, software quality, or bug fix time. In this dissertation we show tech-

niques to effectively mine software repositories, identify significant patterns during software development and maintenance, and recommend actionable aspects that can automate the decision-making process in software engineering. We demonstrate efficient techniques to use the information stored in software repositories and produce results to guide software practitioners so that they can depend less on their intuition and experience and more on actual data.

To this end, in this dissertation, we make several contributions. First, we perform several empirical studies to characterize information needs of software developers and managers in the context of decision making during software development and maintenance. For example, we study what kinds of decision-making problems are important to software practitioners on a daily basis. Second, to facilitate analysis of various types of decision-making problems using a common platform, we design a generic mixed-graph model to capture associations of different software elements. We illustrate how we can build different types of hyper-edges on this mixed-graph to quantify amorphous behaviour and dependencies among various software elements. Third, to demonstrate the effectiveness of our framework, we formalize a set of four important decision-making problems that are challenging to address with the state-of-the-art. We show that our framework can achieve high-levels of prediction accuracies for different types of decision-making problems when tested on large, widely-used, real-world, long-lived software projects.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Software systems are continuously changing and adapting to meet the needs of their users and therefore software development has high associated costs and effort. A survey by the National Institute of Standards and Technology estimated that the annual cost of software bugs is about $59.5 billion [125]. Some software maintenance studies indicate that maintenance costs are at least 50%, and sometimes more than 90%, of the total costs associated with a software product [87, 146], while other estimates place maintenance costs at several times the cost of the initial software version [151]. These surveys suggest that taking the right decision to change any software artifact would be beneficial in reducing the high-costs associated with the software development and maintenance process. Decision-making and strategic-planning are two key attributes in software engineering; everyday software practitioners have to deal with several important decisions. For example, which

parts of the code must be refactored, which parts of the code should be tested with high priority, which parts of the code have high change-risks, which bugs should be prioritized, who can fix a bug, which feature requests should be addressed in the upcoming release, how can developers–testers collaborate more effectively, etc. Human-driven decision-making is a cognitive phenomenon which at an abstract level involves four primary sub-processes [79]: identifying probable solutions to the same problem, analyzing the pros and cons of each of them, prioritizing them based on their effects, and then choosing the solution which intuitively seems to be the best. To ensure that the right decision is being taken, managers need to go through a long, tedious process; they need to manually analyze probable options, always be up-to-date with pertinent information, carefully make long-term and short-term plans and then use their expertise and instincts to make a decision. However, ensuring correctness of these plethora of decisions becomes extremely challenging and error-prone as the size of the software grows both in code size as well as the number of contributors (developers, testers, and managers) working collaboratively. This process is further complicated by the pressure of delivering products that will accommodate the end-user's needs and increase the wide-spread use of the software. These manually taken decisions therefore often lead to wasted resources and increased cost of building and maintaining large complex software systems [53, 110, 72].

Examples. We provide several examples of regularly encountered decision-making problems and why manually-taking decisions for these problems is hard and error-prone.

1. When a bug report is submitted by an user, the status of the bug is *Unconfirmed* by default. After the module owner or the triager of the module the bug has been reported

in can confirm that the bug report is valid, the status of the bug is changed to *New*. After a bug is changed to status *New*, the triager needs to first assign severity and priority to the bug before the bug can be assigned to developers. Assigning severity-priority to bugs manually is a non-trivial task as severity is measured by how badly the software crashes due to the bug while priority measures how important it is to spend the effort in fixing the bug. As of May 2012, the Firefox bug repository contains 14,992 unconfirmed and new bugs. Validating the unconfirmed bugs, assigning severities-priorities manually is a time-consuming and error-prone manual task.

2. After a bug has been confirmed as a valid bug, the bug triager needs to assign manually the bug to a developer who can potentially fix the bug. An empirical study by Jeong et al [76] reports that, on average, the Eclipse project takes about 40 days to assign a bug to the first developer, and then it takes an additional 100 days or more to reassign the bug to the second developer. Similarly, in the Mozilla project, on average, it takes 180 days for the first assignment and then an additional 250 days if the first assigned developer is unable to fix it. These numbers indicate that the lack of effective, automatic assignment and toss reduction techniques results in considerably high effort associated with bug resolution.

3. Open source projects critically depend on external contributions and volunteers who support the community in various ways in addition to fixing bugs and adding code. Lack of automatic tools to manage the source-code and bug-repositories often leads to frustrations among these volunteers. For example, after an individual submits a patch which has passed the review, he/she needs to obtain commit access to the

repository. This commit access application is manual and requires a voting system where multiple senior project-leaders need to unanimously grant access. [1] These applications often take months to grant access leading to committer frustration (e.g., Mozilla bug 609552). [2] Also, inefficient triaging techniques can lead to frustration to the extent where contributors leave the project. [3]

The above examples show that as software grows and the development-maintenance process becomes substantially complex, it is necessary to develop automated techniques that can assist several classes of software practitioners (developers, testers and managers) with *routine* decision-making problems (similar to those instantiated earlier) and help them compare and contrast between multiple probable solutions. Prior works in mining software repositories have concentrated on understanding how software systems evolve [161, 64, 171], how can significant repeatable patterns during software development be used for efficient software quality predictors [175, 144, 122] and how a social network of developers contribute towards community growth in software development [23, 21]. However, none of these studies integrate all the three aspects of software development (i.e., system evolution, quality prediction, and social interactions) together to help automate decision-making in software engineering.

This dissertation builds the missing bridge across these multiple facets of mining software repository. We identify various types of decision making problems in software engineering, explain why automating resolutions to these human-driven decisions are hard, or

[1] Mozilla commit access policies: `http://www.mozilla.org/hacking/commit-access-policy/`
[2] `https://bugzilla.mozilla.org/show_bug.cgi?id=609552)`
[3] http://tylerdowner.wordpress.com/2011/08/27/some-clarification-and-musings/

error-prone, and design efficient data-driven recommendation frameworks that can facilitate the conventional decision-making process in software engineering. We employ large-sized and widely-used open source projects for conducting empirical studies to identify common decision-making problems as well as to validate our techniques. The lack of organized data in open source software further complicates the problem of finding automatic resolutions to human-driven decision making in these projects.

1.2 Challenges

To enable efficient mining and ensuring the results of our empirical studies represent real-world scenarios, gathering clean and structured data from several large, long-lived software repositories was the first challenge we faced during the course of this work. The lack of standard benchmarks to validate the effectiveness of data-driven recommendation or predictions models in empirical software engineering further complicates the problem of constructing data sets that are context-sensitive and of statistically significant size. In this section, we enumerate the additional challenges in mining software repositories to automate decision making.

- *Characterizing information needs of software practitioners*

  The first challenge is to identify a set of routine decisions taken during software development process which are prone to error if taken manually and can be improved using field-data.

- *Amorphous nature of software repositories*

The second challenge in drawing meaningful results by mining large software repositories is the incomplete nature and unstructured format of the information contained in these repositories.

We provide several examples that will help in understanding the problems associated with amorphous nature of the software repositories:

- Software design specifications: open source projects rarely have software design descriptions that record all the design specifications for major releases. Release notes typically contain an itemized set of features and enhancements available with that release and set of blocker bugs from previous release that have been fixed. Unavailability of these specifications makes it difficult to reason about the evolution of the software project.
- Classifying source code changes: source code change logs rarely contain meta data that can help infer if the change was due to a feature enhancement or a bug fix.
- Major architectural changes: code analysis for a project might suggest that significant portions of the source code were rewritten for a specific version. Reasons behind even such major development decisions are never documented in open source projects. For example, Apache Lucene is an information retrieval software library and while mining the release logs we found that version 3.0 of the software was rewritten in Java 5 while all versions for 1.x and 2.x are written entirely in Java 1.4. Detecting factors that affected the decision of rewriting ma-

jor parts of the source code is an useful piece of information that can influence future decisions. Next generation of project managers might ask, "why was this change made at all?," "what features of Java 5 influenced this major change?," or "did this change result in improved user-experience?," or "did this major change introduce more bugs in the code?"

The scenarios presented above demonstrate the need for analyzing software changes. Attaching meaningful information to these various facets of the evolution process by extracting data from these large repositories is therefore the first challenge.

- *Integrating multiple repositories of the same software project*

  Prior works in mining software repositories have looked at one repository from a software project at a time and tried to deduce significant patterns in software development and evolution. However, we observed that multiple repositories of a same software project are deeply interconnected. For example, it is often required to merge commit messages from source code repositories and log messages from bug databases to track all changes made to the code to fix a bug. However, it is not always possible to infer these information from the log message or the commit message alone. Therefore, the third challenge is to combine amorphous meaningful information from multiple repositories of the same project and model heterogeneous relationships among them.

- *Building frameworks to automate decision-making process in software engineering*

  With information about various relationships within software artifacts and other relevant empirical analyses in hand, the over-arching goal of this dissertation is to build

automated frameworks to assist software practitioners in decision making. The fourth challenge is therefore to build scalable, self-evolving, efficient and automated recommendation systems with minimum false positives that can help compare–contrast multiple resolutions to a given decision-making problem. We then plan to validate our frameworks on statistically significant data sets and show how our methods outperform manual approaches.

1.3 Dissertation Overview

The overarching goal of this dissertation is to effectively mine software repositories to identify significant patterns during software development and maintenance, and produce information that can automate decision-making process in software engineering. We show how using the information stored in software repositories we can help generate recommendations to guide software practitioners so that they can depend less on their intuition and experience and more on historical and field data. This dissertation is guided by the following thesis:

Software repositories contain latent information that can be mined to enable quantitative decision making.

Using this principle as the cornerstone, this dissertation makes three major contributions. First, we design a generic data-driven framework for both generating recommendations in the context of decision-making problems and for searching software repositories effectively. Second, we identify and formalize four novel decision-making problems that can

benefit from the data-driven recommendation system we design. Then, we show the effectiveness of our recommendations by evaluating our techniques on large, real-world data sets. Third, we show how the inherent search-based property of our framework can be further improved when using a Prolog-based query-management system.

1.3.1 Overview of the Underlying Framework

Software repositories, namely source code and bug repositories contain a wealth of data. The central idea of this dissertation is to use these data in facilitating various decision-making process in software engineering. However, before this data can be used for forming recommendation engines, it needs to be collected, cleaned and eventually structured in a format that can be used in designing our recommendation and search frameworks. We first illustrate how and what raw data we collect and what other information we extract from this raw data. Next, we describe several metrics we design and networks we build to characterize relationships between several entities. Then, we show how these metrics–dependencies can be utilized to form both recommendation-based and search-based query framework to benefit quantitative decision-making in software engineering.

1.3.2 Recommendations to Facilitate Decision-making

To show the effectiveness of our generic data-driven framework in providing recommendations, we apply our framework to four novel decision-making problems. We show an overview of this process in Figure 1.1. The repositories already contain the entire data and relationships between several entities. For each of the problems, we first extract problem-

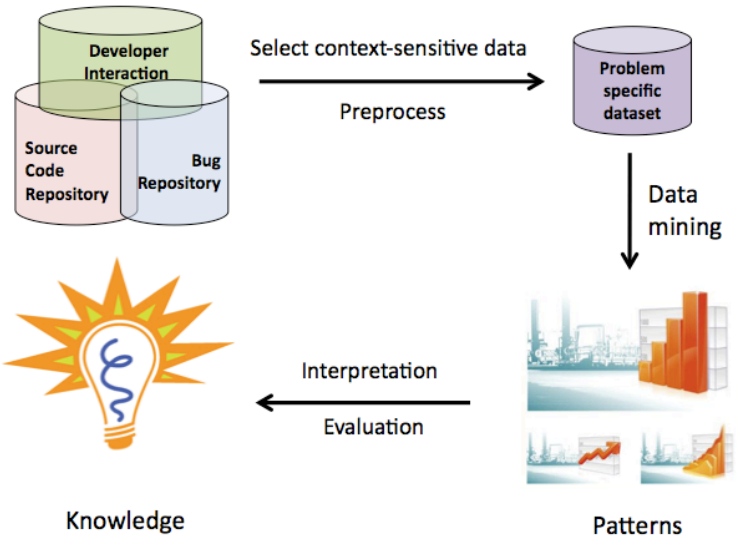


Figure 1.1: Overview of the underlying generic framework.

specific data. Next, we perform several empirical studies to understand what factors affect the problem, what attributes can be used to model the problem and then we design effective metrics to provide recommendations.

Sample Decisions

Empowered by several empirical studies, we identified the following four routine problems in decision making that would benefit highly from automated data driven recommendations.

1. *Choosing the right programming language*

   A programming language is one of the tools involved in developing software. Therefore, ideally, the choice of language should be made based on overall software requirements. However, in the real world, the choice of programming language is dominated

by two factors: (1) expertise of developers who will be working on the project, (2) managerial decisions driven by experience, or legacy reasons, or intuition [30]. Another real-world scenario we often encounter is changing the primary language of the code base. This decision can however be influenced by several factors: (1) initial reasons for the language choice changes due to change in software specifications, (2) characteristics of the programming language that affect development, e.g., performance issues, (3) developer expertise has changed, etc. Therefore the choice of programming language (either before the project has started or after a significant portion of the code base has been developed in another language) in software development puts us in a position to ask how can we measure the benefit of choosing one language over another and thus enable software practitioners to take wiser decisions based on field-data rather than intuitions and experience. To this end, we model the problem of programming language choice based on its effects on software quality and maintenance. We hypothesize that the underlying programming language affects software quality and hence, the choice should be made carefully.

2. *Finding the right developer to fix a bug*

   Software bugs are inevitable and bug fixing is a difficult, expensive, and lengthy process. One of the primary reasons why bug fixing takes so long is due to the difficulty of accurately assigning a bug to the most competent developer for that bug type. Assigning a bug to a potential developer, also known as bug triaging, is a labor-intensive, time-consuming and fault prone process if done manually. Moreover, bugs frequently get re-assigned to multiple developers before they are resolved, a process

known as bug tossing. We show that machine learning and probabilistic graph based recommendation models can efficiently automate the bug triaging process.

3. *Predicting effects of code changes and team reorganizations on software quality*

   Changes in software development are multi-faceted: changes in software requirements, developer expertise, underlying programming language, source code, etc. are crucial to allow continued development of a software project. To effectively manage change, developers and managers must assess the effects involved in making a change. Therefore to enable practitioners understand the potential threats associated with change, it is important quantify and estimate several facets of software changes that can help prioritize debugging efforts and fore-warn about defect-prone releases.

4. *Quantifying roles and expertise in open source projects*

   The inherent nature of open source software development enables volunteers to help in multiple facets of software development. This approach is significantly different from commercial software development where each individual has a defined role in the project. On one hand, the flexibility of working-on-what-you-like mindset in OSS (i.e., an individual can decide his contribution) has led to widely-used, popular projects (e.g., Mozilla, Eclipse, Apache). On the other hand, this freedom has made human resource allocation and management challenging with increase in software size and individuals contributing to the project [117].

1.3.3 Searching Across Repositories

Our search-based framework is designed to serve two purposes: providing recommendations and answering search-based queries. Search-based queries can be used both by software practitioners and empirical software engineering researchers. In the last part of this dissertation, we show that the framework while allowing efficient search and analysis on software evolution data, has two main inconveniences: (1) it is not flexible enough, e.g., it can only permit a limited range of queries, or have fixed search templates; (2) it is not powerful enough, e.g., it does not allow recursive queries, or do not support negation; however, these features are essential for a wide range of search and analysis tasks. We argue the need for a framework built with recursively enumerable languages, that can answer temporal queries, and supports negation and recursion. To this end, as a first step toward such a framework, we present a Prolog-based system that we built, along with an evaluation of real-world integrated data from the Firefox project. Our system allows for elegant and concise, yet powerful queries, and can be used by developers and researchers for frequent development and empirical analysis tasks.

1.4 Contributions

This dissertation makes the following contributions that significantly advance the state-of-the-art in mining software repositories to automate the decision making process in software development.

1. We design a generic data-driven framework by exploiting relationships within and

across multiple software repositories that can be used both for generating recommendations for decision-making problems and for searching across repositories,

2. To demonstrate the effectiveness of our recommendation-property of the framework, we identify four routine and important decision-making problems in software development that could benefit from data-driven recommendations. In particular, we show how we can automate the bug assignment process with high precision, help practitioners differentiate between effects of programming language on software quality, identify pivotal moments during software evolution using graph-mining and predict the role of a contributor plays in the project with high accuracy. We applied and demonstrated the validity of our recommendations framework on large, widely-used, long-lived, real-world software projects.

3. Using a Prolog-driven query management system in addition to our generic framework, we show how we can further improve searching in software repositories.

1.5 Organization

This dissertation is organized as follows: in the first part, we provide an overview of the generic data-driven framework we build (Chapter 2). In the second part (Chapters 3– 6), we show how we use this framework for providing recommendations for four novel decision-making problems. In the third part, we show how we use our framework for building an effective search engine (Chapter 7). In Chapter 8 we discuss related work and describe our experiences-lessons learned while carrying out this work and outline future directions

in Chapter 9.

Chapter 2

Framework Overview

In this chapter we describe the data extraction process and the skeleton of our data mining–query framework. Software repositories —source code, bug, and developer repositories— contain a wealth of data. The central idea of this dissertation is to accumulate data from these multiple sources, discover relationships among various components and help guide the decision-making process in software engineering. However, before these gigabytes of data can be used for constructing recommendation engines, the data needs to be collected, cleaned and put in an appropriate format. This chapter describes in two parts parts how we build and use this generic data-driven recommendation framework. To give a quick intuition, in Figure 2.1, we illustrate abstractly the various repositories we consider, and provide a glimpse of various inter- and intra- relationships we analyze. In our study, we consider all three software repositories: source code repository, bug repository and developer repository. As shown in the Figure 2.1, all these repositories are inter-dependent: a change in one repository induces a change in the other. In this chapter, we describe these inter-

and intra- dependencies between various elements that form these repositories and how we can formalize these relationships to build the underlying generic framework. First, in Section 2.1 we describe the raw data we collect from these multiple software repositories and various kinds of information we extract from it. Second, in Section 2.2, we describe a novel *mixed-multi graph model* that evolves from our framework (as shown in Figure 2.1) and how we can use the concept of *hyperedges* to further understand inter-and intra-dependencies among multiple software components. Third, in Section 2.3, using concrete examples we illustrate the effectiveness of our framework in answering decision-making or search-based querying in software engineering.

2.1 Populating the Framework Databases

2.1.1 Raw Data

Source Code Data

The source code of an open source software project is usually stored using standard open-source revision control systems such as Concurrent Versioning System (CVS), Apache Subversion (SVN) or Git. The source code of each release is available for check out from these repositories. After we have checked-out the code for each publicly available release for each project, we extract the logs of each of the files for a specific release. The log of each source code file returns the list of changes made to the file. In Figure 2.6 we provide a snippet of a log of a file from Mozilla Firefox. We collect the following data from log files:

1. *File name*: we collect the path of each file denoted as RCS in the log file.

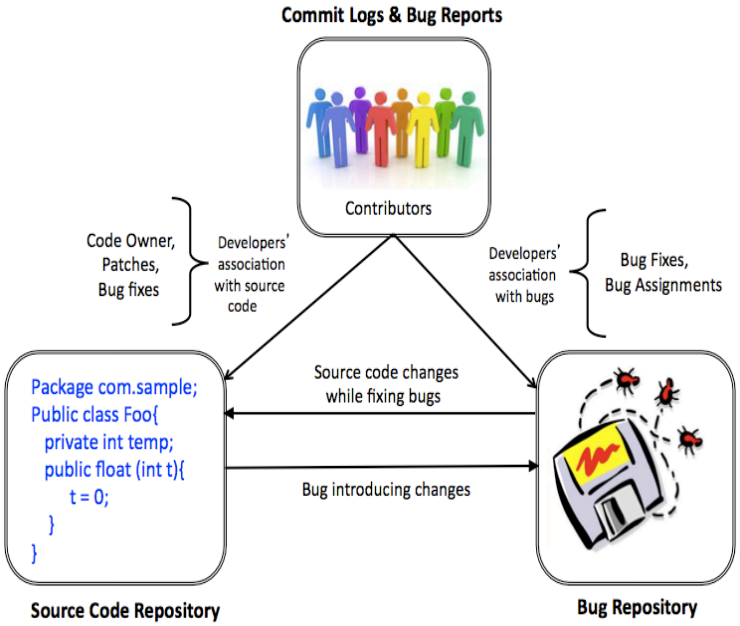


Figure 2.1: Overview of inter-repository dependencies.

2. *Committer ID*: we collect the email address of the developer who committed the code.

3. *Temporal information*: for each commit, we extract the date and time of the commit.

4. *Number of lines changed*: we extract the number of lines added and deleted for each commit to the file.

5. *Commit message*: each commit contains a brief description of the commit. For example, it might either be the record of the Bug ID which this patch is targeted for fixing, or the new feature that is being added with this commit.

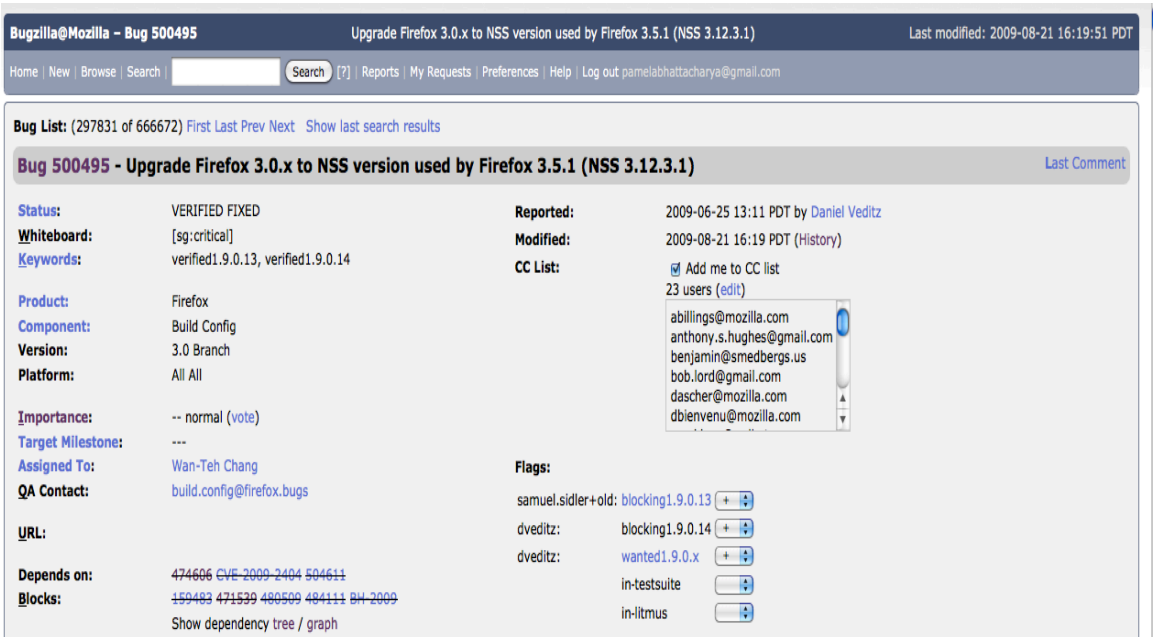


Figure 2.2: Bug report header information (sample bug ID 500495 in Mozilla).

Bug Data

Bug databases archive all bug reports and feature enhancement requests for a project. In Figures 2.2– 2.5, we show parts of sample bug report from Mozilla and activity related to it. We collect the following data from bug reports:

1. *Title and description*: the individual who reports a bug submits a title and description of the bug. The description often contains the series of steps to reproduce the bug.

2. *Name or ID* of the bug reporter.

3. *Temporal information*: date and time the bug was reported.

4. *Severity*: when a bug is reported, the administrators first review it and then assign it a severity rank based on how severely it affects the program. Table 5.2 shows levels

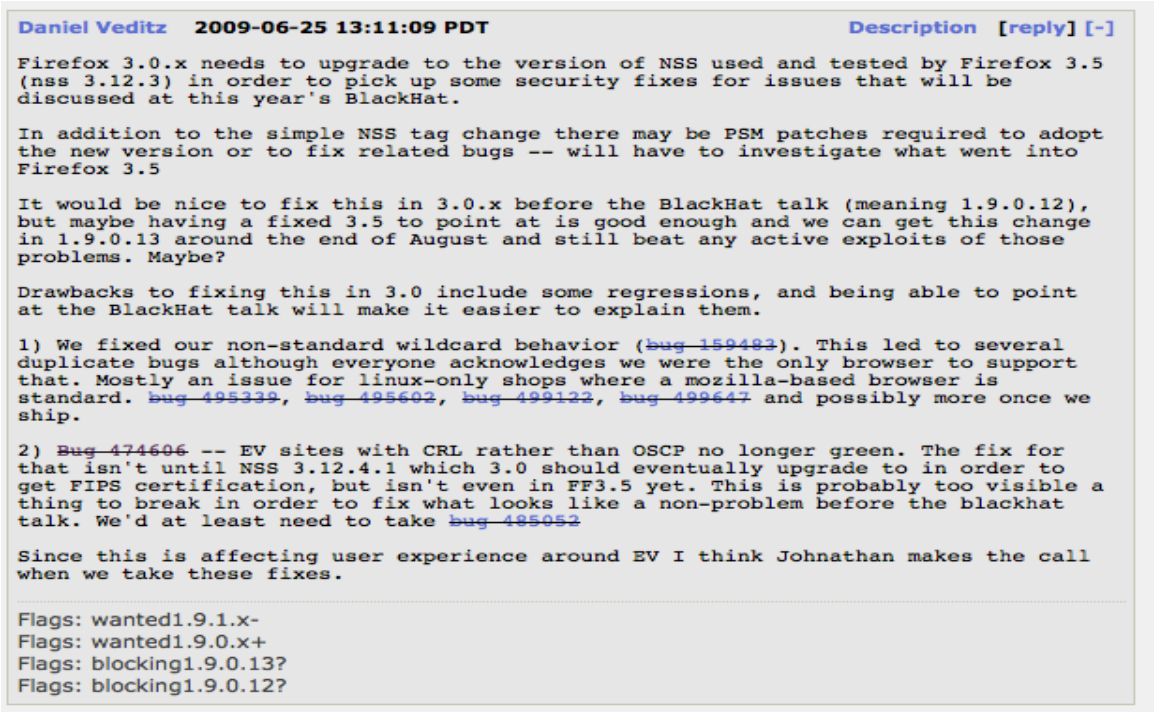


Figure 2.3: Bug description (sample bug ID 500495 in Mozilla).

of bug severity and their ranks in the Bugzilla bug tracking system.

5. *Priority*: a bug's priority rates the urgency and importance of fixing the bug, relative to the stated project goals and priorities.

6. *OS or platform*: a bug for the same software might be only present in one OS and not in others. Hence, the bug report contains either the specific OS the bug was found in or if it occurs across all platforms, this attribute is reported as *All*.

7. *Dependencies*: a bug X can be dependent on another bug Y when X is manifested only when bug Y occurs.

8. *Product and Component*: large projects like Mozilla and Eclipse contain subset of

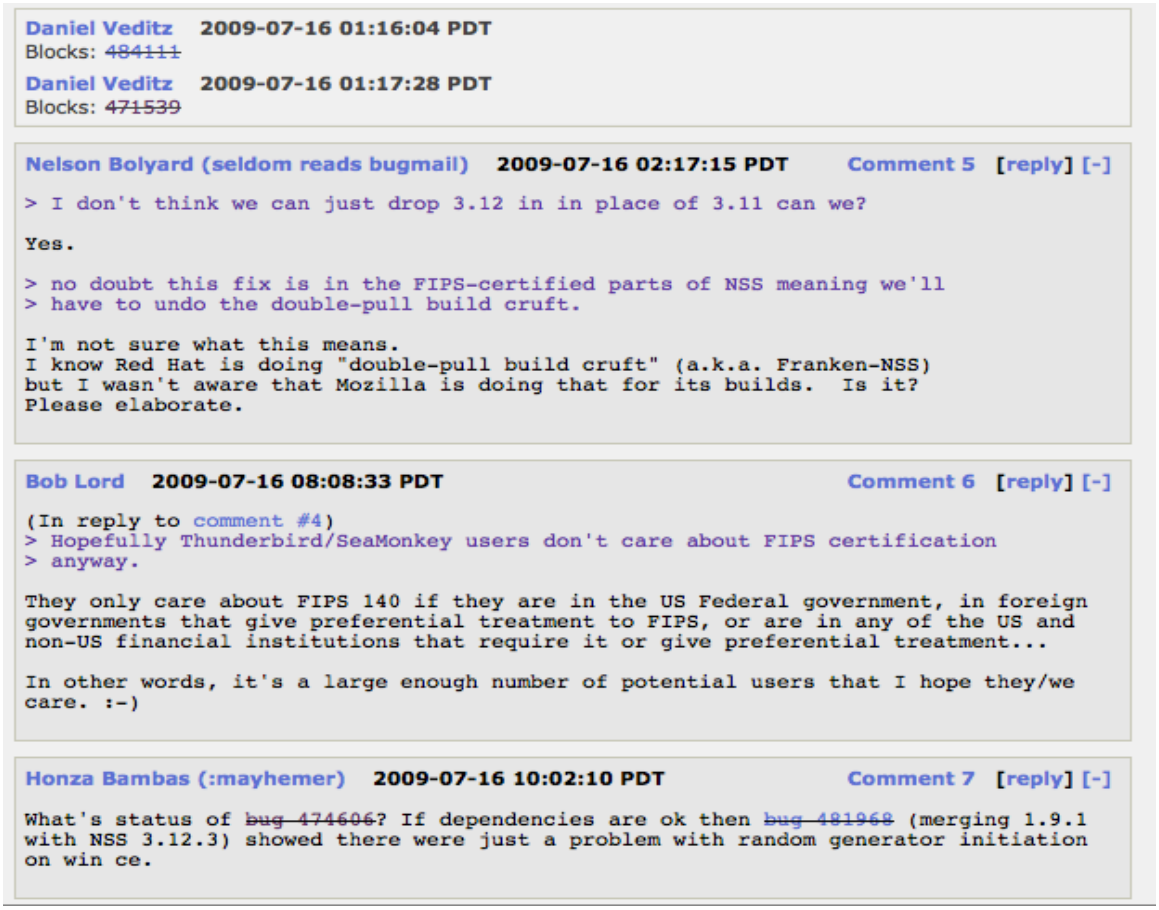


Figure 2.4: Comments for a bug report (sample bug ID 500495 in Mozilla).

products and components. For example, a bug in Mozilla might either belong to product–component Firefox–Security or Thunderbird–GUI.

9. *Set of comments*: several contributors and contributors discuss about a bug and the bug report archives all these discussions in the comment section. For each comment, we retrieve the contributor who commented, the temporal information, and his/her comments.

10. *Contributor activity*: we extract the list of contributors who were related to the bug,

Back to bug 50049

Who	When	What	Removed	Added
momoi	2000-08-23 15:21:06 PDT	CC		jaimejr
andreas.koenig	2000-08-25 10:17:17 PDT	CC		andreas.koenig
ftang	2000-08-28 11:55:03 PDT	Status	NEW	ASSIGNED
teruko	2000-12-14 15:42:49 PST	Keywords		intl
bobj	2001-01-05 13:50:18 PST	CC	bobj	
ftang	2001-01-16 17:52:12 PST	Assignee	ftang	nhotta
		Status	ASSIGNED	NEW
		Keywords		nsbeta1
		Target Milestone	---	mozilla0.9
momoi	2001-01-17 02:06:27 PST	CC		teruko
		Status	NEW	RESOLVED
		QA Contact	teruko	ylong
		Resolution	---	WORKSFORME
amyy	2001-04-03 17:10:07 PDT	Status	RESOLVED	VERIFIED

Figure 2.5: Bug activity (sample bug ID 50049 in Mozilla).

for example, who was assigned the bug, who fixed it, who changed the status of the bug etc.

2.1.2 Data Extraction

In this section, we describe the additional information we compute about the source code, bugs in the code and the contributors involved in a project using the raw data that is extracted from various software repositories as explained in Section 2.1.1.

Source Code Data

We compute the following types of information about source code files:

```
RCS file:
/cvsroot/mozilla/browser/components/migration/src/ns
CaminoProfileMigrator.cpp,v
Working file:
browser/components/migration/src/nsCaminoProfileMigr
ator.cpp
total revisions: 14;   selected revisions: 14

description:
----------------------------
revision 1.12
date: 2006/11/13 17:53:00;  author:
benjamin%smedbergs.us;  state: Exp;  lines: +1 -0
Re-land bug 345517 now that the logging issues are
hopefully fixed, r=darin/mento/mano
----------------------------
revision 1.11
date: 2006/11/10 04:42:01;  author:
pavlov%pavlov.net;  state: Exp;  lines: +0 -1
backing out 345517 due to leak test bustage
----------------------------
revision 1.10
date: 2006/11/09 15:02:27;  author:
benjamin%smedbergs.us;  state: Exp;  lines: +1 -0
Bug 345517, try #2, make the browser component use
frozen linkage, so that ff+xr builds. This does
*not* --enable-libxul by default for Firefox (yet).
That will wait until after 1.9a1. Older patch
r=darin+mento, revisions r=mano
----------------------------
revision 1.9
date: 2006/08/10 14:06:46;  author:
benjamin%smedbergs.us;  state: Exp;  lines: +0 -1
Backout bug 345517 due to various issues.
----------------------------
```

Figure 2.6: Example log file from the Firefox source code.

- *Effective Lines of Code (eLOC)*: we compute the effective lines of code for each source code file. eLOC is the measurement of all lines that are not comments, blanks or standalone braces or parenthesis in a source code file.

- *Cyclomatic Complexity:* cyclomatic complexity (also known as McCabe's Complexity) of a program is defined as the number of linearly independent paths through the source code [105]. For instance, if a source code file contains no decision points such as *IF*

Bug Severity	Description	Rank
Blocker	Blocks development testing work	6
Critical	Crashes, loss of data, severe memory leak	5
Major	Major loss of function	4
Normal	Regular issue, some loss of functionality	3
Minor	Minor loss of function	2
Trivial	Cosmetic problem	1
Enhancement	Request for enhancement	0

Table 2.1: Bug severity: descriptions and ranks.

statements or $FOR-NEXT$ loops, the complexity would be 1, since there is only a single path through the code. On the other hand, if the code has a single IF statement containing a single condition there would be two paths through the code, one path where the IF statement is evaluated as $TRUE$ and one path where the IF statement is evaluated as $FALSE$.

- *Interface Complexity:* interface complexity is computed as the number of parameters and the number of return points for a function.

- *Defect Density:* defect density is calculated as the number of bugs per line of code for a file.

- *Maintenance Effort:* we compute maintenance effort as the number of commits required per line of code in a program.

- *Bug Severity:* we compute the median bug severity of each function and module. To compute this metric, we collect the severity of bugs whose fixing led to a change in

the function or a module.

- *Contributor Profile:* we create contributor profiles based on their contribution to the source code of a project: both for adding new source code to the project and in the event of fixing bugs. We collect the following four types of information to create a contributor profile:

    1. *Contributor ID:* a contributor to a software component is someone who has made commits/software changes to the component. We extract the commit id based on the log entry to identify all contributors associated with a software project.
    2. *Seniority:* we define seniority as the time a contributor D has been associated with a project. Hence, seniority is computed as the difference in the number of years between the first and last commit found in the entire log history of a project by contributor D.
    3. *Number of lines of code added:* we count the total number of lines of code added by a contributor during his lifetime association with a project.
    4. *Number of bug-fix commits:* we count the number of source code changes that a contributor committed in the event of a bug-fix.
    5. *Number of files committed to:* we count the number of files a contributor D has committed to.
    6. *Ownership:* file ownership is computed as the proportion of commits a contributor has made relative to the total number of commits for that file. For example, if file abc.cpp has 10 commits, and contributor D has committed twice, his pro-

portion of ownership would be 20%.

7. *Language-based contribution:* we compute how many different file types (e.g., .c, .cpp, .java, .html, .pl, .py etc.) a contributor has committed to and the frequency of those contribution.

Bug Data

We compute the following types of information about source code files:

- *Bug-fix time:* we compute the time it took to fix a bug; i.e., the difference in the number of days from the date the bug was reported until it was fixed and closed.
- *Bug-tossing paths:* we collect the bug-tossing paths by referring to the bug activity section of a bug report. A bug-tossing path is defined as the sequence of contributors who are involved in fixing a bug.
- *Bug-fix based contribution profile:* similar to source code based contributor profile, we create bug-fix based contribution profiles of contributors in a project. We extract the following information creating these profiles:

  1. *Contributor ID:* the contributor ID is the login id an individual uses to participate in the bug reporting and fixing process.
  2. *Number of bugs assigned:* for each contributor, we count the number of bugs the person had been assigned during his lifetime.
  3. *Percentage of bug fixed:* for each contributor, we compute the proportion of bugs he was assigned that he could fix.

Source code		**Bugs**	**Contributors**	
Functions	*Modules*		*Source-code*	*Bug-based*
eLOC	eLOC	Title, Description	Seniority	Seniority
Interface Complexity	Cyclomatic Complexity	Dependencies	LOC added	Total number of bugs assigned
Defect Density	Defect Density	Severity, Priority	Ownership	Total number of bugs fixed
Maintenance Effort	Maintenance Effort	Comments	Number of bug-fix commits	Average bug severity
Bug Severity	Bug Severity	Status	Language-based contribution	Sub products-components contributed to

Table 2.2: Summary of raw and computed data from various repositories.

4. *Average bug severity:* when a bug is reported, the administrators first review it and then assign it a severity rank based on how severely it affects the program. We compute the weighted average severity of bugs that a contributor has worked on.

5. *Seniority:* for bug-fix induced seniority, we define seniority as the difference in time between the first and last time the member fixed a bug, as recorded in the bug tracker.

6. *Sub-Product-Component contribution*: large projects like Mozilla, Eclipse have sub-products and sub-components within each product. For example, Firefox, Thunderbird, SeaMonkey are three popular sub-products in Mozilla and each of these products have sub-component modules like GUI, Security, etc. Each contributor has a list of all products and components he/she has worked on during his association with the project and the frequency of each association.

To summarize, we provide a compact table (2.2) to show what different types of data we have related to various software artifacts. With both the raw data (i.e., information available directly from the repositories) and the **computed data** (i.e., data we compute using the raw data) in hand, we now move to show how the data collected and computed from multiple repositories are interconnected and dependent on each other.

2.2 Representing Software Repositories as a Mixed-multi Graph

In this section we explain how various relationships that exist between different components in different repositories — source-code, bugs and contributors — can be used to build a mixed-multi graph, as shown in Figure 2.1.

***Definition*(Mixed-multi graph).** $\mathbb{G}_{InterRepoDep} = (\mathbb{V}, \mathbb{E}, \mathbb{W})$ such that,

$\mathbb{V}$: set of vertices's such that a vertex v can denote a function call (v_{func}), or a module (v_{mod}), or a bug (v_{bugid}) or a contributor ($v_{contributor}$),

$\mathbb{E}$: set of directed and undirected edges between any two vertices's,

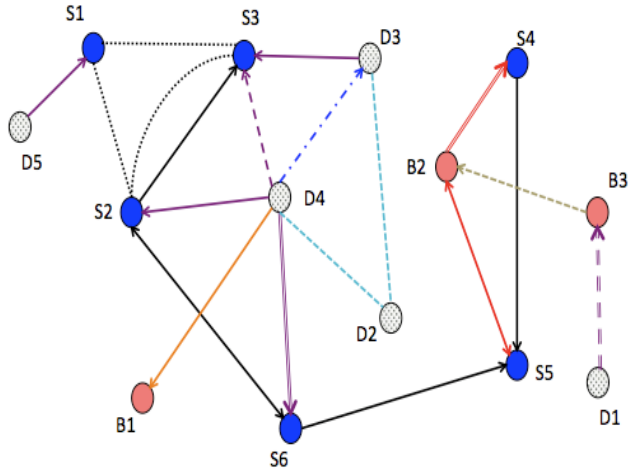
$\mathbb{W}$: weight of an edge W is optional and would refer to the strength of a connection between two vertices's depending on which two types of nodes the edge is connecting and the significance of the relationship.

Using data from multiple repositories as shown in Figure 2.1, [1] we create the mixed-multi graph that has the following properties: (1) contains nodes of three types, namely, source code elements (functions or modules), bugs and contributors, (2) contains directed edges that denote any dependency or direct relationship, and (3) undirected edges

[1] We describe this data collection-extraction process in Section 2.1.

Edge in the graph	Relationship
$S_4 \rightarrow S_5$ $S_2 \rightarrow S_3$ $S_6 \rightarrow S_5$	Source code dependencies (e.g., S_4 depends on S_5)
$S_2 \rightarrow S_6$	Source code mutual dependency (recursive-call)
$S_1 - S_2 - S_3$	Co-changed source code elements
$D_3 \rightarrow S_3$ $D_4 \rightarrow S_2$ $D_5 \rightarrow S_1$	Developer added code as enhancement (e.g., developer D_3 added code to source code file S_3)
$D_4 \rightarrow S_3$	Developer changed code for bug-fix
$D_4 \rightarrow S_6$	Developer changed code both during bug-fix and enhancement
$B_3 \rightarrow B_2$	Bug dependency
$D_4 \rightarrow B_1$	Developer D_4 fixed bug B_1
$D_1 \rightarrow B_3$	Developer D_1 was assigned bug B_3 (was unable to fix)
$B_2 \leftrightarrow S_5$	Bug from code and bug-fix changed file
$B_2 \rightarrow S_4$	Fixing bug B_2 changed file S_5
$D_4 \rightarrow D_3$	Tossing probability
$D_4 - D_2$ $D_2 - D_3$	Social interactions (e.g., commented on same bug, replied to same forum discussion topic)

(a) Examples of relationships between multiple repositories.



(b) Graph $\mathbb{G}_{InterRepoDep}$ generated using relationships in Table 2.7(a)

Figure 2.7: Example of a mixed-multi graph created from inter-repository dependencies.

which refer to indirect relationships. We demonstrate a sample mixed graph in our case in Figure 2.7(b) where blue nodes (labelled as S_x) refer to source code elements, dotted nodes (labelled as D_x) refer to contributors and pink nodes (labelled as B_x) refer to bugs. Various relationships that can exist among these nodes are instantiated in Table 2.7(a).

Next, we describe the notion of *hyperedges* to model and discover amorphous relationships between various software elements. Hypergraphs are a generalization of a graph, where an edge can connect any number of vertices. Formally, a hypergraph $\mathbb{H}$ is a pair $\mathbb{H} = (\mathbb{X}, \mathbb{E})$ where $\mathbb{X}$ is a set of elements, called nodes or vertices, and $\mathbb{E}$ is a set of non-empty subsets of $\mathbb{X}$ called hyperedges or links. In Figure 2.8 we show a hypergraph $\mathbb{G}$, such that $\mathbb{X} = \{A, B, C, D, E, F, G, H, I\}$ and $\mathbb{E} = \{e_1, e_2, e_3, e_4, e_5\} = \{\{B, C\}, \{A, B, C\}, \{D, E\}, \{C, F, G, H\}, \{I\}\}$. In our case, we construct hyperedges linking any combination of the different types of nodes: source code elements (functions or modules), contributors and bugs to establish relationships between several artifacts across software repositories.

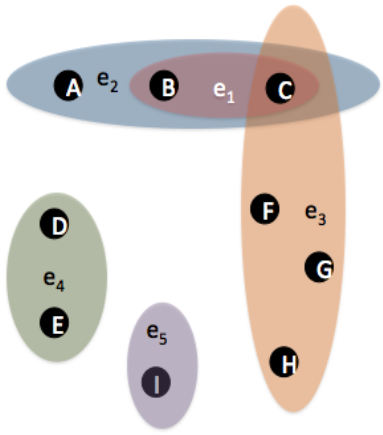


Figure 2.8: Example of hyperedges in a mixed graph.

As shown in Figure 2.7(b), the mixed-multi graphs are first created by using various information available from repositories. In Figure 2.7(b), blue nodes (labelled as S_x) refer to source code elements, green nodes (labelled as D_x) refer to contributors and pink nodes (labelled as B_x) refer to bugs. Various example relationships that can form edges between these nodes are instantiated in Table 2.7(a).

2.2.1 Intra-repository Dependencies

In this section we illustrate the networks that are formed within each software repository and how the attributes depend on each other within a repository. Further, we show how these networks can be deduced from mixed-multi graph $\mathbb{G}_{InterRepoDep}$ using the concept of hyperedges as described earlier in this section.

Source-code Repository

We construct two kinds of dependency graphs using the source code data:

1. *Function-call graphs*:

   ***Definition*(Function-call).** Directed graph $\mathbb{G}_{Func} = (\mathbb{V}, \mathbb{E})$ such that,

   $\mathbb{V}$: set of functions in a program which form the nodes in $\mathbb{G}_{Func}$,

   $\mathbb{E}$: set of edges such that e is an edge $v_1 \rightarrow v_2$ such that v_1 calls to v_2, and $v_1, v_2 \, \varepsilon \, \mathbb{V}$.

   We can deduce $\mathbb{G}_{Func}$ from $\mathbb{G}_{InterRepoDep}$ using the following relation:

$$\mathbb{G}_{Func} = \{(v_1, v_2) | (v_1, v_2) \ \in \ \mathbb{G}_{InterRepoDep} | v_{func} \ \wedge \ W(v_1 \rightarrow v_2) \ \geq \ 1\}$$

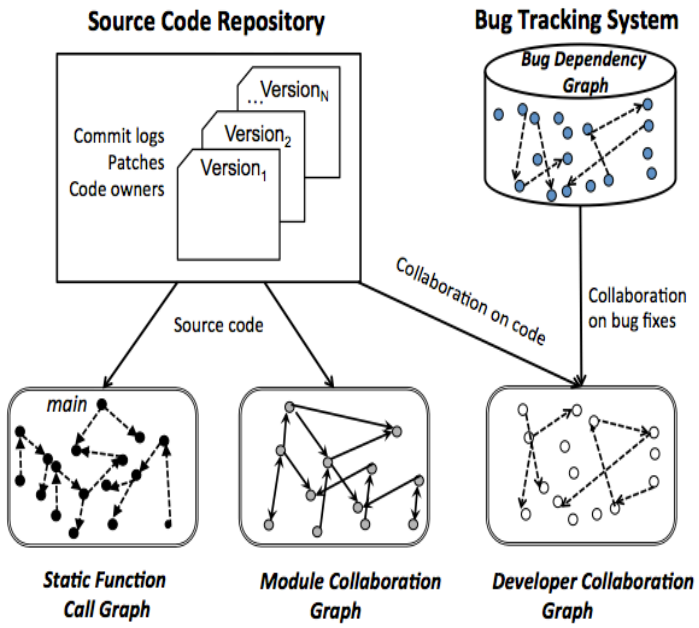


Figure 2.9: An overview of intra-repository dependencies.

The function call graph captures the static *caller-callee* relationship. If function A calls function B, the function call graph contains two nodes, A and B, and a directed edge from node A to node B. Our data set contains several applications written in a combination of C and C++; for virtual C++ methods, we add edges to soundly account for dynamic dispatch. Function call graphs are essential in program understanding and have been shown effective for recovering software architecture for large programs [27].

2. *Module collaboration graphs*:

***Definition*(Module-collaboration)**. Directed graph $\mathbb{G}_{Mod} = (\mathbb{V}, \mathbb{E})$ such that,

$\mathbb{V}$: set of modules in a program which form the nodes in $\mathbb{G}_{Mod}$,

$\mathbb{E}$: set of edges such that e is an edge $v_1 \rightarrow v_2$ such that at least one function in v_1 calls to another function in v_2, and $v_1, v_2 \in \mathbb{V}$.

We can deduce $\mathbb{G}_{Mod}$ from $\mathbb{G}_{InterRepoDep}$ using the following relation:

$$\mathbb{G}_{Mod} = \{(v_1, v_2) | (v_1, v_2) \ \in \ \mathbb{G}_{InterRepoDep} | v_{mod} \ \wedge \ W(v_1 \rightarrow v_2) \ \geq \ 1\}$$

This graph captures communication between modules, and is coarser-grained than the function call graph. We construct the module collaboration graph as follows: if a function in module A calls a function in module B, the graph contains a directed edge from A to B. Similarly, if a function in module A accesses a variable defined in module B, we add an edge from A to B. Module collaboration graphs help us understand how software components communicate.

Bug Repository

We construct the bug dependency graphs as described next using the data from bug repositories:

***Definition*(Bug-dependency)**. Mixed graph $\mathbb{G}_{BugDep} = (\mathbb{V}, \mathbb{E})$ such that,

$\mathbb{V}$: set of bugs in a project,

$\mathbb{E}$: set of directed edges $\mathbb{E}_1$ and undirected edges $\mathbb{E}_2$ such that:

$\mathbb{E}_1$: a directed edge between two bugs $v_1 \rightarrow v_2$ when v_1 is a child bug of v_2,

$\mathbb{E}_2$: an undirected edge between two bugs v_1 and v_2 such that bugs v_1 and v_2 have similar

issues (or are *Duplicate* bugs).

We can deduce $\mathbb{G}_{BugDep}$ from $\mathbb{G}_{InterRepoDep}$ using the following relation:

$$\mathbb{G}_{BugDep} = \{(v_1, v_2) | (v_1, v_2) \in \mathbb{G}_{InterRepoDep} | v_{bug} \wedge W(v_1 \rightarrow v_2) \geq 1\}$$

Mixed graph $\mathbb{G}_{BugDep}$ shows how bugs depend on each other. A directed edge between two bugs, say bug X and bug Y, demonstrates direct dependency because bug X is exhibited only when bug Y manifests. In other words, X is a child bug of Y. On the other hand, an undirected edge between X and Y would mean that X and Y are similar bugs, and on fixing X, bug Y was also fixed by using the same patch or similar resolution strategy.

Developer Networks

1. *Source-code based collaboration graph*:

   ***Definition*(Source-code based collaboration)**. Mixed directed weighted graph $\mathbb{G}_{SrcCodeColl} = (\mathbb{V}, \mathbb{E}, \mathbb{W})$ such that,

   $\mathbb{V}$: set of contributors who have been committed to the source code repository,

   $\mathbb{E}$: set of edges such that e is an edge between contributors v_1 and v_2 if v_1 and v_2 changed the same file,

   $\mathbb{W}$: set of weights w_1 and w_2 computed such that:

   w_1: weight of an edge which measures how many times two contributors have co-worked on the same file while adding new code to the project (feature enhancement),

   w_2: weight of an edge which measures how many times two contributors have co-

worked on the same file while fixing a bug in the project.

We can deduce $\mathbb{G}_{SrcCodeColl}$ from $\mathbb{G}_{InterRepoDep}$ using the following relation:

$$\mathbb{G}_{SrcCodeColl} = \{(v_1, v_2) | (v_1, v_2) \in \mathbb{G}_{InterRepoDep} | v_{contributor} \wedge \exists (v_1 \rightarrow src1 \wedge v_2 \rightarrow src2, (src1, src2) \in v_{func} \vee (src1, src2) \in v_{mod})\}$$

To capture the relationship between how contributors interact with each other while making changes to the source code either while adding new code to the repository or fixing a bug we build this mixed undirected graph.

2. *Bug tossing graphs*:

   ***Definition*(Bug-tossing)**. Directed weighted graph $\mathbb{G}_{BugToss} = (\mathbb{V}, \mathbb{E}, \mathbb{W})$ such that,

   $\mathbb{V}$: set of contributors who have been assigned a bug at least once during his association with a software project,

   $\mathbb{E}$: set of edges such that e is an edge from developer v_1 from v_2 if v_2 fixed a bug that was assigned to v_1,

   $\mathbb{W}$: weight of an edge w is equal to tossing probability between two contributors as described using equation 3.2.

   We can deduce $\mathbb{G}_{BugToss}$ from $\mathbb{G}_{InterRepoDep}$ using the following relation:

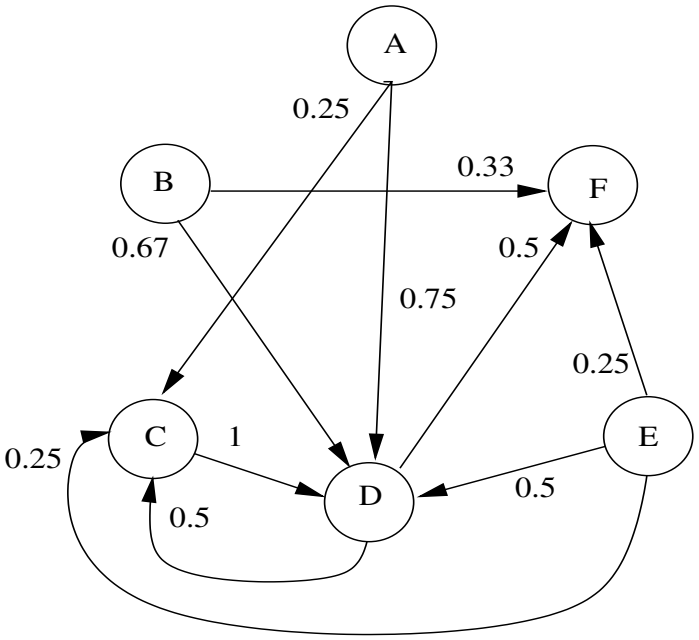
   $$\mathbb{G}_{BugToss} = \{(v_1, v_2) | (v_1, v_2) \in \mathbb{G}_{InterRepoDep} | v_{contributor} \wedge 0 \leq W(v_1 \rightarrow v_2) \leq 1\}$$

   When a bug is assigned to a developer for the first time and she is unable to fix it, the bug is assigned (tossed) to another developer. Thus a bug is tossed from one developer

to another until a developer is eventually able to fix it. Based on these tossing paths, *goal-oriented tossing graphs* were proposed by Jeong et al. [76]; for the rest of the dissertation, by "tossing graph" we refer to a goal-oriented tossing graph. Tossing graphs are weighted directed graphs such that each node represents a developer, and each directed edge from D_1 to D_2 represents the fact that a bug assigned to developer D_1 was tossed and eventually fixed by developer D_2. The weight of an edge between two developers is the probability of a toss between them, based on bug tossing history. We denote a tossing event from developer D to D_j as $D \hookrightarrow D_j$. The *tossing probability* (also known as the *transaction probability*) from developer D to D_j is defined by the following equation where k is the total number of developers who fixed bugs that were tossed from D:

$$Pr(D \hookrightarrow D_j) = \frac{\#(D \hookrightarrow D_j)}{\sum_{i=1}^{k} \#(D \hookrightarrow D_i)} \tag{2.1}$$

In this equation, the numerator is the number m of tosses from developer D to D_j such that D_j fixed the bug, while the denominator is the total number of tosses from D to any other developer D_i such that D_i fixed the bug. Note that if $k = 0$ for any developer D, it denotes that D has no outgoing edge in the bug tossing graph. To illustrate this, in Table 3.1 we provide sample tossing paths and show how toss probabilities are computed. For example, developer A has tossed four bugs in all, three that were fixed by D and one that was fixed by C, hence $Pr(A \hookrightarrow D) = 0.75$, $Pr(A \hookrightarrow C) = 0.25$, and $Pr(A \hookrightarrow F) = 0$. Note that developers who did not toss any bug (e.g., F) do not appear in the first column, and developers who did not fix



[t]

Figure 2.10: Tossing graph built using tossing paths in Table 3.1.

Tossing paths
$A \rightarrow B \rightarrow C \rightarrow D$
$A \rightarrow E \rightarrow D \rightarrow C$
$A \rightarrow B \rightarrow E \rightarrow D$
$C \rightarrow E \rightarrow A \rightarrow D$
$B \rightarrow E \rightarrow D \rightarrow F$

Developer who tossed the bug	**Total tosses**	**Developers who fixed the bug**					
		C		D		F	
		#	Pr	#	Pr	#	Pr
A	4	1	0.25	3	0.75	0	0
B	3	0	0	2	0.67	1	0.33
C	2	-	-	2	1.00	0	0
D	2	1	0.50	-	-	1	0.50
E	4	1	0.25	2	0.50	1	0.25

Table 2.3: Tossing paths and probabilities as used by Jeong et al.

any bugs (e.g., A) do not have a probability column. In Figure 3.3, we show the final tossing graph built using the computed tossing probabilities. It is common in open

source projects that when a bug in a module is first reported, the developers associated with that module are included in the list of assignees by default. The purpose of our automatic bug assignment approach is, given a bug report, to predict developers who could be potential fixers and email them, so that human intervention is reduced as much as possible.

To summarize, in this section we described the various networks that emerge from analyzing the various dependencies and relations among various software artifacts and contributors.

2.3 Search and Recommendation

As shown in Section 2.2, all software elements—code, contributors, bugs—of a project are deeply interconnected. As software projects grow, contributors are overwhelmed whenever they are faced with tasks such as program understanding or searching through the evolution data for a project. Examples of such frequent development tasks include understanding the control flow, finding dependencies among functions, finding modules that will be affected when a module is changed, etc. Similarly, during software maintenance, frequent tasks include developer coordination, keeping track of files that are being changed due to a bug-fix, what other module are affected when a bug is being fixed, finding which developer is suitable for fixing a bug (e.g., given that she has fixed similar bugs in the past or she has worked on the modules that the bug occurs in). Therefore, an integrated system that combines these multiple repositories along with efficient search techniques that and can answer a broad range of queries regarding the project's evolution history would

be beneficial to all classes of software practitioners: contributors, testers and managers. In addition, a framework that allows querying and aggregation on integrated evolution data for large projects would be beneficial for research in empirical software engineering, where data from multiple repositories are frequently used for hypothesis testing. Search-based querying refers to finding information that are already there in the repositories. On the other hand, recommendation-based querying refers to *intelligent searching* such that the results are *not readily* available and needs to be compiled from various data sources using several metrics and heuristics.

2.3.1 Querying Software Repositories

An example. Bug fix patch ID's should ideally be recorded both in the bug report (in the bug repository) and in the commit log for that patch (in the source code repository). However, in practice, this is often not the case. For instance, either commit logs contain bug ID's that refer to a corresponding bug-fix patch while bug repsoitories do not have information about which files were changed during a bug-fix or vice versa but not both. Let an example search query $\mathbb{Q}$ be: "*Return the list of files that were changed while fixing Mozilla bug 500495*". According to the commit logs, a fix to Mozilla bug 500495 resulted in changes to 186 files in Mozilla source code. However, the bug report just contains the patch from one of the files */cvsroot/mozilla/client.mk*. Additionally, the commit log contains the information of the developer who committed the final changes. The bug report on the other hand contains the information about who submitted the first patch for review and list of people who super-reviewed the patch. Using our framework, query $\mathbb{Q}$ can be written

as:

$$\mathbb{Q} = \{(v_1, v_2) | (v_1, v_2) \ \in \ \mathbb{G}_{InterRepoDep} | v_1 \ \in \ (v_{func} \vee v_{mod}) \ \wedge$$

$$(v_2 \ \in \ v_{bug} \ \wedge \ v_{bug}\texttt{[BugId = 500495]})\}$$

Search-based framework. To answer queries (similar to the example shown earlier in this subsection) which involves integrating information across multiple repositories, our search-based query framework simply uses the mixed-multi graph, $G_{InterRepoDep}$ we built in Section 2.2. For example, for the same example query instantiated earlier, "*Return the list of files that were changed while fixing Mozilla bug 500495*", our framework would simply return all nodes that are of type module and are connected to bug node of ID 500495. This relationship can be referred to the set of edges between a bug and a source code element (for instance, edge $B_2 \rightarrow S_4$ in Figure 2.7(b)).

2.3.2 Recommendation-based Querying

While integrating information across multiple repositories can help answer search-based queries, it is not enough for providing *recommendations* to software practitioners. For example, consider a query $\mathbb{Q}_1$: "Return a set of contributors who has the required expertise to fix bug B." For queries like this, it is not sufficient to use straight-forward relationships by querying the multi-mixed graphs. We show that these kinds of complex queries can be solved in two steps: first, by extracting entities and relationships among them that affect the complex query (by using the notion of hyperedges) and second, by using additional statistical-machine learning-network science based techniques on these extracted graphs to provide recommendations for the query. For example, to answer query $\mathbb{Q}_1$, we first extract

the network among all contributors involved in the bug fix process (such that they were assigned at least one bug during their lifetime) from $\mathbb{G}_{InterRepoDep}$ using the query:

$$\mathbb{Q}_1 = \{(v_1, v_2) | (v_1, v_2) \ \in \ \mathbb{G}_{InterRepoDep} | v_{contributor} \ \wedge \ (v_1[BugsAssigned > 0], v_2[BugsAssigned > 0]) \ \wedge \ 0 \leq W(v_1 \rightarrow v_2) \ \leq \ 1\}$$

This query returns the graph of contributors who were assigned at least one bug during their lifetime and the bugs were either fixed by them or tossed (aka reassigned) to other contributors in the project. After extracting this graph, as described in detail in Chapter 3, we show how we apply text-mining and probabilistic graph algorithms to answer query $\mathbb{Q}_1$ at a high precision rate. We provide another example of a commonly used recommendation-based query $\mathbb{Q}_2$ by software practitioners for finding patch reviewers: "Return a set of contributors who has the required expertise to review a patch submitted for bug B." Using the query below, we first extract the list of all contributors who have the required expertise to review a patch, i.e., they have either reviewed patches before, or they have worked on source code files directly or indirectly related to the patch file or they have fixed bugs dependent on bug B. Next, as we show in Chapter 6, we rank contributors (based on several other expertise metrics) in the list returned by query $\mathbb{Q}_2$ to further generate top-k experts for this task.

$$\mathbb{Q}_2 =$$

$$\{(v_1, v_2) | (v_1, v_2) \ \in \ \mathbb{G}_{InterRepoDep} | d \ \in \ v_{contributor}, b \ \in \ v_{bug}, s \ \in \ v_{source}, s[bugfix(b) = 1]$$

$$\wedge \ ((d[PatchReviewed > 0]) \ \vee \ (d[changed(s) = 1] \ \vee \ d[author(s) = 1])$$

$$\vee \ (d[changed(src_1) = 1] \ \vee \ d[author(src1) = 1], src_1 \ \in v_{source} \ \wedge \ W(s, src_1) > 0)$$

$$\vee \ (d[FixedBug(b_1) = 1] \ \wedge \ W(b, b_1) = 1 \ \wedge \ b_1 \ \in \ v_{bug}))\}$$

Therefore, at an abstract level, recommendation-based queries help us first filter out a small set of nodes and their relationships (i.e., edges between them) from the large mixed graphs to answer specific queries of interest. In this dissertation, we consider *four* such commonly-used, novel recommendation problems in the context of software development that can facilitate decision-making. We show that each of these recommendation problems require different set of graphs (i.e., software components) that are extracted primarily from the $G_{InterRepoDep}$ graph which we can further analyze to provide highly-accurate data-driven recommendations.

2.4 Open-source Projects Used As Benchmarks

To validate our recommendation and search algorithms, we used eleven popular open source applications written mainly in C, or combinations of C and C++. We select applications that have: (a) long release history, (b) significant size (in lines of code and modules), (c) a large set of developers who maintain them, (d) a large user base, who report bugs and submit patches. The above criteria are necessary for making meaningful statistical, behavioral, and evolutionary observations. We now provide a brief overview of each application.

- *Firefox* is the second most widely-used web browser [49]. Originally named *Phoenix*, it was renamed to *Firebird*, and then renamed to Firefox in 2004. We considered Phoenix

Application	Time span	Release	Language	Size (kLOC)		Ref. Chapters
				First release	Last release	
Firefox	1998-2010	92	C,C++	1,976	3,780	3,4,5,6,7
Eclipse	2001-2010	27	Java	828	1,903	3,5,6
Blender	2001-2009	28	C,C++	253	1,144	4,5
VLC	1998-2009	83	C,C++	144	293	4,5
MySQL	2000-2009	13	C,C++	815	991	5
Samba	1993-2009	78	C	5	1,045	5
Bind	2000-2009	171	C	169	321	5
Sendmail	1993-2009	55	C	25	87	5
OpenSSH	1999-2009	77	C	12	52	5
SQLite	2000-2009	169	C	17	65	5
Vsftpd	2001-2009	59	C	6	15	5

Table 2.4: Applications' evolution span, number of releases, programming language, size of first and last releases.

and Firebird in our study because the application's source code remained unchanged after the renamings. Firefox is mostly written in C and C++; it also contains HTML and JavaScript code that contribute to less than 3% of the total code.

- *Blender* is a 3D content creation suite, available for all major operating systems. It is mostly written in C and C++; it also has a Python component that contributes to less than 2% of the total code. We used the source code available in the SVN repository for our analyses [26].

- *VLC* is a popular [162], cross-platform open-source multimedia framework, player and server maintained by the VideoLAN project.

- *MySQL* is a popular [120] open source relational DBMS. MySQL was first released internally in 1995, followed by a publicly available Windows version in 1998. In 2001, with version 3.23, the source code was made available to the public. Therefore, for

measuring internal quality and maintenance effort, we consider 13 major and minor releases since 3.23. Our external quality measurements depend on the bug databases of the applications; for MySQL, the database stores bug and patch reports for major releases 3.23, 4.1, 5.0, 5.1, and 6.0 only, thus our external quality findings for MySQL are confined to major releases only.

- *Samba* is a tool suite that facilitates Windows-UNIX interoperability. According to its change log and history files, initial development for the program that would eventually become Samba was on and off between Dec. 1991 and Dec. 1993. However, the first officially announced release, then called Netbios for Unix was version 1.5.00, on Dec. 1, 1993. The first official release we could find was 1.5.14, dated Dec. 8, 1993. As shown in Table 1, over the past 15 years, the server grew from 5,514 LOC to more than 1,000,000 LOC.

- *Sendmail* is the leading email transfer agent today. While its initial development goes back to the early 1980s, we had to limit our analysis to version 8.6.4 (Oct. 1993) due to configuration and preprocessing problems that make analyzing earlier versions very difficult.

- *Bind* is the leading DNS server on the Internet. According to its official history, [2] Bind development goes back to the early 1980s, but the current line, Bind 9, is a major rewrite. We analyzed all the 171 versions, from 9.0.0b1 (Feb. 2000) to 9.6.1b1 (March 2009).

[2] `https://www.isc.org/software/bind/history`

- *OpenSSH* is the standard open source suite of the widely-used secure shell protocols. The first official release we could find was 1.0pre2, dating back to October 1999. Since then, OpenSSH has grown more than four-fold, from 12,819 LOC to 52,284 LOC over 78 official releases.

- *SQLite* is a popular library implementation of a self-contained SQL database engine. Starting from its initial version, 1.0 (Aug. 2000), comprising 17,723 LOC, SQLite has grown to 65,108 LOC in version 3.6.11 (Feb. 2009).

- *Vsftpd* stands for Very Secure FTP Daemon and is the FTP server in major Linux distributions. The first beta version, 0.0.9, was released on January 28, 2001. We analyzed its entire history, 60 versions over 8 years.

- *Quagga* is a tool suite for building software routers. Similar to Sendmail, we had to stop our analysis at version 0.96 (Aug. 2003) due to configuration and preprocessing problems with earlier versions.

- *Eclipse* is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. It is written mostly in Java.

In Table 2.4 we list these applications involved in our study, along with some key properties. The second column shows the time span we consider for each application, the third column contains the number of official releases within that time span; we analyzed all these releases. Column 4 shows the main language(s) the application was written in; some of the applications have small parts written in other languages, e.g., JavaScript. Columns

5 and 6 show application size, in effective LOC, for the first and last releases. Column 7 lists the reference to the chapters where these applications have been used as benchmarks for evaluating our prediction–recommendation models. The long time spans we consider (e.g., Samba has grown by a factor of 200x over 16 years) allow us to analyze evolution rigorously, obtain statistically significant results, and observe a variety of change patterns in the graphs. For each application, we have used its website to obtain the source code of official releases. We used applications' version control systems for extracting file change histories and patches. Finally, we extracted bug information from application-specific bug tracking systems.

2.5 Summary

In this chapter, we presented an overview of the data extraction and query framework. We first showed how various software development entities are related between and amongst themselves. Second, we showed the need for integrating information across multiple data repositories to precisely answer user queries. Third, we introduced the concept of search-based and recommendation-based frameworks and showed with specific instances the significance of each. In the next part of the dissertation, we describe the four recommendation frameworks we built to answer user-queries to facilitate quantitative decision-making in software development.

Recommendations for Four Decision-making Problems

Chapter 3

Automating Bug Assignment

Empirical studies indicate that automating the bug assignment process has the potential to significantly reduce software evolution effort and costs. Prior work has used machine learning techniques to automate bug assignment but has employed a narrow band of tools which can be ineffective in large, long-lived software projects. To redress this situation, in this chapter we employ a comprehensive set of machine learning tools and a probabilistic graph-based model (bug tossing graphs) that lead to highly-accurate predictions, and lay the foundation for the next generation of machine learning-based bug assignment. Our work is the first to examine the impact of multiple machine learning dimensions (classifiers, attributes, and training history) along with bug tossing graphs on prediction accuracy in bug assignment. We validate our approach on Mozilla and Eclipse, covering 856,259 bug reports and 21 cumulative years of development. We demonstrate that our techniques can achieve up to 86.09% prediction accuracy in bug assignment and significantly reduce tossing path lengths. We show that for our data sets the Naïve Bayes classifier coupled with

product–component features, tossing graphs and incremental learning performs best. Next, we perform an ablative analysis by unilaterally varying classifiers, features, and learning model to show their relative importance of on bug assignment accuracy. Finally, we propose optimization techniques that achieve high prediction accuracy while reducing training and prediction time.

3.1 Introduction

Most software projects use bug trackers to organize the bug fixing process and facilitate application maintenance. For instance, Bugzilla is a popular bug tracker used by many large projects, such as Mozilla, Eclipse, KDE, and Gnome [33]. These applications receive hundreds of bug reports a day; ideally, each bug gets assigned to a developer who can fix it in the least amount of time. This process of assigning bugs, known as *bug assignment*[1], is complicated by several factors: if done manually, assignment is labor-intensive, time-consuming and fault-prone; moreover, for open source projects, it is difficult to keep track of active developers and their expertise. Identifying the right developer for fixing a new bug is further aggravated by growth, e.g., as projects add more components, modules, developers and testers [75], the number of bug reports submitted daily increases, and manually recommending developers based on their expertise becomes difficult. An empirical

The work presented in this chapter have been published in the proceedings of the 2010 IEEE International Conference on Software Maintenance [17].

[1]In the software maintenance literature, "bug triaging" is used as a broader term referring to bug assignment, bug validation, marking duplicate bugs, etc. In this chapter, by bug triaging we mean bug assignment *only*, i.e., given a bug report that has been validated as a real bug, find the right developer whom the bug can be assigned to for resolution.

study by Jeong et al [76] reports that, on average, the Eclipse project takes about 40 days to assign a bug to the first developer, and then it takes an additional 100 days or more to reassign the bug to the second developer. Similarly, in the Mozilla project, on average, it takes 180 days for the first assignment and then an additional 250 days if the first assigned developer is unable to fix it. These numbers indicate that the lack of effective, automatic assignment and toss reduction techniques results in considerably high effort associated with bug resolution.

Effective and automatic bug assignment can be divided into two sub-goals: (1) assigning a bug for the first time to a developer, and (2) reassigning it to another promising developer if the first assignee is unable to resolve it, then repeating this reassignment process (*bug tossing*) until the bug is fixed. Our findings indicate that at least 93% of all "fixed" bugs in both Mozilla and Eclipse have been tossed at least once (tossing path length ≥ 1). Ideally, for any bug triage event, the bug should be resolved in a minimum number of tosses.

In this chapter, we explore the use of machine learning toward effective and automatic bug assignment along three dimensions: the choice of classification algorithms, the software process attributes that are instrumental to constructing accurate prediction models, and the efficiency–precision trade-off. Our thorough exploration along these dimensions have lead us to develop techniques that achieve high levels of bug assignment accuracy and bug tossing reduction. In Figure 3.1 we show the nodes and edges we extract from the multi-mixed graph to build the bug tossing graph (explained in Section 2.2.1) for automating the bug assignment process.

Similar to prior work, we test our approach on the fixed bug data sets for Mozilla

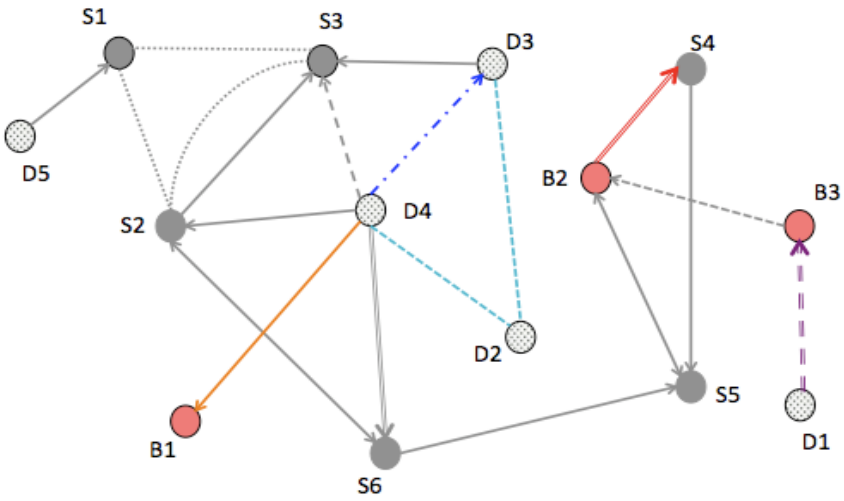


Figure 3.1: Hypergraph extraction for automatically assigning bugs (edges for graph $\mathbb{G}_{BugToss}$).

and Eclipse. Our techniques achieve a bug assignment prediction accuracy of up to 85% for Mozilla and 86% for Eclipse. We also find that using our approach reduces the length of tossing paths by up to 86% for correct predictions and improves the prediction accuracy by up to 10.78 percentage points compared to previous approaches. We demonstrate that on average, the highest prediction accuracy is achieved using a Naïve Bayes classifier, with products/components as attributes, with bug triaging graphs, and with incremental learning (aka intra-fold updates). We then follow a standard machine learning *ablative analysis*: [2] we take our best case (top of the figure) and unilaterally vary the underlying attributes to show their relative importance in Section 3.4—the corresponding subsections are shown on the bottom of the figure. The primary goal of our work is to find the optimal set of machine learning techniques (classifiers, features, tossing graphs and incremental learning) to improve bug assignment accuracy in large projects and we show this optimal set for our

[2] Ablative analysis is a methodology to quantify the effects of each attribute in a multi-attribute model.

data sets, Mozilla and Eclipse. The optimal set of techniques we report can change with changes in data sets for the same project or across other projects, or with changes in the underlying supervised learning algorithm and we address these issues as potential threats to validity of our approach in Section 3.5.

Wide range of classification algorithms. Machine learning is used for recommendation purposes in various areas such as climate prediction, stock market analysis, or prediction of gene interaction in bioinformatics [170]. Machine learning techniques, in particular classifiers,[3] have also been employed earlier for automating bug assignment. These automatic bug assignment approaches [8, 39, 12, 36] use the history of bug reports and developers who fixed them to train a classifier. Later, when keywords from new bug reports are given as an input to the classifier, it recommends a set of developers who have fixed similar classes of bugs in the past and are hence considered potential bug-fixers for the new bug. Prior work that has used machine learning techniques for prediction or recommendation purposes has found that prediction accuracy depends on the choice of classifier, i.e., a certain classifier outperforms other classifiers for a specific kind of a problem [170]. Previous studies [76, 8, 39, 12] only used a subset of text classifiers and did not aim at analyzing which is the best classifier for this problem. Our work is the first to examine the impact of multiple machine learning dimensions (classifiers, attributes, and training history) on prediction accuracy in bug assignment and tossing. In particular, this is the first study in the area of bug assignment to consider, and compare the performance of, a broad range of classifiers along with tossing graphs: Naïve Bayes Classifier, Bayesian Networks, C4.5 and

[3] A *classifier* is a machine learning algorithm that can be trained using input attributes (also called feature vectors) and desired output classes; after training, when presented with a set of input attributes, the classifier predicts the most likely output class.

Support Vector Machines.

Effective tossing graphs. Jeong et al. [76] have introduced tossing graphs for studying the process of tossing, i.e., bug reassignment; they proposed automating bug assignment by building bug tossing graphs from bug tossing histories. While classifiers and tossing graphs are effective in improving the prediction accuracy for assignment and reducing tossing path lengths, their accuracy is threatened by several issues: outdated training sets, inactive developers, and imprecise, single-attribute tossing graphs. Prior work [76] has trained a classifier with fixed bug histories; for each new bug report, the classifier recommends a set of potential developers, and for each potential developer, a tossing graph—whose edges contain tossing probabilities among developers—is used to predict possible re-assignees. However, the tossing probability alone is insufficient for recommending the most competent active developer (see Section 3.3.6 for an example). In particular, in open source projects it is difficult to keep track of active developers and their expertise. To address this, in addition to tossing probabilities, we label tossing graph edges with developer expertise and tossing graph nodes with developer activity, which help reduce tossing path lengths significantly. We demonstrate the importance of using these additional attributes in tossing graphs by performing a fine-grained per-attribute ablative analysis which reveals how much each attribute affects the prediction accuracy. We found that each attribute is instrumental for achieving high prediction accuracy, and overall they make pruning more efficient and improve prediction accuracy by up to 22% points when compared to prediction accuracy obtained in the absence of the attributes.

Accurate yet efficient classification. Anvik's dissertation [9] has demonstrated

that choosing a subset of training data can reduce the computation time during the classification process, while achieving similar prediction accuracies to using the entire data set; three methods—random, strict and tolerant, were employed for choosing a subset of training data set as we explain in related work (Section 8.1). In our work, in addition to classification, we also use a probabilistic ranking function based on bug tossing graphs for developer recommendation. Since bug tossing graphs are time-sensitive, i.e., tossing probabilities change with time, the techniques used by Anvik are not applicable in our case (where bug reports were not sorted by time for selection). Therefore, in this chapter, we propose to shorten the time-consuming classification process by selecting the most recent history for identifying developer expertise. As elaborated in Section 3.4.7 we found that by using just one third of all bug reports we could achieve prediction accuracies similar to the best results of our original experiments where we used the complete bug history. Therefore, our third contribution in this chapter is showing how, by using a subset of bug reports, we can achieve accurate yet efficient bug classification that significantly reduces the computational effort associated with training.

Our chapter is structured as follows. In Section 3.2 we define terms and techniques used in bug assignment. In Section 3.3 we elaborate on our contributions, techniques and implementation details. We present our experimental setup and results in Section 3.4. Finally, we discuss threats to validity of our study in Section 3.5.

3.2 Preliminaries

We first define several machine learning and bug assignment concepts that form the basis of our approach.

3.2.1 Machine Learning for Bug Categorization

Classification is a supervised machine learning technique for deriving a general trend from a training data set. The *training data set* (TDS) consists of pairs of input objects (called feature vectors), and their respective target outputs. The task of the supervised learner (or classifier) is to predict the output given a set of input objects, after being trained with the TDS. Feature vectors for which the desired outputs are already known form the *validation data set* (VDS) that can be used to test the accuracy of the classifier. A bug report contains a description of the bug and a list of developers that were associated with a specific bug, which makes text classification applicable to bug assignment. Machine learning techniques were used by previous bug assignment works [8, 39, 12]: archived bug reports form feature vectors, and the developers who fixed the bugs are the outputs of the classifier. Therefore, when a new bug report is provided to the classifier, it predicts potential developers who can fix the bug based on their bug fixing history.

Feature vectors. The accuracy of a classifier is highly dependent on the feature vectors in the TDS. Bug titles and summaries have been used earlier to extract the keywords that form feature vectors. These keywords are extracted such that they represent a specific class of bugs. For example, if a bug report contains words like “icon,” “image,” or “display,” it can be inferred that the bug is related to application layout, and is assigned to the

"layout" class of bugs. We used multiple text classification techniques (`tf-idf`, stemming, stop-word and non-alphabetic word removal [101]) to extract relevant keywords from the actual bug report; these relevant keywords constitute a subset of the attributes used to train the classifier.

Text Classification Algorithms

We now briefly describe each classifier we used.

Naïve Bayes Classifier. Naïve Bayes is a probabilistic technique that uses Bayes' rule of conditional probability to determine the probability that an instance belongs to a certain class. Bayes' rule states that "the probability of a class conditioned on an observation is proportional to the prior probability of the class times the probability of the observation conditioned on the class" and can be denoted as follows:

$$P(class|observation) = \frac{P(observation|class) * P(class)}{P(observation)} \tag{3.1}$$

For example, if the word *concurrency* occurs more frequently in the reports resolved by developer A than in the reports resolved by developer B, the classifier would predict A as a potential fixer for a new bug report containing the word *concurrency*. "Naïve Bayes" is so called because it makes the strong assumption that features are independent of each other, given the label (the developer who resolved the bug). Even though this assumption does not always hold, Naïve Bayes-based recommendation or prediction performs well in practice [43].

Bayesian Networks. A Bayesian Network [86] is a probabilistic model that is

used to represent a set of random variables and their conditional dependencies by using a directed acyclic graph (DAG). Each node in the DAG denotes a variable, and each edge corresponds to a potential direct dependence relationship between a pair of variables. Each node is associated with a conditional probability table (CPT) which gives the probability that the corresponding variable takes on a particular value given the values of its parents.

C4.5. The C4.5 algorithm [135] builds a decision tree based on the attributes of the instances in the training set. A prediction is made by following the appropriate path through the decision tree based on the attribute values of the new instance. C4.5 builds the tree recursively in a greedy fashion. Each interior node of the tree is selected to maximize the information gain of the decision at that node as estimated by the training data. The information gain is a measure of the predictability of the target class (developer who will resolve the bug report) from the decisions made along the path from the root to this node in the tree. The sub-trees end in leaf nodes at which no further useful distinctions can be made and thus a particular class is chosen.

Support Vector Machines. A SVM (Support Vector Machine [28]) is a supervised classification algorithm that finds a decision surface that maximally separates the classes of interest. That is, the closest points to the surface on each side are as far as possible from the decision surface. It employs kernels to represent non-linear mappings of the original input vectors. This allows it to build highly non-linear decision surfaces without an explicit representation of the non-linear mappings. Four kinds of kernel functions are commonly used: Linear, Polynomial, Gaussian Radial Basis Function (RBF) and Sigmoid. In our study we use Polynomial and RBF functions as they have been found to be most

effective in text classification.

3.2.2 Folding

Early bug assignment approaches [76, 8, 39] divided the data set into two subsets: 80% for TDS and 20% for VDS. Bettenburg et al. [12] have used folding (similar to split-sample validation techniques from machine learning [170]) in the context of detecting duplicate bug reports. In a folding-based training and validation approach, also known as cross-validation, (illustrated in Figure 3.2), the algorithm first collects all bug reports to be used for TDS, [4] sorts them in chronological order (based on the fixed date of the bug) and then divides them into n folds. In the first run, fold 1 is used to train the classifier and then to predict the VDS. [5] In the second run, fold 2 bug reports are added to TDS. In general, after validating the VDS from fold n, that VDS is added to the TDS for validating fold $n+1$. To reduce experimental bias [170], similar to Bettenburg et al., we chose $n = 11$ and carried out 10 iterations of the validation process using incremental learning. Note that incremental learning is not a contribution of our work; incremental learning is a standard technique to improve the prediction accuracy in any supervised or unsupervised learning algorithms in machine learning [85]. Rather, we show that, similar to other software maintenance problems like duplicate bug detection [12], fine-grained incremental learning is important for improving bug assignment accuracy, i.e., to have the classifier trained with most recent data (or bug reports). Therefore, we only use folding to compare our work with prior studies in automatic bug assignment where split-sample validation was used; though

[4] Training Data Set (TDS) used to train the classifier; see Section 3.2.1 for more details.
[5] Validation Data Set (VDS) used to validate the classifier; see Section 3.2.1 for more details.

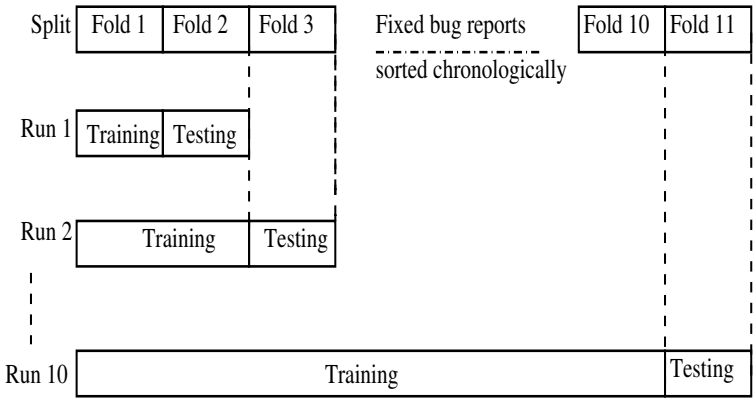


Figure 3.2: Folding techniques for classification as used by Bettenburg et al.

our best result was achieved using fine-grained incremental learning.

3.2.3 Goal-oriented Tossing Graphs

When a bug is assigned to a developer for the first time and she is unable to fix it, the bug is assigned (tossed) to another developer. Thus a bug is tossed from one developer to another until a developer is eventually able to fix it. Based on these tossing paths, *goal-oriented tossing graphs* were proposed by Jeong et al .[76]; for the rest of the chapter, by "tossing graph" we refer to a goal-oriented tossing graph. Tossing graphs are weighted directed graphs such that each node represents a developer, and each directed edge from D_1 to D_2 represents the fact that a bug assigned to developer D_1 was tossed and eventually fixed by developer D_2. The weight of an edge between two developers is the probability of a toss between them, based on bug tossing history. We denote a tossing event from developer D to D_j as $D \hookrightarrow D_j$. The *tossing probability* (also known as the *transaction probability*) from developer D to D_j is defined by the following equation where k is the total number of

developers who fixed bugs that were tossed from D:

$$Pr(D \hookrightarrow D_j) = \frac{\#(D \hookrightarrow D_j)}{\sum_{i=1}^{k} \#(D \hookrightarrow D_i)} \tag{3.2}$$

Tossing paths
$A \rightarrow B \rightarrow C \rightarrow D$
$A \rightarrow E \rightarrow D \rightarrow C$
$A \rightarrow B \rightarrow E \rightarrow D$
$C \rightarrow E \rightarrow A \rightarrow D$
$B \rightarrow E \rightarrow D \rightarrow F$

Developer who tossed the bug	**Total tosses**	**Developers who fixed the bug**					
		C		D		F	
		#	Pr	#	Pr	#	Pr
A	4	1	0.25	3	0.75	0	0
B	3	0	0	2	0.67	1	0.33
C	2	-	-	2	1.00	0	0
D	2	1	0.50	-	-	1	0.50
E	4	1	0.25	2	0.50	1	0.25

Table 3.1: Tossing paths and probabilities as used by Jeong et al.

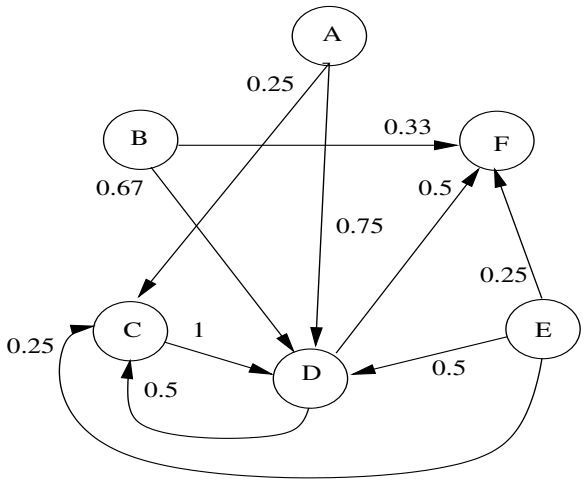


Figure 3.3: Tossing graph built using tossing paths in Table 3.1.

In this equation, the numerator is the number m of tosses from developer D to D_j

such that D_j fixed the bug, while the denominator is the total number of tosses from D to any other developer D_i such that D_i fixed the bug. Note that if $k = 0$ for any developer D, it denotes that D has no outgoing edge in the bug tossing graph. To illustrate this, in Table 3.1 we provide sample tossing paths and show how toss probabilities are computed. For example, developer A has tossed four bugs in all, three that were fixed by D and one that was fixed by C, hence $Pr(A \hookrightarrow D) = 0.75$, $Pr(A \hookrightarrow C) = 0.25$, and $Pr(A \hookrightarrow F) = 0$. Note that developers who did not toss any bug (e.g., F) do not appear in the first column, and developers who did not fix any bugs (e.g., A) do not have a probability column. In Figure 3.3, we show the final tossing graph built using the computed tossing probabilities. It is common in open source projects that when a bug in a module is first reported, the developers associated with that module are included in the list of assignees by default. The purpose of our automatic bug assignment approach is, given a bug report, to predict developers who could be potential fixers and email them, so that human intervention is reduced as much as possible.

Prediction accuracy. If the first developer in our prediction list matches the actual developer who fixed the bug, we have a hit for the Top 1 developer count. Similarly, if the second developer in our prediction list matches the actual developer who fixed the bug, we have a hit for the Top 2 developer count. For example, if there are 100 bugs in the VDS and for 20 of those bugs the actual developer is the first developer in our prediction list, the prediction accuracy for Top 1 is 20%; similarly, if the actual developer is in our Top 2 for 60 bugs, the Top 2 prediction accuracy is 60%.

3.3 Methodology

3.3.1 Choosing Effective Classifiers and Features

In this section we discuss appropriate selection of machine learning algorithms and feature vectors for improving the classification process.

Choosing the Right Classifier

Various approaches that use machine learning techniques for prediction or recommendation purposes have found that prediction accuracy depends on the choice of classifier, i.e., for a specific kind of a problem, a certain classifier outperforms other classifiers [170]. Previous bug classification and assignment studies [76, 8, 39, 12] only used a subset of text classifiers and did not aim at analyzing which classifier works best for bug assignment. Our work is the first study to consider an extensive set of classifiers which are commonly used for text classification: Naïve Bayes Classifier, Bayesian Networks, C4.5 and two types of SVM classifiers (Polynomial and RBF). We found that for bug assignment it is not possible to select one classifier which is better than the rest, either for a specific project or for any project in general. Since classifier performance is also heavily dependent on the quality of bug reports, in general we could not propose choosing a specific classifier *a priori* for a given project. Interestingly, computationally-intensive classification algorithms such as C4.5 and SVM do not consistently outperform simpler algorithms such as Naïve Bayes and Bayesian Networks. We provide details of our prediction accuracy using each classifier in Section 3.4.2.

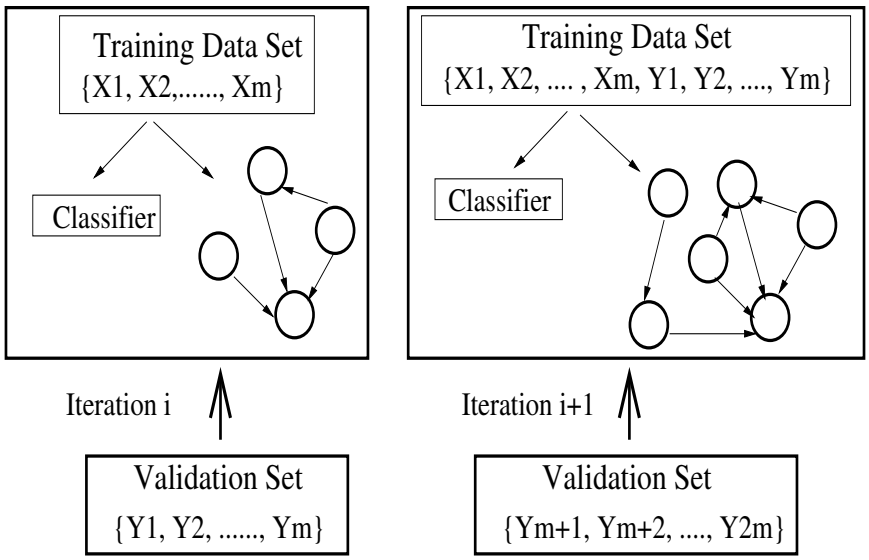
Feature Selection

Classifier performance is heavily dependent on feature selection [170]. Prior work [8, 39, 12] has used keywords from the bug report and developer name or ID as features (attributes) for the training data sets; we also include the product and component the bug belongs to. For extracting relevant words from bug reports, we employ `tf-idf`, stemming, stop-word and non-alphabetic word removal [101]. We use the Weka toolkit [165] to remove stop words and form the word vectors for the dictionary (via the `StringtoWordVector` class with tf-idf enabled).

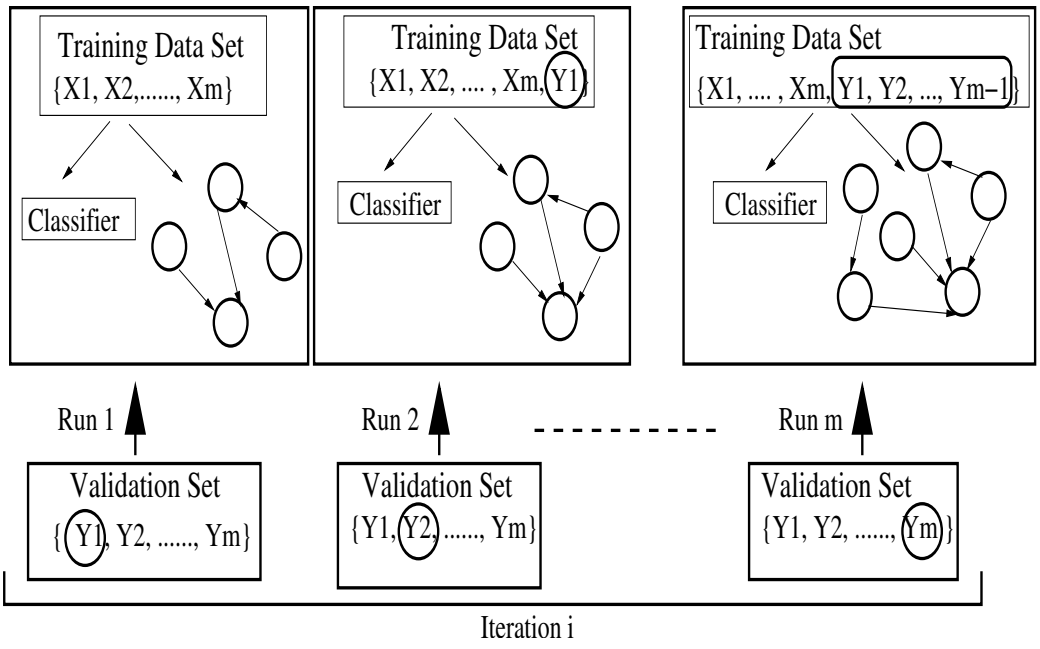
3.3.2 Incremental Learning

Prior work [76, 12] has used *inter*-fold updates, i.e., the classifier and tossing graphs are updated after each fold validation, as shown in Figure 3.4(a). With inter-fold updates, after validating the VDS from fold n, the VDS is added to the TDS for validating fold $n + 1$. However, consider the example when the TDS contains bugs 1–100 and the VDS contains bugs 101–200. When validating bug 101, the classifier and tossing graph are trained based on bugs 1–100, but from bug 102 onwards, the classifier and tossing graph are not up-to-date any more because they do not incorporate the information from bug 101. As a result, when the validation sets contain thousands of bugs, this incompleteness affects prediction accuracy. Therefore, to achieve high accuracy, it is essential that the classifier and tossing graphs be updated with the latest bug fix; we use a fine-grained, *intra*-fold updating technique (i.e., incremental learning) for this purpose.

We now proceed to describing intra-fold updating. After the first bug in the



(a) Updates after each validation set (Bettenburg et al.)



(b) Updates after each bug (our approach)

Figure 3.4: Comparison of training and validation techniques.

validation fold has been used for prediction and accuracy has been measured, we add it to the TDS and re-train the classifier as shown in Figure 3.11(b). We also update the tossing graphs by adding the tossing path of the just-validated bug. This guarantees that for each bug in the validation fold, the classifier and the tossing graphs incorporate information about all preceding bugs.

3.3.3 Multi-featured Tossing Graphs

Tossing graphs are built using tossing probabilities derived by analyzing bug tossing histories, as explained in Section 3.2.3. Jeong et al. [76] determined potential tossees as follows: if developer A has tossed more bugs to developer B than to developer D, in the future, when A cannot resolve a bug, the bug will be tossed to B, i.e., tossing probabilities determine tossees. However, this approach might be inaccurate in certain situations: suppose a new bug belonging to class K_1 is reported, and developer A was assigned to fix it, but he is unable to fix it; developer B has never fixed any bug of type K_1, while D has fixed 10 bugs of type K_1. The prior approach would recommend B as the tossee, although D is more likely to resolve the bug than B. Thus, although tossing graphs reveal tossing probabilities among developers, they should also contain information about which classes of bugs were passed from one developer to another; we use multi-feature tossing graphs to capture this information.

Another problem with the classifier- and tossing graph-based approaches is that it is difficult to identify retired or inactive developers. This issue is aggravated in open source projects: when developers work voluntarily, it is difficult to keep track of the current

Product	Component	Tossing paths
P_1	C_1	$A \rightarrow B \rightarrow C$
P_1	C_3	$F \rightarrow A \rightarrow B \rightarrow E$
P_2	C_5	$B \rightarrow A \rightarrow D \rightarrow C$
P_1	C_3	$C \rightarrow E \rightarrow A \rightarrow D$
P_1	C_1	$A \rightarrow B \rightarrow E \rightarrow C$
P_1	C_3	$B \rightarrow A \rightarrow F \rightarrow D$

Developer bug assigned	Total tosses	Developers who fixed the bug					
		C		D		E	
		#	Pr	#	Pr	#	Pr
A	6	3	0.5	2	0.33	1	0.17

Developer	Last Activity (in days)
A	20
C	70
D	50
E	450

Table 3.2: Example of tossing paths, associated tossing probabilities and developer activity.

set of active developers associated with the project. Anvik et al. [8] and Jeong et al. [76] have pointed out this problem and proposed solutions. Anvik et al. use a heuristic to filter out developers who have contributed fewer than 9 bug resolutions in the last 3 months of the project. Jeong et al. assume that, when within a short time span many bugs get tossed from a developer D to others, leading to an increase in the number of outgoing edges in the tossing graph from D's node, D is a potentially retired developer. They suggest that this information can be used in real-world scenarios by managers to identify potentially inactive developers. Therefore, in their automatic bug assignment approach they still permit assignment of bugs to inactive developers, which increases the length of the predicted tossing paths. In contrast, we restrict potential assignees to active developers only, and do so with a minimum number of tosses.

The tossing graphs we build have additional labels compared to Jeong et al.: for

each bug that contributes to an edge between two developers, we attach the bug class (product and component)[6] to that edge; moreover, for each developer in the tossing graph, we maintain an activity count, i.e., the difference between the date of the bug being validated and the date of the last activity of that developer.

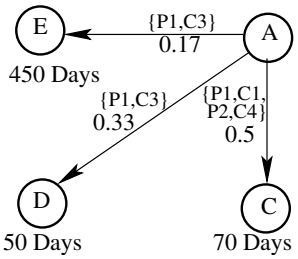


Figure 3.5: Multi-feature tossing graph (partial) derived from data in Table 3.2.

Building Multi-feature Tossing Graphs

As discussed earlier in Section 3.3.3, tossing probabilities are a good start toward indicating potential bug fixers, but they might not be appropriate at all times. Therefore, the tossing graphs we generate have three labels in addition to the tossing probability: bug product and bug component on each edge, and number of days since a developer's last activity on each node. For example, consider three bugs that have been tossed from D_1 to D_2 and belong to three different product-component sets: $\{P_1, C_1\}$, $\{P_1, C_3\}$, and $\{P_2, C_5\}$. Therefore, in our tossing graph, the product-component set for the edge between D_1 and D_2 is $\{\{P_1, C_1\}, \{P_1, C_3\}, \{P_2, C_5\}\}$. Maintaining these additional attributes is also helpful when bugs are re-opened. Both developer expertise and tossing histories change

[6] *Products* are smaller projects within a large project. *Components* are sub-modules in a product. For example, Firefox is a product in Mozilla and Bookmarks is a component of Firefox.

over time, hence it is important to identify the last fixer for a bug and a potential tossee after the bug has been re-opened.

We now present three examples that demonstrate our approach and show the importance of multi-feature tossing graphs. The examples are based on the tossing paths, the product–component the bug belongs to, and the developer activity, as shown in Table 3.2. Suppose that at some point in our recommendation process for a specific bug, the classifier returns A as the best developer for fixing the bug. However, if A is unable to resolve it, we need to use the tossing graph to find the next developer. We will present three examples to illustrate which neighbor of A to choose, and how the selection depends on factors like bug source and developer activity, in addition to tossing probability. For the purpose of these examples, we just show a part of the tossing graph built from the tossing paths shown in Table 3.2; we show the node for developer A and its neighbors in the tossing graph in Figure 3.5, as the tossee selection is dependent on these nodes alone.

Example I. Suppose we encounter a new bug B_1 belonging to product P_1 and component C_5, and the classifier returns A as the best developer for fixing the bug. If A is unable to fix it, by considering the tossing probability and product–component match, we conclude that it should be tossed to C.

Example II. Consider a bug B_2 belonging to product P_1 and component C_3. If A is unable to fix it, although C has a higher transaction probability than D, because D has fixed bugs earlier from product P_1 and component C_3, he is more likely to fix it than C. Hence in this case the bug gets tossed from A to D.

Example III. Based on the last active count for E in Figure 3.5, i.e., 450 days,

it is likely that E is a retired developer. In our approach, if a developer has been inactive for more than 100 days,[7] we choose the next potential neighbor (tossee) from the reference node A. For example, consider bug B_3 which belongs to product P_1 and component C_3, which has been assigned to A and we need to find a potential tossee when A is unable to resolve it. We should never choose E as a tossee as he is a potential retired developer and hence, in this particular case, we choose C as the next tossee. We also use activity counts to prune inactive developers from classifier recommendations. For example, if the classifier returns n recommendations and we find that the i^{th} developer is probably retired, we do not select him, and move on to the $(i+1)^{st}$ developer.

Ranking Function

As explained with examples in Section 3.3.3, the selection of a tossee depends on multiple factors. We thus use a ranking function to rank the tossees and recommend a potential bug-fixer. We first show an example of our developer prediction technique for a real bug from Mozilla and then present the ranking function we use for prediction.

Example (Mozilla bug 254967). For this particular bug, the first five developers predicted by the Naïve Bayes classifier are {*bugzilla*, *fredbezies*, *myk*, *tanstaafl*, *ben.bucksch*}. However, since *bryner* is the developer who actually fixed the bug, our classifier-only prediction is inaccurate in this case. If we use the tossing graphs in addition to the classifier, we select the most likely tossee for *bugzilla*, the first developer in the classifier ranked list. In Figure 3.6, we present the node for *bugzilla* and its neighbors.[8] If we

[7] Choosing 100 days as the threshold was based on Anvik et al. [8]'s observation that developers that have been inactive for three months or more are potentially retired.

[8] For clarity, we only present the nodes relevant to this example, and the labels at the point of validating

rank the outgoing edges of *bugzilla* based on tossing probability alone, the bug should be tossed to developer *ddahl*. Though *bryner* has lower probability, he has committed patches to the product "Firefox" and component "General" that bug 254967 belong to. Therefore, our algorithm will choose *bryner* as the potential developer over *ddahl*, and our prediction matches the actual bug fixer. Our ranking function also takes into account developer activity; in this example, however, both developers *ddahl* and *bryner* are active, hence comparing their activities is not required. To conclude, our ranking function increases prediction accuracy while reducing tossing lengths; the actual tossing length for this particular Mozilla bug was 6, and our technique reduces it to 2.

We now describe our algorithm for ranking developers. Similar to Jeong et al., we first use the classifier to predict a set of developers named CP (Classifier Predicted). Using the last-activity information, we remove all developers who have not been active for the past 100 days from CP. We then sort the developers in CP using the fix counts from the developer profile (as described in Section 3.3.6).

Suppose the CP is $\{D_1, D_2, D_3, \ldots, D_j\}$. For each D_i in the sorted CP, we rank its tossees T_k (outgoing edges in the tossing graph) using the following ranking function:

$$\begin{aligned}\textbf{Rank}\,(T_k) = {} & Pr(D_i \hookrightarrow T_k) + \\ & MatchedProduct(T_k) + \\ & MatchedComponent(T_k) + \\ & LastActivity(T_k)\end{aligned}$$

this bug; due to incremental learning, label values will change over time.

The tossing probability, $Pr(D_i \hookrightarrow T_k)$, is computed using equation 3.2 (Section 3.2). The function $MatchedProduct(T_k)$ returns 1 if the product the bug belongs to exists in developer T_k's profile, and 0 otherwise. Similarly, the function $MatchedComponent(T_k)$ returns 1 if the component the bug belongs to exists in developer T_k's profile. Note that the $MatchedComponent(T_k)$ attribute is computed only when $MatchedProduct(T_k)$ returns 1. The *LastActivity* function returns 1 if T_k's last activity was in the last 100 days from the date the bug was reported. As a result, $0 < \text{Rank}(T_k) \leq 4$. We then sort the tossees T_k by rank, choose the developer T_i with highest rank and add it to the new set of potential developers, named ND. Thus after selecting T_i, where $i = 1, 2, \ldots, j$, the set ND becomes $\{D_1, T_1, D_2, T_2, D_3, T_3, \ldots, D_j, T_j\}$. When measuring our prediction accuracy, we use the first 5 developers in ND.

If two potential tossees T_i and T_j have the same rank, and both are active developers, and both have the same tossing probabilities for bug B (belonging to product P and component C), we use developer profiles to further rank them. There can be two cases in this tie: (1) both T_i and T_j's profiles contain $\{P, C\}$, or (2) there is no match with either P or C. For the first case, consider the example in Table 3.3: suppose a new bug B belongs to $\{P_1, C_1\}$. Assume T_i and T_j are the two potential tossees from developer D (where D has been predicted by the classifier) and suppose both T_i and T_j have the same tossing probabilities from D. From developer profiles, we find that T_j has fixed more bugs for $\{P_1, C_1\}$ than T_i, hence we choose T_j (case 1). If the developers have the same fix count, or neither has P and/or C in their profile (case 2), we randomly choose one.

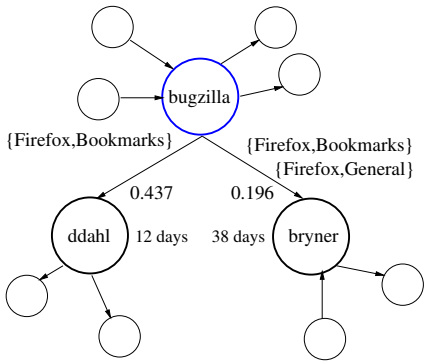


Figure 3.6: Actual multi-feature tossing graph extracted from Mozilla.

Developer ID	Product-Component	Fix count
T_i	$\{P_1, C_1\}$	3
	$\{P_1, C_7\}$	18
	$\{P_9, C_6\}$	7
T_j	$\{P_1, C_1\}$	13
	$\{P_4, C_6\}$	11

Table 3.3: Sample developer profiles: developer IDs and number of bugs they fixed in each product–component pair.

3.3.4 Ablative Analysis for Tossing Graph Attributes

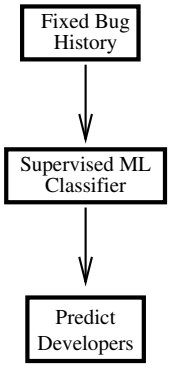
As explained in Section 3.3.6, our ranking function for tossing graphs contains additional attributes compared to the original tossing graphs by Jeong et al. Therefore, we were interested to evaluate the importance of each attribute; to measure this, we performed another ablative analysis. We choose only two attributes out of three (product, component and developer activity) at a time and compute the decrease in prediction accuracy in the absence of the other attribute. For example, if we want to measure the significance of the "developer activity" attribute, we use only product and component attributes in our ranking function described in Section 3.3.6 and compute the decrease in prediction accuracy. In Section 3.4.5 we discuss the results of our ablative analysis and argue the importance of

the attributes we propose.

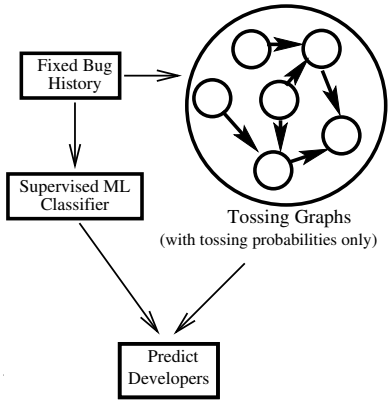
3.3.5 Accurate Yet Efficient Classification

One of the primary disadvantages of fine-grained incremental learning is that it is time consuming [127, 132, 154]. Previous studies which used fine-grained incremental learning for other purposes [89] found that using *a part* of the bug repository history for classification might yield comparable and stable results to using the *entire* bug history. Similarly, we intended to find how many past bug reports we need to train the classifier on in order to achieve a prediction accuracy comparable to the highest prediction accuracy attained when using fold 1–10 as the TDS and fold 11 as the VDS.

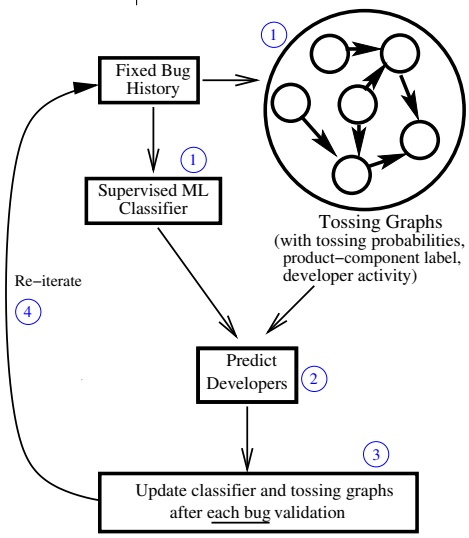
We now present the procedure we used for finding how much history is enough to yield high accuracy. We first built the tossing graphs using the TDS until fold 10; building tossing graphs and using them to rank developers is not a time consuming task, hence in our approach tossing graphs cover the entire TDS. We then incrementally started using sets of 5,000 bug reports from fold 10 downwards, in descending chronological order, as our TDS for the classifier, and measured our prediction accuracy for bugs in fold 11 (VDS); we continued this process until addition of bug reports did not improve the prediction accuracy any more, implying stabilization. Note that by this method our VDS remains constant. We present the results of our optimization in Section 3.4.7.



(a) Classifier-based bug assignment



(b) Classifiers coupled with tossing graphs



(c) Incremental learning and multi-feature tossing graphs (our approach)

Figure 3.7: Comparison of bug assignment techniques.

3.3.6 Implementation

In Figure 3.7 we compare our approach to previous techniques. Initial work in this area (Figure 3.7(a)) used classifiers only [8, 39, 12, 36]; more recent work by Jeong et al. [76] (Figure 3.7(b)) coupled classifiers with tossing graphs. Our approach (Figure 3.7(c)) adds fine-grained incremental learning and multi-feature tossing graphs. Our algorithm consists of four stages, as labeled in the figure: (1) initial classifier training and building the tossing graphs, (2) predicting potential developers, using the classifier and tossing graphs, (3) measuring prediction accuracy, (4) updating the training sets using the bugs which have been already validated, re-running the classifier and updating the tossing graphs. We iterate these four steps until all bugs have been validated.

Developer Profiles

Developer ID	Product-Component	Fix count
D_1	$\{P_1, C_2\}$	3
	$\{P_1, C_7\}$	18
	$\{P_9, C_6\}$	7

Table 3.4: Sample developer profile.

We maintain a list of all developers and their history of bug fixes. Each developer D has a list of product-component pairs $\{P, C\}$ and their absolute count attached to his or her profile. A sample developer profile is shown in Table 3.4, e.g., developer D_1 has fixed 3 bugs associated with product P_1 and component C_2. This information is useful beyond bug assignments; for example, while choosing moderators for a specific product or component it is a common practice to refer to the developer performance and familiarity with that

product or component.

Classification

Given a new bug report, the classifier produces a set of potential developers who could fix the bug. We describe the classification process in the remainder of this subsection.

Choosing fixed bug reports. We use the same heuristics as Anvik et al. [8] for obtaining fixed bug reports from all bug reports in Bugzilla. First, we extract all bugs marked as "verified" or "resolved"; next, we remove all bugs marked as "duplicate" or "works-for-me," which leaves us with the set containing fixed bugs only.

Accumulating training data. Prior work [8, 39, 12] has used keywords from the bug report and developer name or ID as attributes for the training data sets; we also include the product and component the bug belongs to. For extracting relevant words from bug reports, we employ `tf-idf`, stemming, stop-word and non-alphabetic word removal [101].

Filtering developers for classifier training. Anvik et al. refine the set of training reports by using several heuristics. For example, they do not consider developers who fixed a small number of bugs, which helps remove noise from the TDS. Although this is an effective way to filter non-experts from the training data and improve accuracy, in our approach filtering is unnecessary: the ranking function is designed such that, if there are two developers A and B who have fixed bugs of the same class K, but the number of K-type bugs A has fixed is greater than the number of K-type bugs B has fixed, a K-type bug will be assigned to A.

Multi-feature Tossing Graphs

With the training data and classifier at hand, we proceed to constructing tossing graphs as explained in Section 3.3.3. We use the same bug reports used for classification to build the tossing graphs.

Filtering developers for building tossing graphs. We do not prune the tossing graphs based on a pre-defined minimum support (frequency of contribution) for a developer, or the minimum number of tosses between two developers. Jeong et al. [76] discuss the significance of removing developers who fixed less than 10 bugs and pruning edges between developers that have less than 15% transaction probability. Since their approach uses the probability of tossing alone to rank neighboring developers, they need the minimum support values to prune the graph. In contrast, the multiple features in our tossing graphs coupled with the ranking function (as explained in the Section 3.3.6) obviate the need for pruning.

Predicting Developers

For each bug, we predict potential developers using two methods: (1) using the classifier alone, to demonstrate the advantages of incremental learning, and (2) using both the classifier and tossing graphs, to show the significance of multi-feature tossing graphs. When using the classifier alone, the input consists of bug keywords, and the classifier returns a list of developers ranked by relevance; we select the top five from this list. When using the classifier in conjunction with tossing graphs, we select the top three developers from this list, then for developers ranked 1 and 2 we use the tossing graph to recommend a potential

tossee, similar to Jeong et al. For predicting potential tossees based on the tossing graph, our tossee ranking function takes into account multiple factors, in addition to the tossing probability as proposed by Jeong et al. In particular, our ranking function is also dependent on (1) the product and component of the bug, and (2) the last activity of a developer, to filter retired developers. Thus our final list of predicted developers contains five developer id's in both methods (classifier alone and classifier + tossing graph).

Folding

After predicting developers, similar to the Bettenburg et al.'s folding technique [12], we iterate the training and validation for all folds. However, since our classifier and tossing graph updates are already performed during validation, we do not have to update our training data sets after each fold validation. To maintain consistency in comparing our prediction accuracies with previous approaches, we report the average prediction accuracy over each fold.

3.4 Results

3.4.1 Experimental Setup

We used Mozilla and Eclipse bugs to measure the accuracy of our proposed algorithm. We analyzed the entire life span of both applications. For Mozilla, our data set ranges from bug number 37 to 549,999 (May 1998 to March 2010). For Eclipse, we considered bugs numbers from 1 to 306,296 (October 2001 to March 2010). Mozilla and Eclipse bug reports have been found to be of high quality [76], which helps reduce noise

when training the classifiers. We divided our bug data sets into 11 folds and executed 10 iterations to cover all the folds.

We used the bug reports to collect four kinds of data:

1. Keywords: we collect keywords from the bug title, bug description and comments in the bug report.

2. Bug source: we retrieve the product and component the bug has been filed under from the bug report.

3. Temporal information: we collect information about when the bug has been reported and when it has been been fixed.

4. Developers assigned: we collect the list of developer IDs assigned to the bug from the activity page of the bug and the bug routing sequence.

3.4.2 Prediction Accuracy

In Tables 3.5 and 3.6 we show the results for predicting potential developers who can fix a bug for Mozilla and Eclipse using five classifiers: Naïve Bayes, Bayesian Networks, C4.5, and SVM using Polynomial and RBF kernel functions. In our experiments, we used the classifier implementations in Weka for the first three classifiers [165] and WLSVM for SVM [45].[9]

Classifier alone. To demonstrate the advantage of our fine-grained, incremental learning approach, we measure the prediction accuracy of the classifier alone; column

[9]The details of the parameters used for the classifiers in the experiments can be found at: `http://www.cs.ucr.edu/~neamtiu/bugassignment-params/`

ML algorithm (classifier)	Selection	ML only (avg)	ML + Tossing Graphs (average prediction accuracy for VDS fold) 2	3	4	5	6	7	8	9	10	11	Avg. across all folds	Improv. vs prior work [76]
	Top 1	**27.67**	13.33	20.67	22.25	25.39	24.58	30.09	30.05	33.61	35.83	40.97	**27.67**	-
Naïve	Top 2	**42.19**	39.14	44.59	47.72	49.39	52.57	57.36	59.46	62.37	64.99	67.23	**54.49**	8.16
Bayes	Top 3	**54.25**	51.34	62.77	66.15	57.50	63.14	61.33	64.65	77.54	71.76	74.66	**65.09**	11.02
	Top 4	**59.13**	64.20	75.86	79.57	70.66	69.11	69.84	67.68	82.87	68.77	69.71	**71.82**	6.93
	Top 5	**65.66**	74.63	77.69	81.12	79.91	76.15	72.33	75.76	83.62	78.05	79.47	**77.87**	9.22
	Top 1	**26.71**	13.54	14.16	20.21	22.05	25.16	28.47	32.37	35.1	37.11	38.94	**26.71**	-
Bayesian	Top 2	**44.43**	36.98	38.9	37.46	40.89	43.53	48.18	51.7	54.29	57.57	60.43	**46.99**	7.24
Network	Top 3	**49.51**	47.19	49.45	46.42	51.42	53.82	49.59	53.63	59.26	61.91	63.9	**53.65**	2.27
	Top 4	**58.37**	54.31	57.01	54.77	59.88	61.7	63.47	62.11	67.64	68.81	66.08	**61.59**	8.07
	Top 5	**62.19**	59.22	59.44	61.02	68.29	64.87	68.3	71.9	76.38	77.06	78.91	**68.54**	10.78
	Top 1	**25.46**	10.8	14.2	18.3	26.21	24.85	28.77	30.7	32.29	33.64	34.87	**25.46**	
	Top 2	**31.03**	29.17	34.16	40.34	45.92	51.67	56.35	59.41	62.04	65.26	69.49	**51.38**	
C4.5	Top 3	**38.97**	33.2	38.39	43.37	51.05	56.47	62.68	66.44	69.92	73.41	75.62	**57.05**	N/A
	Top 4	**46.43**	41.16	46.15	51.05	59.16	64.56	69.43	73.4	76.31	80.52	83.84	**64.72**	
	Top 5	**59.18**	47.04	50.49	56.67	64.25	69.07	74.68	78.74	80.37	81.59	84.82	**68.77**	
SVM	Top 1	**23.82**	8.26	13.01	18.54	20.69	22.97	27.14	29.46	32.36	32.69	33.08	**23.82**	
(Polynomial	Top 2	**28.66**	18.94	21.49	26.63	31.29	31.81	37.24	36.87	40.24	43.47	48.06	**33.6**	
Kernel	Top 3	**34.04**	23.85	23.11	27.44	32.46	39.11	32.52	41.92	44.62	45.37	48.85	**35.93**	N/A
Function,	Top 4	**43.92**	26.74	30.78	35.99	28.82	34.77	40.05	46.89	53.47	59.03	63.7	**42.02**	
Degree=2)	Top 5	**51.17**	34.83	32.85	41.14	44.4	46.94	53.76	60.3	62.69	61.01	70.95	**50.89**	
SVM	Top 1	**30.98**	17.37	20.27	28.56	30.46	31.98	34.86	31.56	38.69	33.09	42.97	**30.98**	
(RBF	Top 2	**39.27**	41.51	42.4	49.1	53.02	52.04	59.33	59.24	62.13	66.1	68.29	**55.32**	
Kernel	Top 3	**45.52**	43.7	44.26	49.6	53.96	61.69	54.79	62.79	66.82	67.25	69.75	**57.38**	N/A
Function)	Top 4	**53.42**	48.21	51.51	57.15	51.62	56.95	61.08	67.63	74.23	80.59	84.12	**63.31**	
	Top 5	**62.49**	56.07	53.99	62.2	66.13	68.54	76.23	80.74	84.69	83.04	82.71	**71.43**	

Table 3.5: Bug assignment prediction accuracy (percents) for Mozilla.

ML algorithm (classifier)	Selection	ML only (avg)	ML + Tossing Graphs (average prediction accuracy for VDS fold) 2	3	4	5	6	7	8	9	10	11	Avg. across all folds	Improv. vs prior work [76]
Naïve Bayes	Top 1	**32.35**	12.2	21.09	24.7	23.43	25.17	33.04	38.73	42.03	49.59	53.69	**32.36**	-
	Top 2	**48.19**	39.53	38.66	36.03	39.16	39.29	41.82	43.2	47.94	51.65	54.18	**43.15**	5.99
	Top 3	**54.15**	47.95	50.84	48.46	49.52	59.45	62.77	61.73	68.19	74.95	69.07	**59.30**	2.76
	Top 4	**58.46**	56.29	61.16	59.88	60.81	69.64	69.37	75.64	75.3	78.22	77.31	**68.37**	6.69
	Top 5	**67.21**	66.73	69.92	74.13	77.03	77.9	81.8	82.05	80.63	82.59	81.44	**77.43**	5.98
Bayesian Network	Top 1	**38.03**	24.36	29.53	31.04	36.37	34.09	40.97	40.22	43.99	48.88	50.85	**38.03**	-
	Top 2	**41.43**	36.11	41.49	41.13	44.81	46.34	47.4	48.61	53.84	59.18	63.69	**48.26**	3.97
	Top 3	**59.50**	51.16	52.8	54.62	57.38	56.39	63.26	66.68	70.34	76.72	77.34	**62.67**	8.88
	Top 4	**62.72**	62.92	59.03	63.09	68.27	68.33	71.79	73.37	74.15	76.94	77.04	**69.50**	5.58
	Top 5	**68.91**	74.04	72.41	70.92	71.52	73.5	75.61	79.28	79.68	80.61	81.38	**75.89**	6.93
C4.5	Top 1	**28.97**	11.43	21.35	24.88	28.33	25.12	30.56	31.57	35.19	38.37	42.97	**28.97**	
	Top 2	**36.33**	31.07	37.65	42.24	48.23	51.75	55.54	58.13	59.44	62.61	62.98	**50.96**	
	Top 3	**48.17**	37.95	44.47	48.29	55.82	58.45	62.73	65.28	66.32	69.34	69.57	**57.82**	N/A
	Top 4	**54.62**	44.62	51.11	55.36	61.47	65.62	69.3	71.06	72.39	75.23	76.44	**64.26**	
	Top 5	**65.98**	51.27	57.15	62.44	68.52	71.77	75.95	78.51	79.64	82.36	86.09	**71.37**	
SVM (Polynomial Kernel Function, Degree=2)	Top 1	**22.45**	9.43	13.3	15.59	20.12	24.6	24.65	26.46	30.12	31.71	29.93	**22.45**	
	Top 2	**26.52**	19.51	21.4	27.1	32.02	31.04	37.33	37.24	40.13	44.1	47.29	**33.72**	
	Top 3	**30.08**	21.7	23.26	27.6	32.96	39.69	32.79	41.79	48.11	45.25	50.75	**36.39**	N/A
	Top 4	**33.17**	26.21	30.51	35.15	29.62	34.95	40.08	46.63	53.23	59.59	63.12	**41.91**	
	Top 5	**42.92**	35.07	32.99	41.2	45.13	46.54	54.23	59.74	62.69	61.04	71.71	**51.03**	
SVM (RBF Kernel Function)	Top 1	**29.23**	16.54	22.06	22.29	28.29	26.58	31.86	31.48	33.84	36.01	43.18	**29.21**	
	Top 2	**37.5**	38.4	42.85	43.39	44.41	47.2	46.98	48.29	49.49	48.8	46.68	**45.65**	
	Top 3	**47.04**	46.65	54.17	51.1	55.66	61.41	62.66	66.63	72.46	68.87	72.93	**61.25**	N/A
	Top 4	**53.46**	48.64	49.76	55.96	54.18	59.76	64.61	70.32	74.43	74.83	78.67	**63.12**	
	Top 5	**64.77**	63.5	63.68	59.9	69.52	71.98	75.97	78.24	83.33	80.38	82.02	**72.85**	

Table 3.6: Bug assignment prediction accuracy (percents) for Eclipse.

"ML only" contains the classifier-only average prediction accuracy rate. We found that, for Eclipse and Mozilla, our approach increases accuracy by 8.91 percentage points on average compared to the best previously-reported, no-incremental learning approach, by Anvik et al. [8]. This confirms that incremental learning is instrumental for achieving a high prediction accuracy. Anvik et al. report that their initial investigation of incremental learning did not yield highly accurate predictions, though no details are provided. Note that we use different data sets (their experiments are based on 8,655 reports for Eclipse and 9,752 for Firefox, while we use 306,297 reports for Eclipse and 549,962 reports for Mozilla) and additional attributes for training and validation.

Classifier + tossing graphs. Columns "ML+Tossing Graphs" of Tables 3.5 and 3.6 contain the average accurate predictions for each fold (Top 2 to Top 5 developers) when using both the classifier and the tossing graph; the Top 1 developer is predicted using the classifier only. Consider row 2, which contains prediction accuracy results for Top 2 in Mozilla using the Naïve Bayes classifier: column 4 (value 39.14) represents the percentage of correct predictions for fold 1; column 5 (value 44.59) represents the percentage of correct predictions for folds 1 and 2; column 14 (value 54.49) represents the average value for all iterations across all folds. Column 15 represents the percentage improvement of prediction accuracy obtained by our technique when compared to using tossing graphs with tossing probabilities only. Our best average accuracy is achieved using Naïve Bayes (77.87% for Mozilla and 77.43% for Eclipse). We found that this prediction accuracy is higher than the prediction accuracy we obtained in our earlier work [17] where we used Naïve Bayes and Bayesian Networks only. When compared to prior work [76] (where Naïve

Bayes and Bayesian Networks were used as ML algorithms and tossing probabilities alone were used in the tossing graphs) our technique improved prediction accuracy by up to 11.02 percentage points. However, when measuring across the average of all ten folds, our model achieved highest prediction accuracy of 77.87% for Mozilla using Naïve Bayes and 75.89% for Eclipse using Bayesian Networks. The last column shows the percentage increase in prediction accuracy from using single-attribute tossing graphs with tossing probability alone [76] compared to our approach in which we used a ranking function based on the multi-attribute tossing graphs we proposed.

Classifier selection. In Section 3.3.1 we discussed that one of the objectives of using a broad range of classifiers for evaluating our framework is to analyze if a particular classifier is best suited for the bug assignment problem. Our results in Tables 5 and 6 reveal that the answer is complex. Generally, Naïve Bayes works best for early VDS folds (when there are fewer data) and when considering Top 4 or Top 5 accuracies. The polynomial-kernel SVM performs fairly poorly. The other three are comparable, without an obvious pattern.

Our results are consistent with the standard statistical learning theory of bias-variance [66]. In particular, with fewer data (or more noise in the data) better results are achieved by using a less flexible classifier (one with fewer parameters and more bias). This supports the performance of Naïve Bayes: it does better for small sample sizes and in case where the testing metric does not match the training metric as well (Top 5, for instance) which looks like noisier data. Additionally, if the bias is too far from the true answer, the method will not work well. The polynomial-kernel SVM probably has such a mismatch: its

bias is too far from the correct bug triage classifier. In particular, it is a global classifier in that all training data affect the classifications for all inputs. By contrast, C4.5 and RBF SVM both are local classifiers: only training data near the testing point have a large influence on the resulting classification. This suggests that local classification methods will do best on bug assignment.

Among the more flexible (less biased) local classifiers (Bayesian networks, C4.5, and RBF SVM), there is not a clear winner—all seem equally well suited for bug assignment. On any particular task, one will do better than the others, but a systematic prediction about other tasks cannot be made from these experiments: much will depend on the amount of data, and the noise present. All of these methods have "regularization parameters" that can adjust the amount of bias. Picking a suitable value based on the amount of data and noise is more important for achieving good results than the exact classifier used.

3.4.3 Tossing Length Reduction

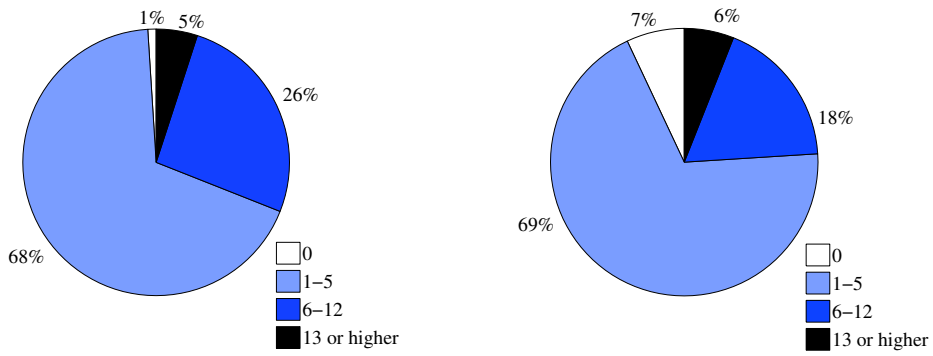
We compute the original tossing path lengths for "fixed" bugs in Mozilla and Eclipse, and present them in Figure 3.8; we observe that most bugs have tossing length less than 13 for both applications. Note that tossing length is zero if the first assigned developer is able to resolve the bug. Ideally, a bug assignment model should be able to recommend bug fixers such that tossing lengths are zero. However, this is unlikely to happen in practice due to the unique nature of bugs. Though Jeong et al. measured tossing lengths for both "assigned" and "verified" bugs, we ignore "assigned" bugs because they are still open, hence we do not have ground truth (we do not know the final tossing length

yet). In Figure 3.9, we present the average reduced tossing lengths of the bugs for which we could correctly predict the developer. We find that the predicted tossing lengths are reduced significantly, especially for bugs which have original tossing lengths less than 13. Our approach reports reductions in tossing lengths by up to 86.67% in Mozilla and 83.28% in Eclipse. For correctly-predicted bugs with original tossing length less than 13, prior work [76] has reduced tossing path lengths to 2–4 tosses, while our approach reduces them to an average of 1.5 tosses for Mozilla and 1.8 tosses for Eclipse, hence multi-feature tossing graphs prove to be very effective.

3.4.4 Filtering Noise in Bug Reports

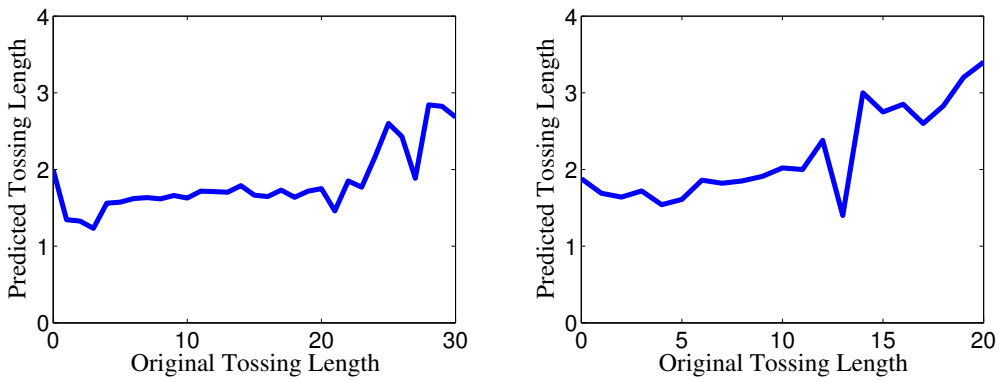
We found that when training sets comprise bugs with resolution "verified" or "resolved" and arbitrary status, the noise is much higher than when considering bugs with resolution "verified" or "resolved" and status "fixed". In fact, we found that, when considering arbitrary-status bugs, the accuracy is on average 23% lower than the accuracy attained when considering fixed-status bugs only. Jeong et al. considered all bugs with resolution "verified" and arbitrary-status for their training and validation purposes. They found that tossing graphs are noisy, hence they chose to prune developers with support less than 10 and edges with transaction probability less than 15%.

Our analysis suggests that bugs whose status changes from "new" or "open" to "fixed" are actual bugs which have been resolved, even though various other kinds of bugs, such as "invalid," "works-for-me," "wontfix," "incomplete" or "duplicate" may be categorized as "verified" or "resolved." We conjecture that developers who submit patches are



(a) Mozilla (b) Eclipse

Figure 3.8: Original tossing length distribution for "fixed" bugs.



(a) Mozilla (b) Eclipse

Figure 3.9: Average reduction in tossing lengths for correctly predicted bugs when using ML + Tossing Graphs (using both classifiers).

more competent than developers who only verify the validity of a bug and mark them as "invalid" or developers who find a temporary solution and change the bug status to "works-for-me." Anvik et al. made a similar distinction between message repliers and contributors/maintainers; they found that only a subset of those replying to bug messages are actually submitting patches and contributing to the source code, hence they only retain the

contributing repliers for their TDS.

3.4.5 Importance of Individual Tossing Graph Attributes

Since our ranking function for tossing graphs contains additional attributes compared to the original tossing graphs by Jeong et al., we were interested in evaluating the importance of each attribute using ablative analysis as described in Section 3.3.4. Therefore, we compute, for each fold, the reduction in accuracy caused by removing one attribute from the ranking function and keeping the other two. In Figure 3.10 we show the minimum (bottom black bar), maximum (top black bar) and average (red bar) across all folds. The decrease in prediction accuracy shows that the removal of product and developer activity attributes affects the prediction accuracy the most. These accuracy reductions underline the importance of using all attributes in the ranking function, and more generally, the advantage of the richer feature vectors our approach relies on. Note that removing developer activity affects prediction accuracy in Mozilla more significantly than in Eclipse. Analyzing the significance of each attribute in our ranking function for individual projects, i.e., build a ranking function per project, is beyond the scope of this chapter.

3.4.6 Importance of Incremental Learning

To assess the significance of incremental learning in our technique, we performed two sets of experiments. We took our best results, i.e., using Naïve Bayes classifier, with tossing graphs, product–component and incremental learning, and then unilaterally varied

Project	Selection	Average Prediction Accuracy (%)		
		With intra- and inter-fold updates (best)	Without intra-fold updates	Without inter-fold updates
Mozilla	Top 1	27.67	13.53 (-14.15)	7.86 (-19.82)
	Top 2	54.49	26.59 (-27.90)	12.54 (-41.95)
	Top 3	65.09	47.20 (-17.88)	28.61 (-36.48)
	Top 4	71.82	53.24 (-18.60)	36.63 (-35.2)
	Top 5	77.87	62.22 (-15.66)	43.86 (-34.02)
Eclipse	Top 1	32.36	8.64 (-23.73)	11.43 (-20.94)
	Top 2	43.15	19.18 (-23.97)	16.02 (-27.13)
	Top 3	59.30	32.82 (-26.48)	27.15 (-32.15)
	Top 4	68.37	44.30 (-24.07)	32.49 (-35.88)
	Top 5	77.43	58.33 (-19.10)	39.47 (-37.96)

Table 3.7: Impact of inter- and intra-folding on prediction accuracy using the Naïve Bayes classifier.

the learning procedure.[10] The best-result data has been shown in Tables 3.5 and 3.6 but for ease of comparison we report the same data in shown column 3 of Table 3.7.

Intra-fold updates. To evaluate the impact of disabling of intra-fold updates we also trained our model using folds 1 to $(N-1)$ and we used fold N for prediction. The results of the average prediction accuracy are presented in column 4 of Table 3.7. For example, our results show that for Top 1 developers in Mozilla, the average prediction accuracy across 10 folds is 13.53%, a decrease of 14.15 percentage points when compared to the incremental learning (inter- and intra-fold updates) technique shown in column 3.

Inter-fold updates. To evaluate the importance of inter-fold updates for each fold, we first trained our model using the first 80% of the bug reports in that fold only. Next, we used the remaining 20% of the bug reports in that fold only for measuring prediction

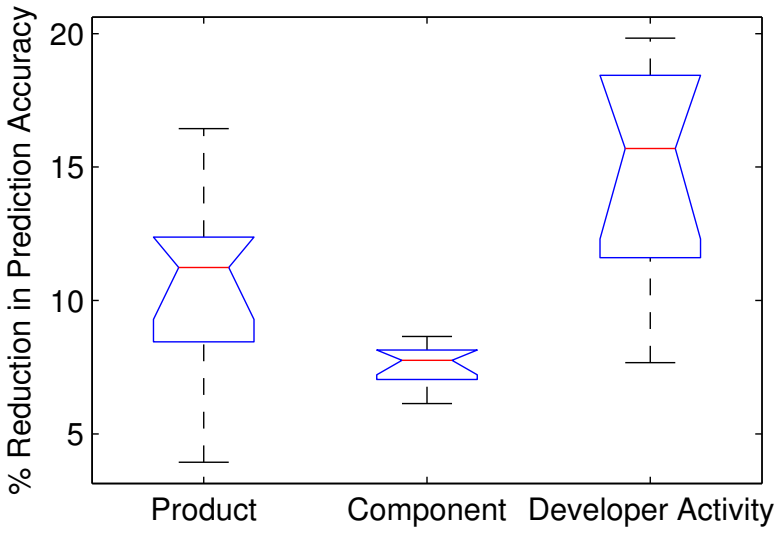
[10]We chose Naïve Bayes since the average prediction accuracy was highest for this classifier compared to other classifiers, hence, consistent with the standard machine learning practice of ablative analysis, we varied incremental learning to quantify its impact.

accuracy. Note that in this case, the bug reports from folds 1 to N-1 are not added to fold N while training the classifier. The average prediction accuracy is presented in column 5 of Table 3.7. For example, our results show that for Top 1 developers in Mozilla, the average prediction accuracy across 10 folds is 7.86%, a decrease of 19.82% when compared to the incremental learning (inter- and intra-fold updates) technique shown in column 3.

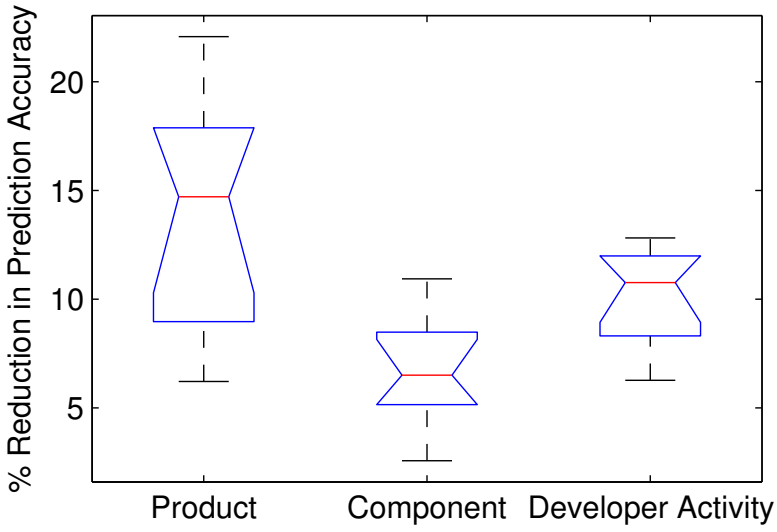
Conclusions. The results in Table 3.7 suggest there is a significant decrease in prediction accuracy (up to 42%) when incremental learning (inter- and intra-fold updates) is removed from our algorithm. This reduction in prediction accuracy suggests that indeed incremental learning is instrumental to achieving higher prediction accuracy for bug assignment: inter-and intra-folding lead to tossing graphs with highly accurate transaction probabilities which, helps improve our prediction accuracy. Note that incremental learning (or folding) is not a contribution of our work; incremental learning is a standard technique to improve the prediction accuracy in any supervised or unsupervised learning algorithms in machine learning [85]. Rather, these experiments were performed to demonstrate that in comparison to prior work, where split-sample validation was used, automatic bug assignment can benefit significantly from incremental learning.

3.4.7 Accurate Yet Efficient Classification

One of the primary disadvantages of fine-grained incremental learning is that it is very time consuming. As described in Section 3.3.5, we performed a study to find how many past bug reports we need to train the classifier to achieve approximately similar prediction accuracy when compared to the highest prediction accuracy attained when using



(a) Mozilla



(b) Eclipse

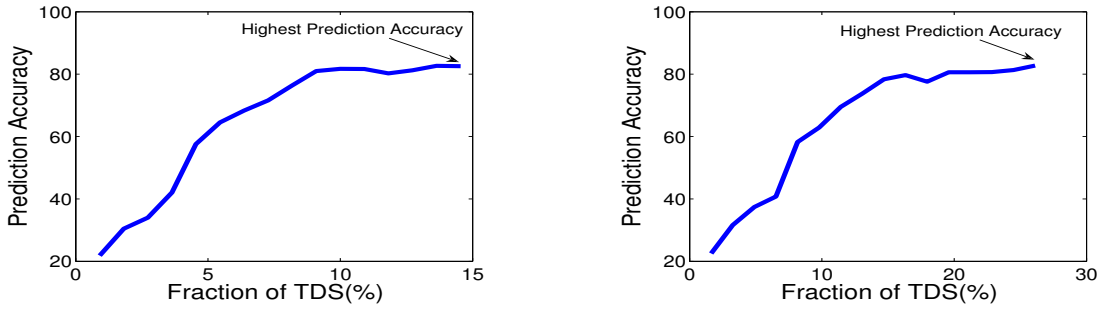
Figure 3.10: Impact of individual ranking function attributes on prediction accuracy.

folds 1–10 as the TDS and fold 11 as the VDS. We used the Naïve Bayes classifier as our ML algorithm in this case. We present our results in Figure 3.11. We found that Mozilla required approximately 14% and Eclipse required about 26% of all bug reports (in reverse chronological order, i.e., most recent bugs) to achieve prediction accuracies greater than 80%—within 5 percentage points of the best results of our original experiments where we used the complete bug history to train our classifier. Therefore, a practical way to reduce the computational effort associated with learning, yet maintain high prediction accuracy, is to prune the bug report set and only use a recent subset (e.g., the most recent 14% to 26% of bug reports, depending on the project).

Computational effort. The intra-fold updates used in our approach are more computationally-intensive than inter-fold updates. However, for practical purposes this is not a concern because very few bugs get fixed the day they are reported. Before we use the algorithm to predict developers, we train it with all fixed bug reports in the history; when a new bug gets fixed, the TDS needs to be updated and we need to re-train the classifier. However, while about 100 bugs are reported every day for large projects like Mozilla and Eclipse, less than 1 bug gets fixed every day, on average [76]. Since we use fixed bug reports only, if we update the TDS overnight with the new fixed bug reports and retrain the classifier, we can still achieve high prediction accuracies.

3.5 Threats To Validity

We now present possible threats to the validity of this chapter's work.



(a) Mozilla

(b) Eclipse

Figure 3.11: Change in prediction accuracy when using subsets of bug reports using Naïve Bayes classifier.

3.5.1 Internal Validity

In our study we collected bug reports from Bugzilla for both Eclipse and Mozilla. Bug reports can have various status at a given point in time: "unconfirmed," "new," "assigned," "reopened," "resolved," "verified," and "closed". A bug which has status resolution status as "fixed" can be either "verified" or "closed" at a given point. For our training and validation purposes, we look at bugs which have the resolution status as fixed irrespective of whether it is "verified" or "closed". We filter our data set to fixed bugs only for the following reasons: (1) for bugs which are unconfirmed, it is not possible to say if they are indeed bugs, (2) for new bugs it is not known who the developer will be who will fix that bug and hence these bugs cannot be used for training a supervised classifier where the end-result knowledge is necessary, (3) reopened bugs are similar to new bugs and hence are not a part of our training/validation, (4) resolved bugs are those for which a resolution has been provided by a developer but is still in the review process which implies that the bug might be re-assigned (or tossed) if the resolution is not satisfactory. For accurate supervised

learning, we need to ensure that the training set includes the correct expertise of the developers. One potential threat to validity in our study is that a bug B which has been fixed and closed can be reopened at a later time. In that case developer D who earlier resolved bug B might not resolve the issues with reopening the bug again and might affect our classification results. However, it is impossible to predict what percentage of currently-fixed bugs will be reopened in future and quantify the effects of bug reopening on our results. Another potential threats to validity in our study is not differentiating between bugs and enhancement requests.

3.5.2 External Validity

Generalization to other systems. The high quality of bug reports found in Mozilla and Eclipse [76] facilitates the use of classification methods. However, we cannot claim that our findings generalize to bug databases for other projects. Additionally, we have validated our approach on open source projects only, but commercial software might have different assignment policies and we might require considering different attribute sets.

Small projects. We used two large and widely-used open source projects for our experiments, Mozilla and Eclipse. Both projects have multiple products and components, hence we could use this information as attributes for our classifier and labels in our tossing graphs. For comparatively smaller projects which do not have products or components, the lack of product-component labels on edges would reduce accuracy. Additionally, for smaller projects the 90-days heuristic we use for pruning inactive developers might have to change. In the future when we analyze smaller projects, we plan to empirically study the

average lifetime of a developer for the project to determine inactive and active developers. Nevertheless, fine-grained incremental learning and pruning inactive developers would still be beneficial.

3.5.3 Construct Validity

For the projects we used, we did not differentiate between various roles (e.g., developers, triagers, managers) contributors serve in the project. Our approach neither divides contributors according to the roles they play in the community, nor ranks them higher based on their familiarity with the source code. In the future, we plan to include developer's source code expertise in the future to further improve our ranking function. Additionally, it is not possible to find out in our framework if the first developer who was assigned the bug was a default assignee or assigned by the triager explicitly for any projects. However, for the projects we chose—Mozilla and Eclipse—developers were cc'ed by default when they are responsible for a specific product or component, but they are not assigned the bug by default for fixing it.

3.5.4 Content Validity

Information retrieval and learning tools. We used Weka for extracting relevant keywords after stop-word removal and tf-idf as explained in Section 3.3.6. We also used the built-in classifiers of Weka and LibSVM for learning our model. Hence, another potential threat to validity is error in these tools or how changes in implementation of these classifiers might affect our results.

Developer identity. The assignee information in Bugzilla does not contain the domain info of the email address for a developer. Therefore, we could not differentiate between users with same email id but different domains. For instance, in our technique, bugzilla@alice.com, and bugzilla@bob.com will be in the same bucket as bugzilla@standard8.plus.com. This might potentially lead to inaccurate predictions and decrease the prediction accuracy of our model.

Load balancing. Our technique does not consider load balancing while assigning bugs to developers. This is a potential threat to validity in the following sense: if our approach predicts that developer D is the best match to fix a bug, he/she might be overloaded, so assigning them another bug might increase the bug-fix time.

3.6 Contribution Summary

In summary, the main contributions of this chapter are:

- We employ a comprehensive set of machine learning tools and a probabilistic graph-based model (bug tossing graphs) that lead to highly-accurate predictions, and lay the foundation for the next generation of machine learning-based bug assignment.
- Our work is the first to examine the impact of multiple machine learning dimensions (classifiers, attributes, and training history) along with bug tossing graphs on prediction accuracy in bug assignment.
- We validate our approach on Mozilla and Eclipse, covering 856,259 bug reports and 21 cumulative years of development. We demonstrate that our techniques can achieve

up to 86.09% prediction accuracy in bug assignment and significantly reduce tossing path lengths.

- We show that for our data sets the Naïve Bayes classifier coupled with product–component features, tossing graphs and incremental learning performs best. Next, we perform an ablative analysis by unilaterally varying classifiers, features, and learning model to show their relative importance of on bug assignment accuracy. Finally, we propose optimization techniques that achieve high prediction accuracy while reducing training and prediction time.

3.7 Conclusions

Machine learning and tossing graphs have proved to be promising for automating bug assignment. In this chapter we lay the foundation for future work that uses machine learning techniques to improve automatic bug assignment by examining the impact of multiple machine learning dimensions—learning strategy, attributes, classifiers—on assignment accuracy.

We used a broad range of text classifiers and found that, unlike many problems which use specific machine learning algorithms, we could not select a specific classifier for the bug assignment problem. We show that, for bug assignment, computationally-intensive classification algorithms such as C4.5 and SVM do not always perform better than their simple counterparts such as Naïve Bayes and Bayesian Networks. We performed an ablative analysis to measure the relative importance of various software process attributes

in prediction accuracy. Our study indicates that to avoid the time-consuming classification process we can use a subset of the bug reports from the bug databases and yet achieve stable-high prediction accuracy.

Chapter 4

Effects of Programming Language on Software Development and Maintenance

Anecdotal evidence suggests that the choice of programming language before developing a software project is primarily determined by three factors: performance issues, developer expertise and task difficulty. In this chapter we argue that the choice of programming language affects the quality of software produced using one language over another. Existing studies that analyze the impact of choice of programming language on software quality suffer from several deficiencies with respect to methodology and the applications they consider. For example, they consider applications built by different teams in different languages, hence fail to control for developer competence, or they consider small-sized, infrequently-used, short-lived projects. We design a novel methodology which controls for

development process and developer competence, and quantifies how the choice of programming language impacts software quality and developer productivity. We conduct a study and statistical analysis on a set of long-lived, widely-used, open source projects—Firefox, Blender, VLC, and MySQL. The key novelties of our study are: (1) we only consider projects which have considerable portions of development in two languages, C and C++, and (2) a majority of developers in these projects contribute to both C and C++ code bases. We found that using C++ instead of C results in improved software quality and reduced maintenance effort, and that code bases are shifting from C to C++. Our methodology lays a solid foundation for future studies on comparative advantages of particular programming languages.

4.1 Introduction

We are currently witnessing a shift in the language choice for new applications: with the advent of Web 2.0, dynamic, high-level languages are gaining more and more traction [40, 155]; these languages raise the level of abstraction, promising faster development of higher-quality software. However, the lack of static checking and the lack of mature analysis and verification tools makes software written in these languages potentially more prone to error and harder to maintain, so we need a way to quantitatively assess whether they indeed improve development and maintenance.

To that end, in this chapter we present a methodology for assessing the impact of

The work presented in this chapter have been published in the proceedings of the 2011 IEEE International Conference on Software Engineering [14].

programming language on development and maintenance, a long-standing challenge [108]. We first introduce an approach for attributing software quality and ease of maintenance to a particular programing language, then exemplify the approach by comparing C and C++. C++ was designed extending C to include features—object-oriented constructs, overloading, polymorphism, exception handling, stronger typing—aimed at faster construction of less error-prone software. To understand whether using C++ instead of C leads to better, easier to maintain software, we answer several questions directly related to software construction and maintenance: Are programs written in C++ easier to understand and maintain than programs written in C? Are C++ programs less prone to bugs than C programs? Are seasoned developers, with equal expertise in C and C++, more productive in C++ than in C? Are code bases shifting from C to C++?

We answer these questions via an empirical study; we are now in a good position to conduct such a study because both C and C++ are mature, and have been used in large projects for a time long enough to study the effects of using one language versus the other.

Prior efforts on analyzing the impact of choice of programming language suffer from one or more deficiencies with respect to the applications they consider and the manner they conduct their studies: (1) they analyze applications written in a combination of two languages but these applications are *small-sized* and have *short lifespans*, or (2) they consider software built entirely using a *single* language, rather than performing a cross-language evaluation, or (3) they examine applications that are written in *different languages* by *different teams*. Using such methodologies often results in analyses which cannot be generalized to large real-world applications. We aimed to address all these shortcomings.

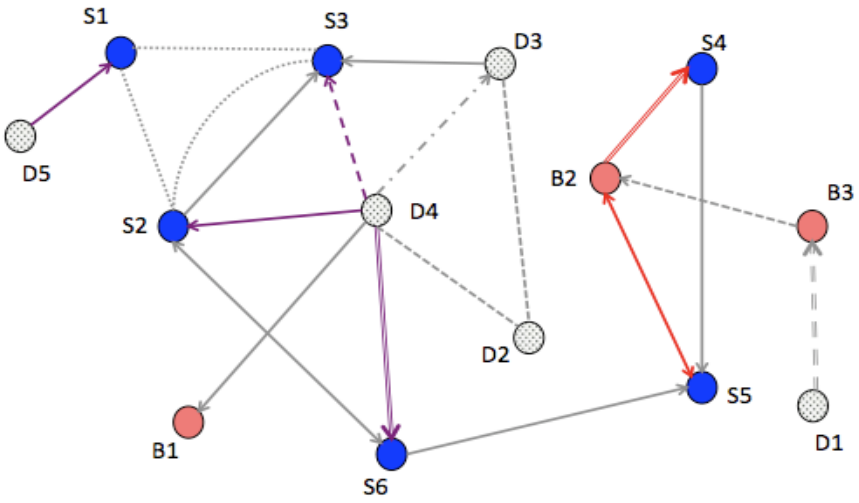


Figure 4.1: Hypergraph extraction for assessing the effects of programming language on software evolution.

First, we consider four large, long-lived, open source applications: Mozilla Firefox, VLC Media Player, Blender Animation Software and MySQL; our analysis covers 216 official releases and a combined 40 years of evolution. All these applications are mature, stable, have large code bases in both C and C++, and have large user bases; their long histories help us understand issues that appear in the evolution of multi-developer widely-used software.

Second, we ensure the uniformity of software development process when comparing C and C++ code. Prior work has compared languages by considering applications written exclusively in a single language, e.g., by implementing the same small task in C, C++, Fortran, or Visual Basic [2, 129, 71, 93, 77]. We only studied projects that contain *both* C and C++ code, to guarantee uniformity in the development process of the application.

Third, we effectively control for developer competence to ensure that changes to software facets, e.g., quality, can be attributed to the underlying programming language. We use a statistical analysis of committer distribution to show that the majority of devel-

opers contribute to both C and C++ code bases (Section 4.3.1); we confirm this with the developers as well (Section 6.6).

We present our research hypotheses in Section 4.2, followed by data collection and statistical methodology in Section 5.2. In Figure 4.1 we show the nodes and edges we extract from the multi-mixed graph (explained in Section 2) for our analysis in understanding the effects of programming language on software evolution. Formally, we can represent the hypergraph extracted as shown in Figure 4.1 as:

$$\{(v_1, v_2) | (v_1, v_2) \in \mathbb{G}_{InterRepoDep} | v_1 \in (v_{func} \vee v_{mod}) \wedge v_2 \in (v_{bug} \vee v_{contributor})\}$$

We first investigated whether code bases are shifting from C to C++ and found that this shift occurs for all but one application (Section 4.4.1). We then compared internal qualities for code bases in each language and could confirm that C++ code has higher internal quality than C code (Section 4.4.2). We found the same trend for external quality, i.e., that C++ code is less prone to bugs than C code (Section 4.4.3). Finally, we found that C++ code takes less maintenance effort than C code (Section 4.4.4).

To our knowledge, this is the first study that compares programming languages while controlling for variations in both developer expertise and development process, and draws *statistically significant* conclusions.

4.2 Research Hypotheses

Our study is centered around four research hypotheses designed to determine whether C++ (a higher-level programming language) produces better software than C (a

lower-level language):

H1: C++ is replacing C as a main development language

At the beginning of the development process for an application, the best-suited language is chosen as the primary language. Later on, developers might decide to replace the primary language, e.g., if the potential benefits of migrating to a new language outweigh the costs. Our hypothesis is that, as the advantages of C++ become apparent, applications that have started with C as their primary language are shifting to C++. To verify this, we measured the change in percentage of C and C++ code over an application's lifetime; if the C++ percentage increases over time, we can conclude that C is being replaced by C++.

H2: C++ code is of higher internal quality than C code

One of the trademarks of high-level languages is that they enable the construction of software that displays higher internal quality than software written in low-level language, i.e., software that is less complex, easier to understand and easier to change. To test this hypothesis, for each application, we computed normalized code complexities for C and C++ using several metrics. If the hypothesis held, we should observe that, on average, C++ code is less complex than C code.

H3: C++ code is less prone to bugs than C code

Software bugs are due to a variety of reasons, e.g., misunderstood requirements, programmer error, poor design. The programming language plays a key role in preventing bugs; for example, polymorphic functions can avoid code cloning and copy-paste errors, and strongly-typed language eliminate many potential runtime errors. We use this reasoning to postulate our next hypothesis: due to the higher-level features, C++ code is less bug-prone

than C code.

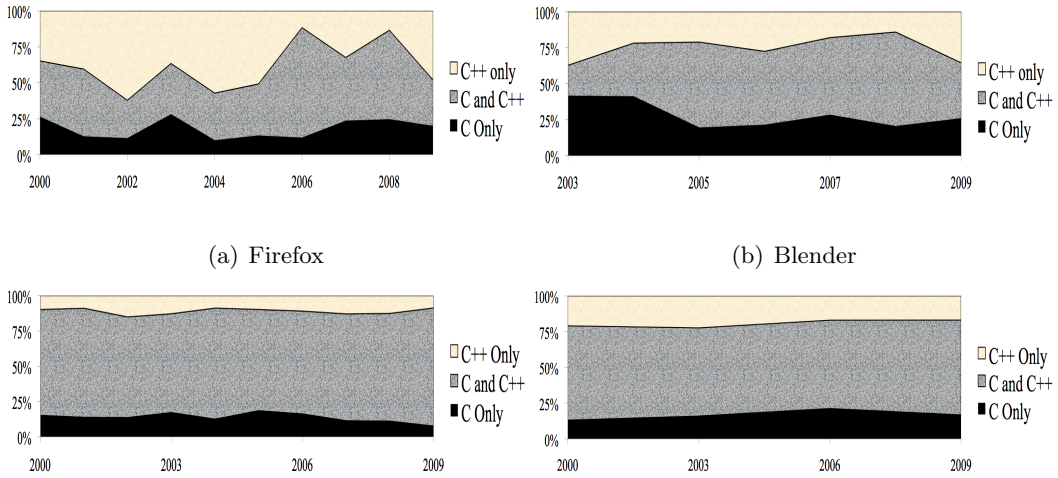
H4 : C++ code requires less effort to maintain than C code

Computing the effort that goes into software development and maintenance is difficult, especially for open-source projects, where the development process is less structured than in commercial settings [112]. Our findings indicate that even when there is no explicit allocation of tasks to developers, most developers contribute to both the C and C++ code bases. Our hypothesis is that the effort required to maintain and extend the C++ code base is lower than the effort associated with the C code base.

4.3 Methodology and Data Sources

We ran our empirical study on four popular open source applications written in a combination of C and C++, namely, Firefox, VLC, Blender and MySQL. [1] We used several criteria for selecting our test applications. First, since we are interested in long-term software evolution and pursue statistically significant results, the applications had to have a long release history. Second, applications had to be sizeable, so we can understand the issues that appear in the evolution of realistic, production-quality software. Third, the applications had to be actively maintained by a large number of developers. Fourth, the applications had to be used by a wide number of users who report bugs and submit patches.

[1] Details on these applications can be found in Chapter 2.



(a) Firefox

(b) Blender

(c) VLC

(d) MySQL

Figure 4.2: Committer Distribution.

4.3.1 Data Collection

We now describe our data collection methodology. We first checked out the source code of all official releases from the version control management systems the applications use, then collected file change histories, and finally extracted bug information from the application-specific bug databases.

Committer distribution. An explicit goal of our study was to look at C and C++ code that was part of the *same* project, to keep most factors of the software development process constant. One such factor is developer expertise; anecdotal evidence suggests that expertise greatly affects software quality [31]. Ideally, to understand the difference between the C and C++ languages, we need to study code written by developers who are proficient in both C and C++. In Figure 4.2 we plot the percentages of developers who contribute to C++ code base only (top area), C code base only (bottom area)

and to both C and C++ code bases (middle area). We observe that a large percentage of developers contribute to both C and C++ code. To verify that developers in the middle area commit in equal measures to both code bases, we selected random versions from each application. We then compared the mean values for the C commits and C++ commits for all those developers who commit to both code bases. We found that the mean values for C and C++ commits are comparable (using *Welch's* t-test as explained in Section 4.3.2), i.e., most developers commit in equal measures to both code bases. This ensures that we effectively control for developer competence, and any changes to software attributes (e.g., quality) can be attributed to the underlying programming language only. In Section 6.6 we present further evidence against selection bias, i.e., that perceived task difficulty and developer competence do not determine language choice.

Dividing source code into C and C++ groups. Identifying whether a file belongs to the C code base or the C++ code base is not trivial, because header files often use the extension ".h" for both C and C++ headers, while ".hpp" or ".hh" extensions are reserved for C++ headers. We considered a header file as a C++ header file *if and only if* all the files it is included in are C++ files; otherwise we consider it as a C header file. The implementation files were divided based on extension: ".c" for C files, and ".cpp" or ".cc" for C++ files.

Collecting file change histories. For testing hypotheses 3 and 4 we need precise information about bugs and code changes associated with each version. We obtain this information by analyzing change logs associated with source files, after dividing files into C and C++ groups. Note that it is not sufficient to extract change histories for files in the

last version only, because some files get deleted as the software evolves; rather, we need to perform this process for each version.

Accurate bug counting. We use defect density to assess external quality. Collecting this information is non-trivial, due to incomplete information in bug databases. As we explain shortly, to ensure accuracy, we cross-check information from bug databases[2] with bug information extracted from change logs. One problem arises from bugs assigned to no particular version; for instance, 33% of the fixed bugs in Firefox are not assigned to a specific Firefox version in the Bugzilla database. This problem is compounded in applications which exhibit parallel evolution, as the co-existence of two or more parallel development branches makes version assignment problematic. Another problem is that, often, for bug fixes that span several files, the bug databases report only a partial list of changed files. However, if we search for the bug ID in the change logs, we get the complete list of files that were changed due to a particular bug fix. Therefore, we used both bug databases and change logs as bug data sources. We used a two-step approach for bug counting. First, we searched for keywords such as "bug", "bugs", "bug fixes", and "fixed bug", or references to bug IDs in log files; similar methods have been used by other researchers for their studies [149, 50, 114]. Second, we cross-checked our findings from the log files with the information in the databases to improve accuracy, similar to techniques used by other researchers for computing defect density or fault prediction [80, 149]). With the bug information at hand, we then associate a certain bug to a certain version: we used release tags, dates the bug was reported, and commit messages to find the version in which the bug was reported

[2]Defect tracking systems vary: Firefox uses the Bugzilla database [32], Blender uses it own tracker [24], VLC uses Trac [156], and MySQL uses Bazaar [10] and Launchpad [90].

in, and we attributed the bug to the previous release.

Extracting effort information. To measure maintenance effort, we counted the number of commits and the churned eLOC [3] (sum of the added and changed lines of code) for each file for a release, in a manner similar to previous work by other researchers [121, 48]. This information is available from the log files.

4.3.2 Statistical Analysis

Variations in release frequency. Our applications have different release frequencies: Firefox, VLC, and Blender have pre-releases (alpha or beta) before a major release, while MySQL has major releases only. Differences in release frequency and number of official versions (more than 80 for Firefox and VLC, 27 for Blender and 13 for MySQL) lead to an imbalance while performing statistical analyses across all applications and could affect our study. In particular, if we allowed the values for Firefox and VLC to dominate the sample size, then the net results would be biased towards the mean of the values in the Firefox and VLC sample sets. To preserve generality and statistical significance, we equalize the sample set sizes as follows: for each official release date, we construct an observation for each application; the value for each observation is either actual, or linearly interpolated from the closest official releases, based on the time distance between the actual release and the day of the observation. This procedure ensures that we have an equal number of observations for all applications and eliminates bias due to varying release frequencies. To ensure that the interpolated values do not introduce noise in our sample, we tested whether the

[3] Effective lines of code (eLOC) are those lines that are not comments, blanks or standalone braces or parentheses.

original sample sets are normally distributed by using the *Kolmogorov–Smirnov* normality test. [4] We found that most of our original data sets are normally distributed and hence we can safely add the interpolated values to our sample.

Hypothesis testing. We perform statistical hypothesis testing to validate our analyses and the conclusions we draw. We use the t-test method to analyze our samples. For instance, if we have two sample sets, A and B, the t-test predicts the probability that a randomly chosen value from set A will be greater, lesser or equal to a randomly chosen value in set B. Although our sample sizes are equal, their variances differ, and therefore we use a special case of t-test called *Welch's* t-test [166]. For the rest of the chapter, by t-test we mean *Welch's* t-test. The t-test returns a t-value for a fixed level of statistical significance and the mean values of each of the sample sets. In our study we only consider 1% statistically significant t-values, to minimize chances of *Type I* error.[5] According to standard t-test tables, the results are statistically significant at the 1% level if t-value ≥ 2.08. In our case, we compute the values for a particular metric for both C and C++ code bases and perform a t-test on the individual sample sets. For example, if for a certain metric, the t-test returns a t-value ≥ 2.08 and the mean of the C sample set is *greater* than the mean of the C++ sample set, we claim a *statistical significance of 1%*; that is, if a value is chosen *randomly* from the C sample set, there is a 99% probability that the value of the random variable chosen will be closer to the mean of the C sample set than to the mean value of the C++ sample set. For each hypothesis testing, we report the mean of each sample set from C and C++ code bases, the t-values and the degrees of freedom, *df*.[6] We perform regression

[4] The *Kolmogorov–Smirnov* test is used for testing the normality of a distribution.
[5] A *Type I* error occurs when an acceptable hypothesis is rejected.
[6] Degrees of freedom is the number of independent observations in a sample of data that are available to

analysis for testing hypothesis *H1*, where we report the p-value, which is analogous to the t-value for the t-tests. For 1% statistically significant results, we must have p-value ≤ 0.01.

4.4 Study

In this section we discuss each hypothesis, the metrics we use to test it, as well as our findings. For conciseness, we only present selected graphs for each hypothesis; however, interested readers can refer to our technical report [18] for the complete set of graphs.

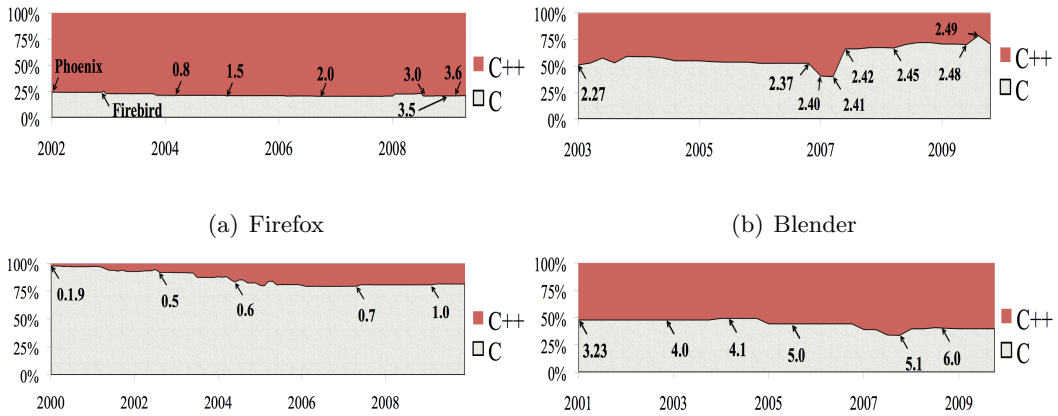
4.4.1 Code Distribution

Hypothesis (H_A^1): C++ is replacing C as a main development language. **Metrics**. To test this hypothesis we study how the percentages of C and C++ code change as an application evolves. We measure the eLOC of C and C++ code for each version using the Resource Standard Metrics (RSM) tool [141]. If the hypothesis holds, we should find that C++ percentages increase over an application's lifetime. **Results**. In Figure 4.3 and Table 4.1 we show the changes in C and C++ code percentages. To verify whether,

estimate a parameter of the population from which that sample is drawn.

Application	**First release**		**Last release**	
	C (%)	C++ (%)	C (%)	C++ (%)
Firefox	25.08	74.91	20.13	79.86
Blender	47.84	52.15	77.52	22.47
VLC	98.65	1.35	79.86	20.14
MySQL	49.82	50.17	40.35	59.64

Table 4.1: Percentage of C and C++ code.



(a) Firefox

(b) Blender

(c) VLC

(d) MySQL

Figure 4.3: eLOC distribution per language.

over time, the code base is shifting from C to C++, we perform a statistical hypothesis testing. Our null hypothesis, H_0^1, is that, over time, the code base division between C and C++ either remains constant, or the percentage of C code increases. We perform a two-step statistical analysis to verify this: (1) we measure the difference δ in the percentages of both C and C++ code (δ = %C++ − %C), and (2) we perform a linear regression analysis, where the independent variable is time (number of days since first release) and the dependent variable is δ. If H_0^1 is true, we should find that $\beta \leq 0$; if H_0^1 is rejected, we should have $\beta > 0$. We first perform this hypothesis testing across all applications (as described in Section 4.3.2) and then for each individual application. We present the results of the hypothesis testing in Table 4.2 when measured across all applications. Since we have $\beta > 0$ and p-value ≤ 0.01, we reject the null hypothesis H_0^1. Therefore, when performing the analysis across all applications, we observe that the primary code base is shifting from C to C++, i.e., H_A^1 is confirmed. In Table 4.3, we present the results for applications when

tested in isolation. We observe that we can reject H_0^1 for all applications except Blender. The p-values presented for H_0^1 do not imply that, for a given version of an application, the percentage of C++ code in that version is higher than the percentage of C code; rather, they imply that if a version of an application is chosen at random, there is a 99% probability that the percentage of C++ code in that version will be higher than in previously released versions of the same application.

Conclusion. Using linear regression, we confirmed that the percentage of C code is decreasing over time. However, when considering Blender in isolation, we notice a decrease in the percentage of C++ code, which is also evident from Figure 4.3(b) and Table 4.1. The reason behind the increase in the percentage of C code in Blender, as explained by one of the main developers [25], is that the developers "try to keep sections in the same code they were originally developed in."

4.4.2 Internal Quality

Hypothesis (H_A^2): C++ code is of higher internal quality than C code.

Metrics. Internal quality is a measure of how easy it is to understand and maintain an application. For each file in each application version, we use RSM to compute two standard metrics: cyclomatic complexity [7] and interface complexity. [8] As pointed out by Mockus et al. [112], normalizing the absolute value of a metric by dividing it by total eLOC is problematic. Since only a fraction of the code changes as the application evolves,

[7] Cyclomatic complexity is the number of logical pathways through a function [104].

[8] Interface complexity is the sum of number of input parameters to a function and the number of return states from that function [142].

Criterion	Conclusion
H_0^1	H_0^1 is **rejected** at 1% significance level ($\beta = 0.0036$, p-value $= 0.0002$, df $= 702$)
H_A^1	H_A^1 is **accepted** (% of C code is decreasing over time, while % of C++ code is increasing)

Table 4.2: Hypothesis testing for shift in code distribution (*H1*).

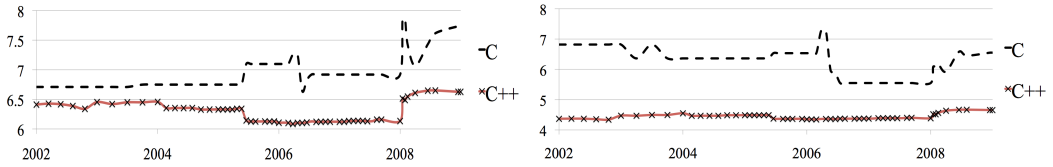
Application	β	***p*-value (1% significance)**	**df**	**Conclusion for H_0^1**
Firefox	0.0019	0.00049	97	Rejected
Blender	−0.0196	0.00001	27	Not rejected
VLC	0.0118	0.0007	72	Rejected
MySQL	0.0041	0.00262	11	Rejected

Table 4.3: Application-specific hypothesis testing for shift in code distribution (*H1*).

normalized values become artificially lower as the size of the source code increases. In our case, we found that the distributions of complexity values (across all files, for a particular application version) are skewed, thus arithmetic mean is not the right indicator of an ongoing trend. Therefore, to measure complexity for a specific version, we use the *geometric mean* computed across the complexity values for each file in that version. These geometric mean values constitute the sample sets for our hypothesis testing.

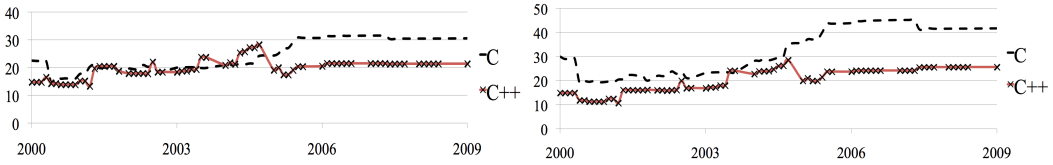
Results. Our null hypothesis is that C code has lower or equal code complexity compared to C++. To test this, we formulate two null sub-hypotheses corresponding to each complexity metric:

H_0^{c1}**:** The cyclomatic complexity of C code is less than or equal to the cyclomatic complexity of C++ code.



(a) Interface complexity (geometric mean) (b) Cyclomatic complexity (geometric mean)

Figure 4.4: Internal Quality in Firefox.



(a) Interface complexity (geometric mean) (b) Cyclomatic complexity (geometric mean)

Figure 4.5: Internal Quality in VLC.

H_0^{c2}**:** The interface complexity of C code is less than or equal to the interface complexity of C++ code.

If we reject hypotheses H_0^{c1} and H_0^{c2}, we conclude that the cyclomatic and interface complexities of C code are greater than those of C++ code. We perform two t-tests on each hypothesis: across all applications, and on individual applications. The results of both t-tests are presented in Tables 4.4, 4.5, and 4.6. Since the t-values are greater than 2.08, both when measured across all applications and when considering the projects in isolation, we could reject both null sub-hypotheses. Moreover, as can be seen in Tables 4.4, 4.5, and 4.6, the mean values for the C sets are significantly higher than the mean values for the C++ sets.

We now discuss application-specific changes we noticed during this analysis. For

VLC (Figure 4.5)[9] we find that, although initially both C and C++ have similar complexity values, starting in 2005, the interface complexity of C++ code decreases, while the interface complexity for C increases. However, for Firefox (Figure 4.4), Blender, and MySQL, the complexity of C code is always greater than that of C++ code. **Conclusion**. For all the applications we consider, we could confirm that C++ code has *higher* internal quality than C code.

4.4.3 External Quality

Hypothesis (H_A^3): C++ code is less prone to bugs than C code.

Metrics. External quality refers to users' perception and acceptance of the software. Since perception and acceptance are difficult to quantify, we rely on *defect density* as a proxy for external quality. Similar to Mockus et al. [112], we use two metrics for defect density: number of defects divided by the total eLOC and number of defects divided by the change in eLOC (ΔeLOC). As discussed by Mockus et al., number of defects per total

[9]To increase legibility, we deleted the markers for some minor or maintenance releases from Figures 4.4–4.9. The actual values are reflected in graph curves, hence this *does not affect* our results and analyses.

Criterion	**Conclusion**
H_0^{c1}	H_0^{c1} is **rejected** at 1% significance level ($\|t\| = 5.055$ when df = 354) Mean values: C = 16.57 , C++ = 11.02
H_0^{c2}	H_0^{c2} is **rejected** at 1% significance level ($\|t\| = 3.836$ when df = 387) Mean values: C = 16.52 , C++ = 12.63
H_A^2	H_A^2 is **accepted** (C++ code is of higher internal quality than C.)

Table 4.4: t-test results for code complexities (*H2*) across all applications.

Application	Mean values		$\|t\|$ (1% significance)	df
	C	C++		
Firefox	6.44	4.46	3.19	146
VLC	31.64	20.41	8.73	144
Blender	7.19	4.11	5.29	54
MySQL	28.17	21.52	2.11	12

Table 4.5: Application-specific t-test results for cyclomatic complexity.

Application	Mean values		$\|t\|$ (1% significance)	df
	C	C++		
Firefox	6.78	6.18	9.45	188
VLC	28.20	22.92	3.61	144
Blender	13.80	5.86	16.63	54
MySQL	27.71	17.13	3.41	22

Table 4.6: Application-specific t-test results for interface complexity.

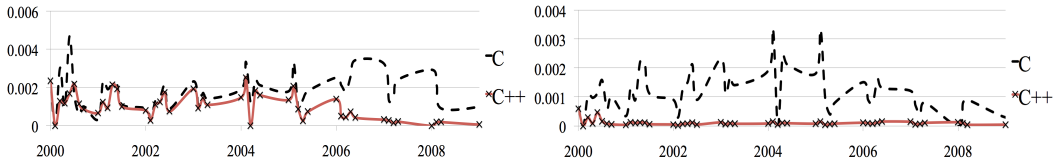
eLOC is potentially problematic as only a fraction of the original code changes in the new version of an application. Measuring defects over ΔeLOC is thus a good indicator of how many bugs were introduced in the newly added code.

Results. Our null hypothesis is: "C code has lower or equal defect density than C++ code." Based on the metrics we use to measure defect density, we divide the main hypothesis into two null sub-hypotheses:

H_0^{d1}**:** The defect density (measured over ΔeLOC) for C code is less than or equal to defect density in C++ code.

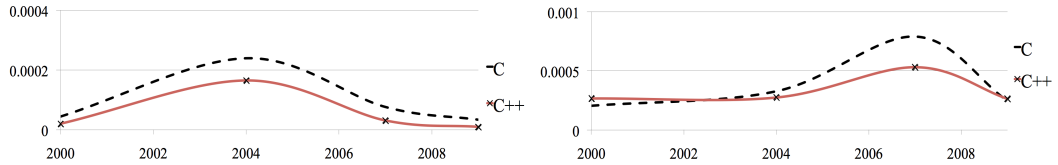
H_0^{d2}**:** The defect density (measured over total eLOC) for C code is less than or equal to defect density in C++ code.

Similar to t-tests for code complexity, we perform two sets of t-tests: one across all ap-



(a) Defect Density over ΔeLOC (b) Defect Density over total eLOC

Figure 4.6: Defect Density in VLC.



(a) Defect Density over ΔeLOC (b) Defect Density over total eLOC

Figure 4.7: Defect Density in MySQL.

plications, and another, for each application individually, using the original values. We present the results of the two tests in Tables 4.7, 4.8, and 4.9. From the t-values and differences in the mean defect densities of C and C++, we could reject both null sub-hypotheses when measured across all applications. When we apply the t-test using absolute values of defect densities for individual applications, we reject the null hypothesis at a statistically significant level for all programs except MySQL. This is caused by the unavailability of bug information for minor MySQL releases (which results in a small sample size for MySQL only) and does not affect the statistical significance of our conclusions, i.e., accepting H_A^3. As can be seen in Tables 4.7, 4.8, and 4.9, the mean defect density values for the C sets can be up to *an order of magnitude* higher than the mean values for the C++ sets.

In Figure 4.6 we present the evolution of defect densities in VLC (Firefox is sim-

ilar [18]), and note that these values tend to oscillate. The oscillations are due to bugs in major releases; these bugs tend to be subsequently fixed in maintenance releases. In MySQL we found that, for the first version only, the defect density of C++ code is slightly higher than the defect density of C code (when measured over total eLOC, see Figure 4.7(b)); this is not the case for subsequent versions. In Blender, we found that C code had higher defect density than C++, for both metrics; we omit the graphs for brevity.

Conclusion. Based on the t-test results, we could confirm H_A^3, that is, C++ code is less prone to bugs than C code.

4.4.4 Maintenance Effort

Hypothesis (H_A^4): C++ code requires less effort to maintain than C code.

Metrics. Prior work [48, 84, 82] has indicated that measuring software maintenance effort, or building effort estimation models for open source software is non-trivial, due to several reasons, e.g., the absence of organizational structure, developers working at their leisure. A widely used metric for effort is the number of commits divided by total eLOC [63, 82]. To avoid considering those parts of code which remain unchanged in a new release (similar to the argument presented for measuring defect density in Section 4.4.3), we also measure number of commits divided by ΔeLOC.

Results. Our null hypothesis is: "C files require less or equal effort to maintain than C++ files." We divide this into two null sub-hypotheses using the effort metrics we discussed:

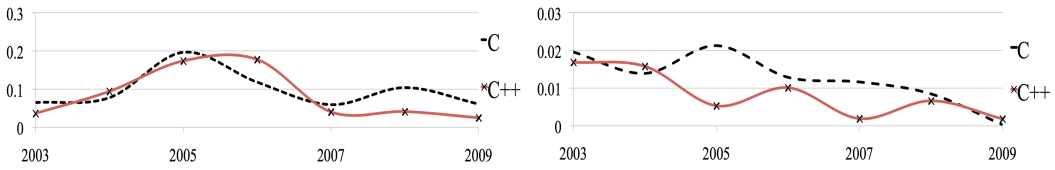
H_0^{e1}: The maintenance effort (measured over ΔeLOC) for C files is less than, or equal to,

Criterion	Conclusion
H_0^{d1}	H_0^{d1} is **rejected** at 1% significance level ($\|t\| = 4.77$ when df = 482) Mean values: C = 0.109 , C++ = 0.015
H_0^{d2}	H_0^{d2} is **rejected** at 1% significance level ($\|t\| = 4.82$ when df = 489) Mean values: C = 0.04 , C++ = 0.006
H_A^3	H_A^3 is **accepted** (C++ code is less prone to bugs than C code.)

Table 4.7: t-test results for defect density ($H3$) across all applications.

Application	Mean values		$\|t\|$ (1% significance)	df
	C	C++		
Firefox	0.02607	0.01939	1.0417	139
VLC	0.00178	0.00093	5.0455	87
Blender	0.03246	0.01981	1.7077	54
MySQL	0.00012	0.00007	0.6080	4

Table 4.8: Application-specific t-test results for defect density over ΔeLOC.



(a) Effort over ΔeLOC

(b) Effort over total eLOC

Figure 4.8: Maintenance Effort for Blender.

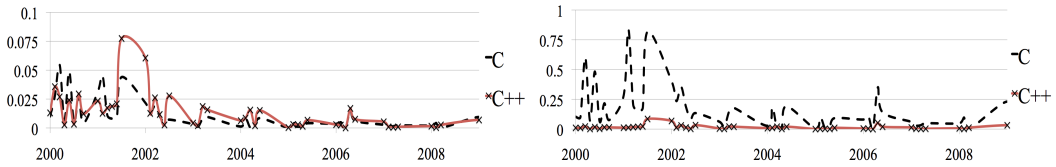
the maintenance effort for C++ files.

H_0^{e2}**:** The maintenance effort (measured over total eLOC) for C files is less than, or equal to, the maintenance effort for C++ files.

When we perform the t-test across all applications, we could *not* reject our null hypothesis H_0^{e1} at 1% significance level, as shown in Table 4.10. This is due to the very small differ-

Application	Mean values		$\|t\|$ (1% significance)	df
	C	C++		
Firefox	0.43246	0.16294	2.6412	106
VLC	0.00119	0.0001	9.8210	49
Blender	0.00551	0.00128	4.5520	30
MySQL	0.00046	0.00036	0.5190	3

Table 4.9: Application-specific t-test results for defect density over total eLOC.



(a) Effort over ΔeLOC

(b) Effort over total eLOC

Figure 4.9: Maintenance Effort for VLC.

ence between the mean values of effort for C and C++ files when measured over ΔeLOC. However, we could reject our null hypothesis H_0^{e2} and confirm that the effort to maintain C files (when measured over total eLOC) is higher than the effort for C++ files. Note that we could reject H_0^{e1} at a weaker level of significance (10%), but, to retain uniformity and reduce the probability of introducing errors in our conclusions, we employ 1% level of significance across all hypothesis testing.

In Tables 4.11 and 4.12 we present our t-test results on H_0^{e1} and H_0^{e2} for individual applications. While we could only reject the null sub-hypotheses for VLC, note that the mean values for C are *higher* than the mean values for C++ for all applications.

From Figures 4.8 and 4.9 we notice how the file maintenance effort changes over time for VLC and Blender. As evident from the mean values from Tables 4.11 and 4.12,

even though for one version the absolute value for effort for C++ files might be higher than C, across the whole evolution period, the maintenance effort values for C files is higher than the effort required to maintain C++ files.

Conclusion. We could confirm our hypothesis only when measuring maintenance effort over total eLOC. When measuring maintenance effort over ΔeLOC, even though the mean values for C++ files are less than the mean values for C files, we could not validate our hypothesis at a statistically significant level.

4.5 Threats to Validity

We now present possible threats to the validity of this chapter's work.

Selection Bias. An important trait of our study is aiming to reduce selection bias, i.e., making sure that high-level languages do not appear to be "better" because they are favored by more competent developers, or are used for easier tasks. Therefore, following our quantitative analysis, we also asked developers several questions to determine whether there is bias in language selection. For example, a key VLC developer stated [42] that

Criterion	**Conclusion**
H_0^{e1}	H_0^{e1} is **not rejected** at 1% significance level ($\|t\| = 1.218$ when df = 147) Mean values: C = 1.07 , C++ = 0.999 H_0^{e1} is **rejected** at 10% significance level
H_0^{e2}	H_0^{e2} is **rejected** at 1% significance level ($\|t\| = 2.455$ when df = 102) Mean values: C = 0.594 , C++ = 0.26

Table 4.10: t-test results for maintenance effort ($H4$) across all applications.

Application	Mean values		$\|t\|$ (1% significance)	df
	C	C++		
Firefox	4.1780	2.9626	1.0824	15
VLC	0.1719	0.0154	5.4499	49
Blender	0.1026	0.0919	1.7077	54
MySQL	0.0004	0.0002	0.6080	4

Table 4.11: Application-specific t-test results for maintenance effort over ΔeLOC.

Application	Mean values		$\|t\|$ (1% significance)	df
	C	C++		
Firefox	2.4110	0.9217	4.9284	12
VLC	0.0119	0.0104	0.5180	95
Blender	0.0114	0.0069	1.2568	9
MySQL	0.0017	0.0012	1.0737	3

Table 4.12: Application-specific t-test results for effort maintenance over total eLOC.

"developers are expected to know C and C++" when they join the project and perceived difficulty of implementing a task "does not really [play a role in selecting the language]." Moreover, for the VLC project, LUA, a high-level language, is preferable to C: "LUA is used most for text processing, where performance is not critical, and C would be too prone to programming errors [...] Where performance is not an issue [...] C code has and will continue to be replaced with LUA code." Perceived task difficulty does not play a role in language selection in Blender either, as indicated in Section 4.4.1.

Empirical Studies. The empirical nature of our study exposes it to construct, content, internal and external threats to validity.

Construct validity relies on the assumption that our metrics actually capture the intended characteristic, e.g., defect density accurately models external quality, source code metrics accurately model internal quality. We intentionally used multiple metrics for each

hypothesis to reduce this threat. We randomly chose several versions from each application and verified that, for those developers who commit to both code bases, the number of C commits and C++ commits are comparable. This balance indicates developers have no selection bias towards which code base they want to commit to—an assumption confirmed by developers.

To ensure *content validity* we selected applications that contain both C and C++ code, written by developers who contribute to both code bases, and we analyzed as long a time span in a program's lifetime as possible. For Firefox, we do not count bugs labeled as "invalid bugs," though we found 7 instances (out of 5786 Firefox bugs) where these bugs were re-opened in subsequent versions. There is a possibility that invalid bugs might be re-opened in the future, which will very slightly change our results.

Internal validity relies on our ability to attribute any change in system characteristics (e.g., metric values or eLOC) to changes in the source code, rather than accidentally including or excluding files, inadvertently omitting bugs or commits. We tried to mitigate this threat by (1) manually inspecting the releases showing large gains (or drops) in the value of a metric, to make sure the change is legitimate, and (2) cross-checking the change logs with information from bug databases as described in Section 4.3.1. When we group committers by the code base they are contributing to, we use committer IDs to assign developers to the C code base, to the C++ code base, or to both code bases. Since we cannot differentiate among committers who have multiple IDs, we run the risk of over-reporting or under-reporting the number of committers.

External validity, i.e., the results generalize to other systems, is also threatened

in our study. We have looked at four open-source projects written in a combination of C and C++ to keep factors such as developer competence or software process uniform. Therefore we cannot claim that arbitrary programs written in C++ are of higher quality than arbitrary programs written in C; nevertheless, we show that all other factors being equal, the choice of programming language does affect quality. It is also difficult to conclude that our proposed hypotheses hold for proprietary software, or for software written in other combinations of lower- and higher-level languages, e.g., C and Java or C and Ruby.

4.6 Contribution Summary

In summary, our main contributions are:

- A novel way to analyze factors that impact software quality while controlling for both developer expertise and the software development process.
- A multi-faceted software evolution study of four large applications, measuring software quality, development effort, and code base shifts between languages.
- Formulation of four hypotheses and statistical analyses designed to capture whether a particular language leads to better software.

4.7 Conclusions

In this chapter we introduce a novel methodology for quantifying the impact of programming language on software quality and developer productivity. To keep factors such as developer competence or software process uniform, we investigate open source applications

written in a combination of C and C++. We formulate four hypotheses that investigate whether using C++ leads to better software than using C. We test these hypotheses on large data sets to ensure statistically significant results. Our analyses demonstrate that applications that start with C as the primary language are shifting their code base to C++, and that C++ code is less complex, less prone to errors and requires less effort to maintain.

Chapter 5

A Graph-based Characterization of Software Changes

In this chapter, we argue that emerging techniques in Network Sciences and graph-mining approaches can help to better understand software evolution, and to construct predictors that facilitate development and maintenance. Specifically, we show how we can use a graph-based characterization of a software system to capture its evolution and facilitate development, by helping us estimate bug severity, prioritize refactoring efforts, and predict defect-prone releases. Our work consists of three main thrusts. First, we construct graphs that capture software structure at two different levels: (a) the product, i.e., source code and module level, and (b) the process, i.e., developer collaboration level. We identify a set of graph metrics that capture interesting properties of these graphs. Second, we study the evolution of eleven open source programs, including Firefox, Eclipse, MySQL, over the lifespan of the programs, typically a decade or more. Our metrics detect some surprising similari-

ties but also significant differences in the evolution of these programs. Third, we show how our graph metrics can be used to construct predictors for bug severity, high-maintenance software parts, and failure-prone releases.

5.1 Introduction

Graph-based analysis and mining of complex systems have experienced a resurgence, under the name of Network Science (or graph mining). There is a good reason for this: Network Science has revolutionized the modeling and analysis of complex systems in many disciplines and practical problems. For example, graph-based methods have opened new capabilities in classifying network traffic [73, 74], modeling the topology of networks and the Web [46, 6], and understanding biological systems [172, 6]. What these approaches have in common is the creation of graph-based models to represent communication patterns, topology or relationships. Given a graph model, one can unleash a large toolset of techniques to discover patterns and communities, detect abnormalities and outliers, or predict trends.

The overarching goal of this chapter is to find whether graph-based methods facilitate software engineering tasks. Specifically, we use two fundamental questions to drive our work: (a) how can we improve maintenance by identifying which components to debug, test, or refactor first?, and (b) can we predict the defect count of an upcoming software

The work presented in this chapter have been published in the proceedings of the 2012 IEEE International Conference on Software Engineering [14]. I would like to thank Dr. Marios Illiofotou, a co-author of this work for letting me use his Perl scripts which were useful in computing several graph metrics.

release?

Note that our intention is not to find the best possible method for each question, but to examine if a graph-based method can help, through the use of an appropriately-constructed graph model of the software system. While we use these two indicative questions here, we believe there could be other questions that can be addressed with graph-based approaches. Our thesis is that graph-based approaches can help to better understand software evolution, and to construct predictors that facilitate development and maintenance. To substantiate, we show how we can create graph-based models that capture important properties of an evolving software system. We analyze eleven open-source software programs, including Firefox, Eclipse, MySQL, Samba, over their documented lifespans, typically a decade or more. Our results show that our graph metrics can detect significant structural changes, and can help us estimate bug severity, prioritize debugging efforts, and predict defect-prone releases.

Our contributions can be grouped in three thrusts.

a. Graphs hide a wealth of information regarding software engineering aspects. We propose the use of graphs to model software at two different levels and for each level, we propose two different granularities.

At the software *product* level, we model the software structure, at the granularity of functions (function-level interaction) and modules (module-level interaction).

At the software *process* level, we model the interactions between developers when fixing bugs and adding new features. We use two construction methods: the *bug-based developer collaboration*, which captures how a bug-fix is passed among developers, and

commit-based developer collaboration which represents how many developers collaborated in events other than bug fixes, by analyzing the commit logs.

b. Graph metrics capture significant events in the software lifecycle. We study the evolution of the graph models of these programs over one to two decades. We find that these graphs exhibit some significant structural differences, and some fundamental similarities. Specifically, some graph metrics vary significantly among different programs, while other metrics captures persistent behaviors and reveal major evolutionary events across all the examined programs. For example, our graph metrics have revealed major changes in software architecture in mid-stream releases (not ending in ".0"): OpenSSH 3.7, VLC-0.8.2 and Firefox 1.5 show big changes in graph metrics which, upon inspection, indicate architectural changes that trump changes observed in ".0" versions of those programs. Similarly, our *edit distance* metric has detected a major change in Samba's code structure in release 1.9.00 (Jan 22, 1995), due to major bug fixes and feature additions; the change is not apparent when examining other metrics such as eLOC.

c. Our graph metrics can be used to predict bug severity, maintenance effort and defect-prone releases. The cornerstone of our work is that our graph metrics and models can be used to suggest, infer, and predict important software engineering aspects. Apart from helping researchers construct predictors and evolution models, our findings can help practitioners in tasks such as: identifying the most important functions or modules, prioritizing bug fixes, estimating maintenance effort:

1. We show how *NodeRank*, a graph metric akin to PageRank, can predict bug severity.

2. We show how the *Modularity Ratio* metric can predict which modules will incur high maintenance effort.

3. We demonstrate that by analyzing the *edit distance* in the developer collaboration graphs we can predict failure-prone releases.

While these predictors might seem intuitive, we are the first to quantify the magnitude and lag of the predictors. More importantly, the goal of our work is to establish that Network Science analysis and metrics capture and reflect important software engineering properties and become a bridge between Network Science concepts with software engineering methods and practices.

5.2 Methodology and Applications

We used four kinds of graphs for this study as shown in Figure 5.1: (1) function call graphs ($\mathbb{G}_{Func}$), (2) module collaboration graphs ($\mathbb{G}_{Mod}$), (3) source-code based developer collaboration graphs ($\mathbb{G}_{SrcCodeColl}$), and (4) bug tossing graphs ($\mathbb{G}_{BugToss}$). The methodology and formal definition used in extracting these graphs have been explained in Chapter 2.2.1. We based our study on eleven popular open source applications. [1]

5.3 Metrics

We introduce the graph metrics, and the software engineering concepts, which we will use in our work.

[1]The details about these applications can be found in Chapter 2.

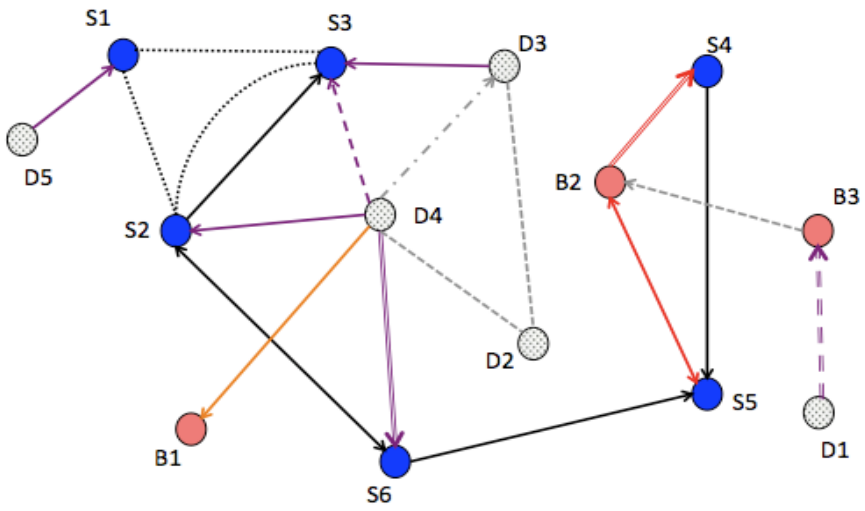


Figure 5.1: Hypergraph extraction for a graph-based approach to characterize software changes (G_{Func}, G_{Mod}, $G_{SrcCodeColl}$, $G_{BugToss}$).

5.3.1 Graph Metrics

For each metric, we indicate if it is calculated on a directed and undirected graph. Our graphs are initially directed, but can be trivially transformed into undirected graphs, by ignoring the directivity of the edges.

Average degree (directed graph). In a graph $G(V, E)$, V denotes the set of nodes and E denotes the set of edges. The average degree is defined as:

$$\bar{k} = \frac{2|E|}{|V|}$$

Clustering coefficient (undirected graph).

The clustering coefficient $C(u)$ of a node u captures the local connectivity, or the probability that two neighbors of u are also connected. Formally, it is defined as the ratio of the number of existing edges between all neighbors of u over the maximum possible number

of such edges. Let $|\{e_{jk}\}|$ be the number of edges between u's neighbors and k_u be the number of u's neighbors. Then, we have:

$$C(u) = \frac{2|\{e_{jk}\}|}{k_u(k_u - 1)}$$

The metric is meaningful for nodes with at least two neighbors. The *average clustering coefficient* of a graph is the average clustering coefficient over all the nodes.

NodeRank (directed graph). We define a measure called *NodeRank* that assigns a numerical weight to each node in a graph, to measure the relative importance of that node in the software—this rank is similar to PageRank [29], which represents the stationary distribution of the graph interpreted as a Markov chain. There are several ways for defining and calculating the PageRank. Here, we use the following recursive calculation. For a node u, let $NR(u)$ be its *NodeRank*, and let the set IN_u contains all the nodes v that have an outgoing edge to u. We assign equal NoderRank values to all nodes initially. In every iteration, the new $NR(u)$ is the sum over all $v \in IN_u$:

$$NR(u) = \sum_{v \in IN_u} \frac{NR(v)}{OutDegree(v)}$$

We stop the iteration when the NodeRank values converge, which is quite fast in all our graphs. Note that to enable convergence, at the end of every iteration, we normalize all values so that their sum is equal to one. Intuitively, the higher the *NodeRank* of a vertex u, the more important u is for the program, because many other modules or functions depend on it (i.e., call it). Similarly, in the developer collaboration graph, a developer D with a high $NR(D)$ signifies a reputable developer.

Graph diameter (undirected graph). is the longest shortest path between any two vertices in the graph.

Assortativity (undirected graph). The assortativity coefficient is a correlation coefficient between the degrees of nodes on two ends of an edge; it quantifies the preference of a network's nodes to connect with nodes that are similar or different, as follows. A positive assortativity coefficient indicates that nodes tend to link to other nodes with the same or similar degree. Assortativity has been extensively used in other Network Science studies. For instance, in social networks, highly connected nodes tend to be connected with other high degree nodes [106]. On the other hand, biological networks typically show disassortativity, as high degree nodes tend to attach to low degree nodes [123].

Edit distance (directed graph). The metrics we described so far characterize a single program release, e.g., the module collaboration graph for release 1. To find out how program structure changes over time, we introduce a metric that captures the number of changes in vertices and edges between two graphs, in our case between successive releases. The *edit distance* $ED(G_1, G_2)$ between two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ is defined as follows:

$$ED(G_1, G_2) = |V_1| + |V_2| - 2 * |V_1 \cap V_2| + |E_1| + |E_2| - 2 * |E_1 \cap E_2|$$

Intuitively, if G_1 and G_2 model software structures for releases 1 and 2, then high values of $ED(G_1, G_2)$ indicate large-scale structural changes between releases.

Modularity ratio (directed graph). Standard software engineering practice suggests that software design exhibiting high cohesion and low coupling provides a host

of benefits, as it makes software easy to understand, easy to test, and easy to evolve [55]. Therefore, we define the *modularity ratio* of a module A as the ratio between its cohesion and its coupling values:

$$ModularityRatio(A) = \frac{Cohesion(A)}{Coupling(A)}$$

where $Cohesion(A)$ is the total number of intra-module calls or variable references in A; $Coupling(A)$ is the total number of inter-module calls or variable references in A.

5.3.2 Defects and Effort

Defect density. We use defect density to assess external application quality. To ensure accuracy, we extract (and cross-check) information from bug databases and bug information extracted from change logs. With the bug information at hand, we then associate a certain bug to a certain version: we use release tags, dates the bug was reported, and commit messages to find the version in which the bug was reported in, and we attributed the bug to the previous release.

Effort. To measure development and maintenance effort, we counted the number of commits and the churned eLOC (sum of the added and changed lines of code) for each file for a release, in a manner similar to previous work by other researchers [121, 48]. This information is available from the log files.

Project	Metric values											
	Nodes		Edges		Average degree		Clustering coefficient		Diameter		Assortativity	
	first	last	first	last	first	last	first	last	first	last	first	last
Firefox	9,332	28,631	89,045	787,297	19.08	54.39	0.062	0.111	12	16	-0.022	-0.07
Blender	5,525	30,955	14,567	80,304	5.27	5.18	0.094	0.091	17	30	-0.126	-0.097
VLC	445	5,049	961	15,131	4.31	5.99	0.122	0.095	10	14	-0.194	-0.086
MySQL	4,980	6,631	19,291	29,707	7.47	5.34	0.114	0.082	18	17	-0.113	-0.142
Samba	110	11,674	408	51,136	7.41	8.76	0.146	0.128	7	12	-0.287	-0.181
Bind	5,133	6,718	10,337	15,573	4.02	7.80	0	0.110	21	17	-0.165	-0.12
Sendmail	1,072	599	3,089	1,435	5.76	4.79	0	0	9	11	-0.198	-0.201
OpenSSH	214	1,030	773	4,056	7.22	7.87	0.187	0.141	6	12	-0.181	-0.224
SQLite	290	2,046	496	4,241	3.42	4.16	0	0	12	15	-0.245	-0.126
Vsftpd	597	982	921	1,712	3.08	3.48	0	0	11	12	-0.202	-0.206

Table 5.1: Metric values (function call graphs) for *first* and *last* releases.

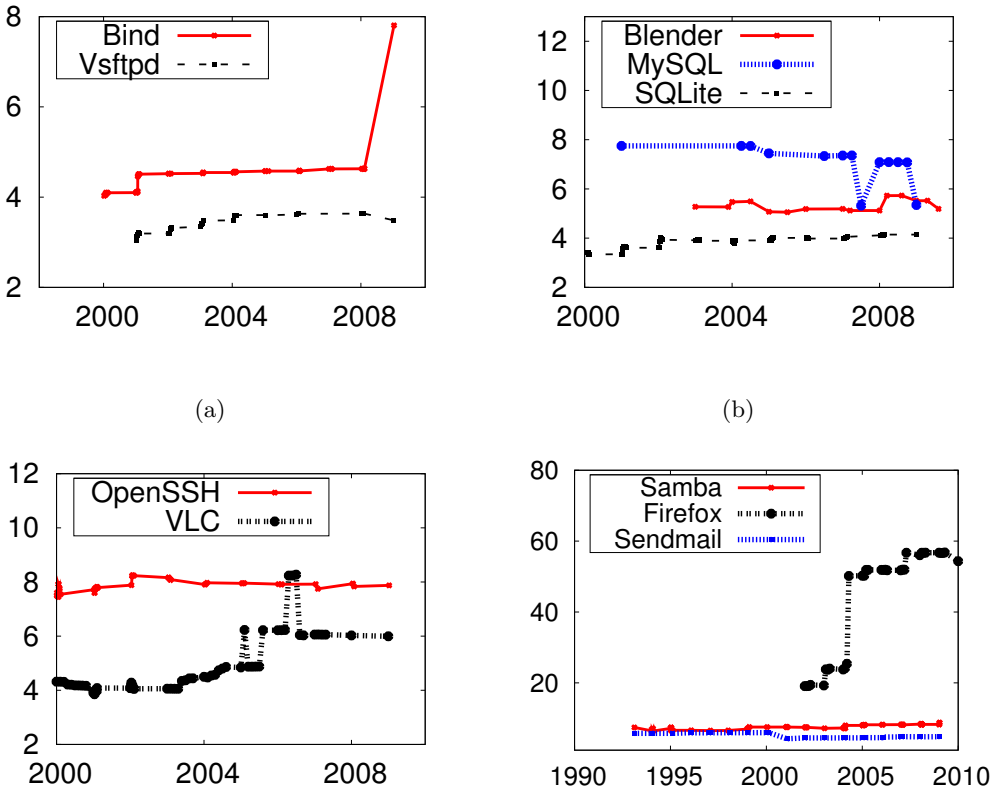


Figure 5.2: Change in *Average Degree* over time.

5.4 A Graph-based Characterization of Software Structure and Evolution

Most prior work on source code-based graph analysis has focused on characterizing single releases [94, 118, 159, 158, 150, 98, 97, 168, 157] or analyzing limited evolution time spans [160], or a longer evolution time span for a single program [163]. Therefore, one of the objectives of our study was to analyze complete lifespans of large projects and observe how the graphs evolve over time. This puts us in a position to answer questions such as:

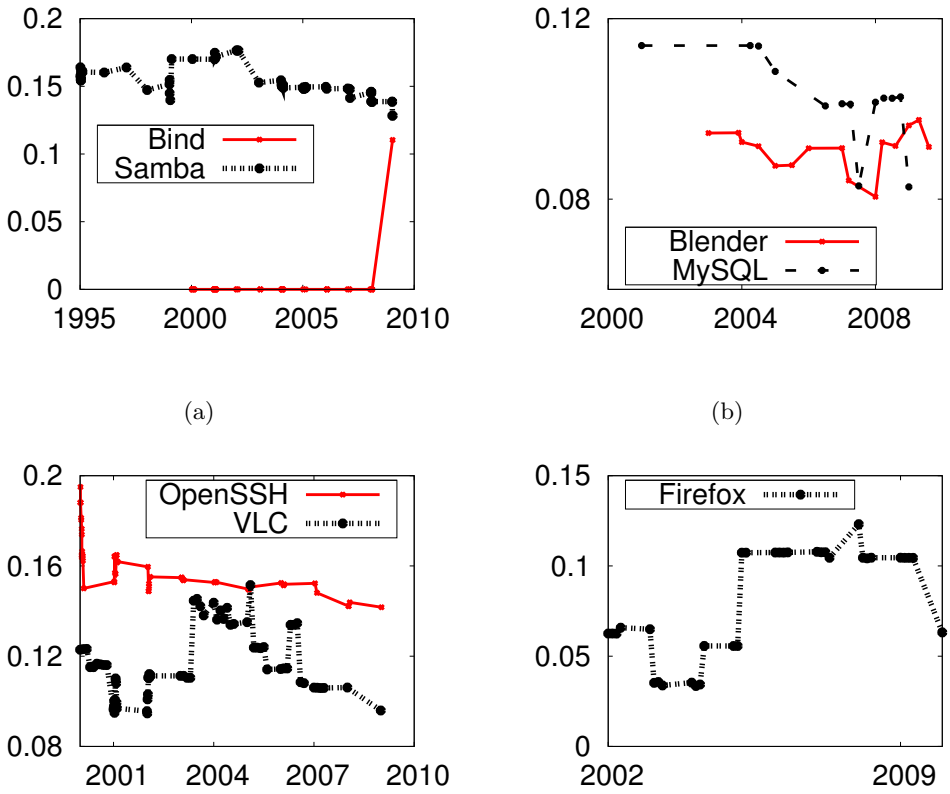


Figure 5.3: Change in *Clustering Coefficient* over time.

Can graph metrics detect non-obvious "pivotal" moments in a program's evolution?

Are there invariants and metric values that hold across all programs?

Are there evolution trends that are common across programs?

We now proceed to showing how these metrics evolve over time for our examined applications and discuss how these changes and trends in graph metrics could affect various software engineering aspects, both for the product and for the process. The numeric results,

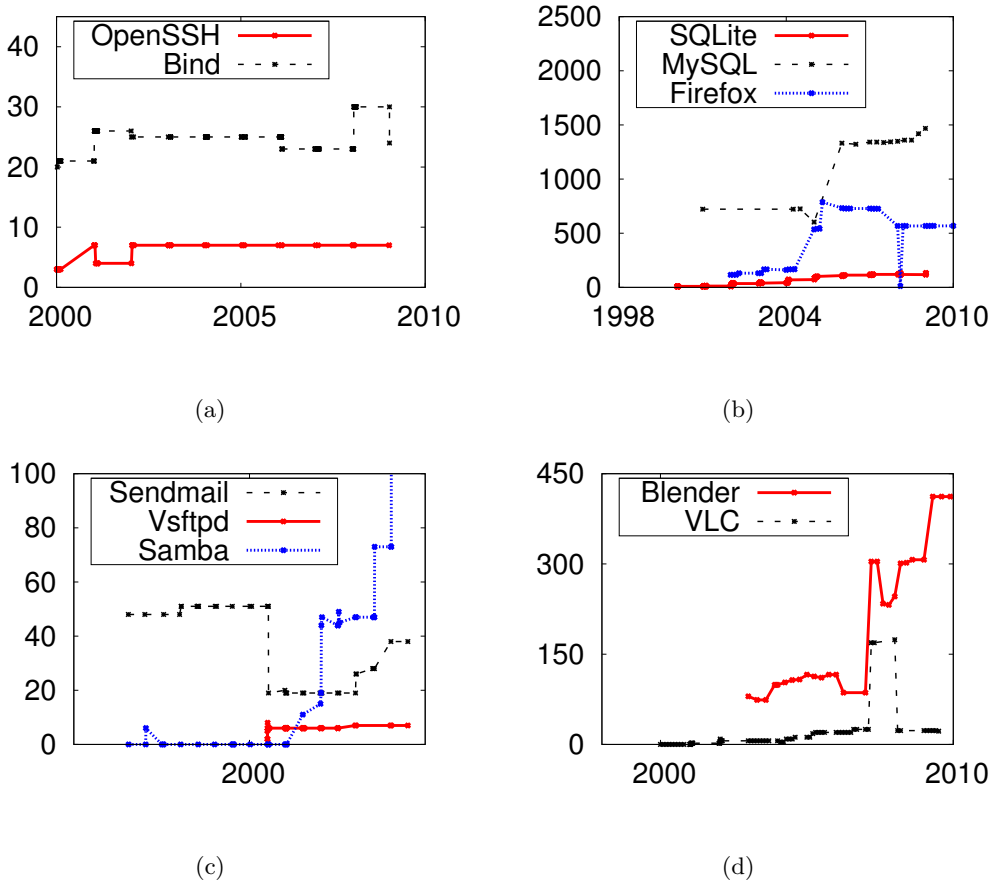


Figure 5.4: Change in *Number of Cycles* over time.

i.e., metric values for the first and last releases, are shown in Table 5.1. The evolution charts are in Figures 5.2 – 5.8. The data and figures refer to function call graphs.

Nodes and edges. The initial and final number of nodes and edges are presented in Table 5.1. Due to lack of space, we do not present evolution charts. However, we have observed that some programs exhibit linear growth (Bind, SQLite, OpenSSH, MySQL) and some super linear growth (Blender Samba, VLC) in terms of number of nodes over time. The same observation holds for the evolution of the number of edges. This is intuitive, since,

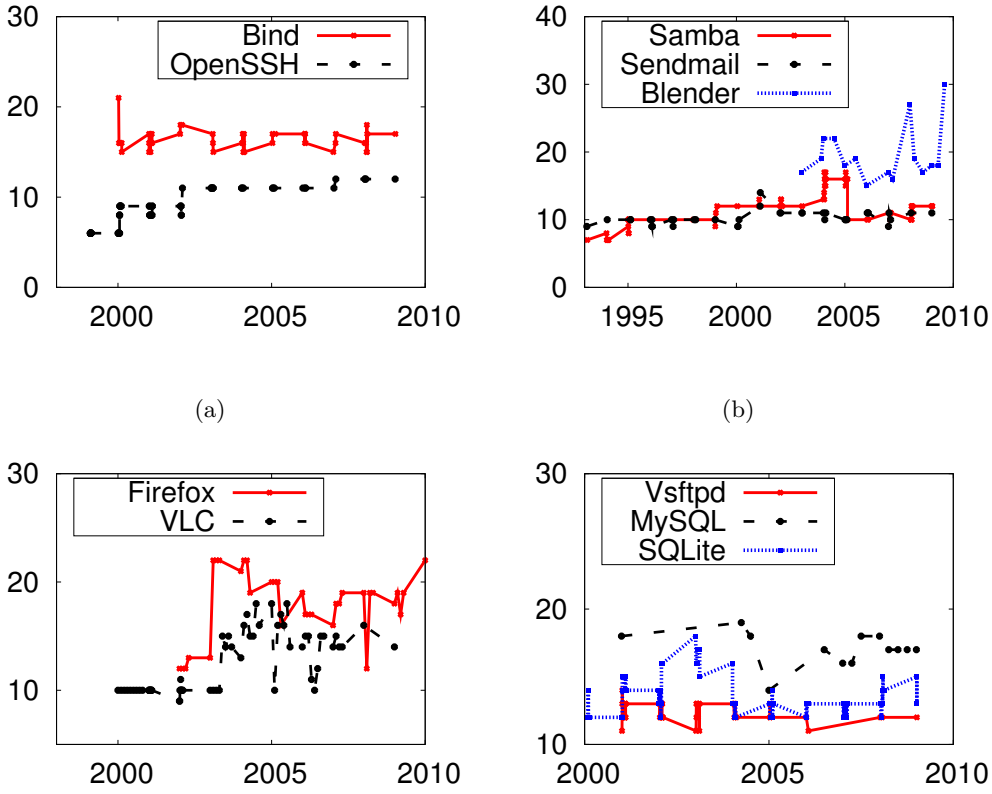


Figure 5.5: Change in *Graph Diameter* over time.

as shown in Table 2.4, program size (eLOC) grows over the studied spans for all programs.

The only outlier was Sendmail where we noticed that neither the number of nodes, nor the number of edges increase, although eLOC increases (cf. Table 2.4). We believe this to be due to the maturity of Sendmail—code is added to existing functions, rather than new functions being added, hence the increase in the size (in eLOC) of each function but no increase in the number of functions. The number of eLOC per node differ significantly across programs, from 10 to 323; the number of eLOC per edge ranged from 5 to 150. Values

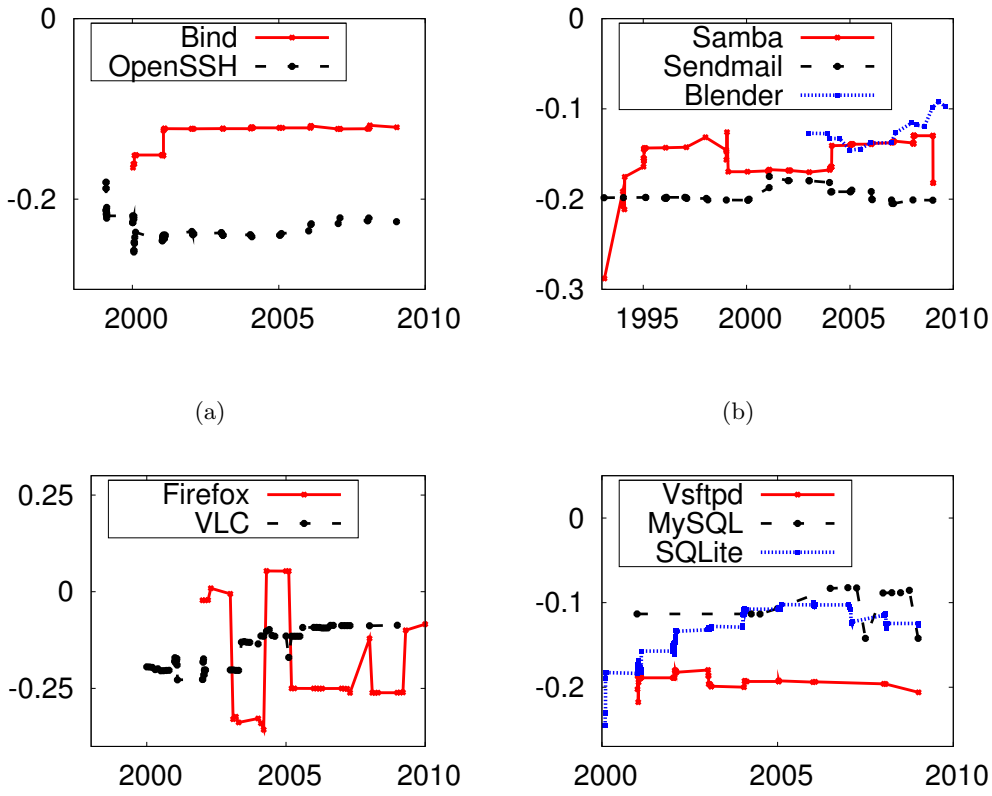


Figure 5.6: Change in *Assortativity* over time.

of both metrics decrease with evolution for Firefox, Blender, VLC, MySQL, OpenSSH and SQLite; and increase for Samba, Bind, Sendmail and Vsftpd.

Average degree. Intuitively, the degree of a function or module indicates its popularity. The average degree of a graph helps quantify coarseness: graphs with high average degrees tend to be tightly connected [99]. In Figures 5.2 we show the evolution of this metric for each program. We find that for all programs but MySQL, the average degree increases with time, albeit this increase tends to be slight, and the value range is 2–10. One

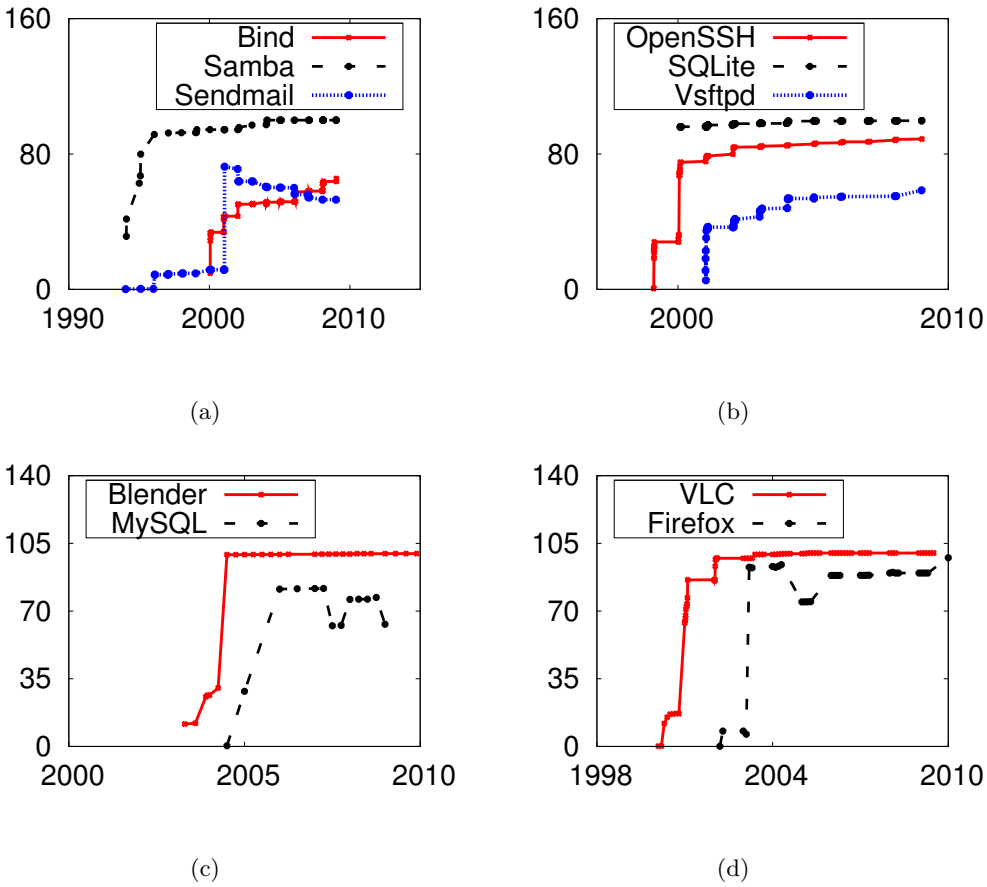


Figure 5.7: Change in *Edit Distance* over time.

interesting aspect to note is the average degree of Firefox, which is orders of magnitude higher, ranging from 20 to 60.

On further investigation we found that the graph topology for Firefox differs significantly from the remaining projects. The three notable observations are: (1) the majority of the nodes have low degree (average degree less than 20) and they are not connected with each other, (2) a large group of high-degree nodes (average degree 200–800) are interconnected with each other and form a dense core in the graph, and (3) most of the low degree

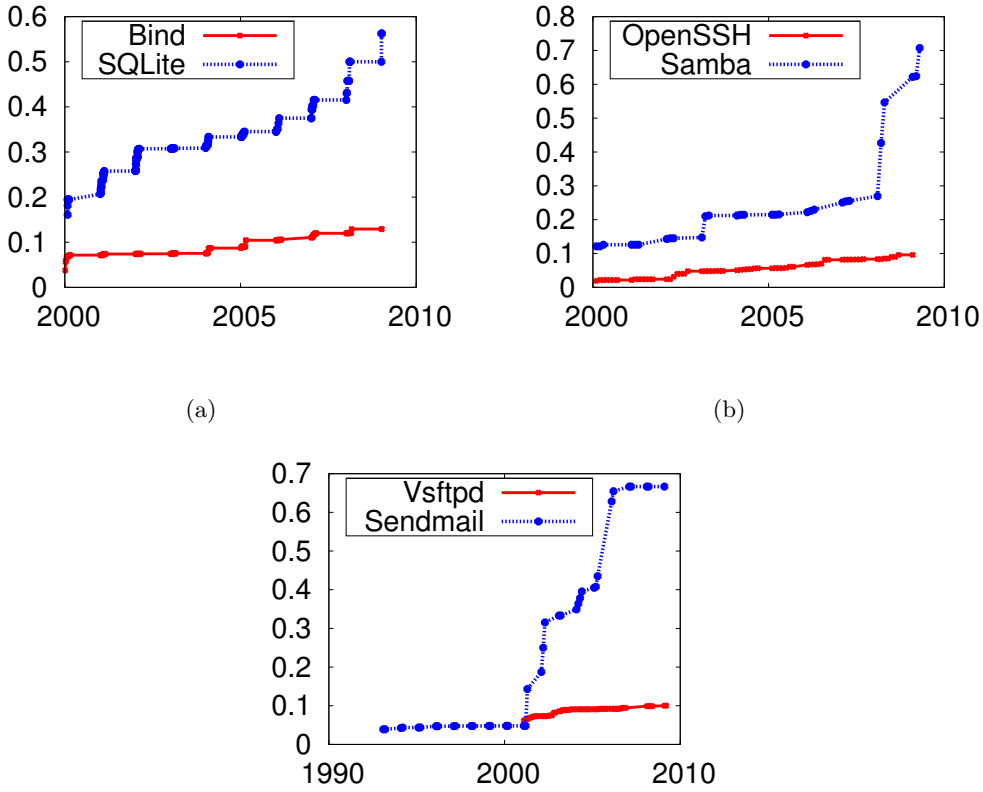


Figure 5.8: Change in *Modularity Ratio* over time.

nodes are connected with this dense core. We found that this group of nodes with high degree and high interconnectivity are part of the common library in Mozilla used by the majority of the products, including Firefox. On the contrary, in the other projects where the average degree is low, we found that: (1) the majority of nodes have degree close to the average degree of the graph, and (2) there are very few nodes of high degree and very few of them connect with each other.

Clustering coefficient. As defined in Section 5.3.1, the clustering coefficient is

a measure of how well-connected the local neighborhoods are in a graph. Zero values of this coefficient indicate a bipartite graph. High values of clustering coefficient indicate tight connectivity and violate good software engineering practice, because graph nodes do not form a hierarchy of levels of abstraction (due to the presence of horizontal and backward edges), which complicates program understanding, testing, and evolution [55, 128].

In our case (Table 5.1), we found that for Vsftpd, SQLite, Sendmail and all but the last release of Bind, the clustering coefficient values are zero throughout the project's lifetime, suggesting bipartite graphs; we verified that indeed these programs have bipartite call graphs. In the case of VLC, we found that in version 0.7.0 there was a sudden rise in the clustering coefficient value. On further investigation we found that the Flac demuxer code was rewritten for this version; although the function signatures remained the same from the previous version, there was a significant change in intra-module calls in the demuxer module leading to an increase in clustering coefficient. For the remaining programs, we find that clustering coefficients are remarkably similar: their range is 0.08–0.20 and values decrease over time, with the exception of Firefox.

Number of nodes in cycles. Cycles in software structure affect software quality negatively. For example, cycles in the module collaboration graph indicate circular module dependencies, hence modules that are hard to understand and hard to test: "nothing works until everything works, " as per standard software engineering literature [128, 55]. Cycles in the call graph indicate a chain of mutually recursive functions, which again are hard to understand, test, and require a carefully orchestrated end-recursion condition. An increase in the number of nodes in cycles from one release to another would signify decrease in soft-

ware quality, and indicate the need for refactoring. We observed that for Samba, MySQL, and Blender the number of nodes in cycles increases linearly with time, for OpenSSH, Bind, SQLite, and Vsftpd the number remains approximately constant with time, whereas for VLC and Sendmail there is no clear trend. For MySQL, a sudden increase in number of nodes in cycles is noticeable in version 5.0 (Oct. 2005); on further investigation we found that newly-added functions in the InnoDB storage engine code form strongly connected components in the graph, leading to an increase in the number of nodes in cycles. For reasons mentioned above, even a constant number of nodes in cycles (let alone an increasing one) is undesirable.

Graph diameter. From a maintenance standpoint, graphs of high diameter are undesirable. As the diameter measures distance between nodes, graphs with large diameter are more likely to result in deep runtime stacks, which hinder debugging and program understanding. As shown in Table 5.1 (columns 10 and 11) and in Figure 5.5, we notice that for our programs the diameter tends to stay constant or vary slightly, and the typical value range (10–20) is similar across all programs.

Assortativity. As explained in Section 5.3.1, high values of assortativity indicate that high-degree nodes tend to be connected with other high degree nodes; low assortativity values indicate that high-degree nodes tend to connect with low-degree nodes [123]. As shown in Table 5.1 (columns 12 and 13) and in Figure 5.6, we notice that all the values of assortativity for all the programs are negative, which implies that, similar to biological networks, software networks exhibit disassortative mixing, i.e., high degree nodes tend to connect to low degree nodes and vice versa.

In Firefox, low assortativity stems from code reuse. Mozilla has its own function libraries for, e.g., memory management, and these functions were used by many other nodes. As a result library functions and modules exhibit very high degrees, and connect to many low degree nodes, hence contributing to low assortativity. In the future, we intend to further investigate the relationship between assortativity and code reuse.

Edit distance. This metric, as defined in Section 5.3.1, captures the dynamic of graph structure changes, i.e., how much of the graph topology changes with each version. In Figure 5.7 we show how the graph edit distance changes over time. We find the same pattern for all programs: after a steep rise, the edit distance plateaus or increase slightly, i.e., is a step-function. This observation strengthens the conclusions of prior research [160], namely that software structure stabilizes over time, and the only tumultuous period is toward the beginning. We found that these steep edit distance rises are due to major changes and they indicate that software has reached structural maturity. For example, the pivotal moment for Samba is release 1.9.00 (Jan 22, 1995), where 131 modification requests were carried out, whereas for the versions prior to 1.9.00, the average number of modification requests per release was 15. We also computed the eLOC difference for release 1.9.00 and found it to be 2kLOC, *less* than the 3kLOC average of the previous releases, which shows how graph-based metrics can reveal changes that would go undetected when using LOC measures.

Modularity ratio. This metric reveals whether projects follow good software engineering practice, i.e., whether, over time, the cohesion/coupling ratio increases, indicating better modularity. This turned out to be the case for all programs, except Firefox version 1.5 (see bottom of Figure 5.8, and Figure 5.10; we show VLC, Blender, MySQL and Firefox

in a separate figure because we used modularity ratio for prediction).

Discussion. We are now in a position to answer the questions posed at the beginning of this section. We have observed that indeed, software structure is surprisingly similar across programs, in absolute numbers (see the similar ranges for average degree, clustering coefficient, graph diameter), which suggests intrinsic lower and upper bounds on variation in software structure. There are also similarities in trends and change patterns (cf. edit distance, clustering coefficient, modularity ratio) which suggest that programs follow common evolution laws. For those releases where graph metrics change significantly, we found evidence that supports the "pivotal moment" hypothesis. For example, in Firefox, we find significant changes in average degree, clustering coefficient, and edit distance for release 2.0 (Oct. 2006); indeed, release notes confirm many architectural and feature enhancements introduced in that version. Similarly, for OpenSSH we found that one such moment was release 2.0.0beta1 (May 2000), a major version bump from prior release (1.2.3), that incorporated 143 modification requests, whereas the average modification requests per release until that point was 27 and this change is reflected in significant change of values for clustering coefficient, edit distance, and assortativity metric. This evidence strengthens our argument that graph metrics are good measures that can reveal events in evolution.

So far our discussion has centered on changes in structural (graph) metrics and understanding how software structure evolves. We now move on to discussing how structural metrics can be used to predict non-structural attributes such as bug severity, effort, and defect count.

5.5 Predicting Bug Severity

We present a novel approach that uses graph-based metrics associated with a function or module to predict the severity of bugs in that function or module. When a bug is reported, the administrators first review it and then assign it a severity rank based on how severely it affects the program. Table 5.2 shows levels of bug severity and their ranks in the Bugzilla bug tracking system. A top priority for software providers is to not only minimize the total number of bugs, but to also try to ensure that those bugs that do occur are low-severity, rather than *Blocker* or *Critical*. Moreover, providers have to do this with limited numbers of developers and testers. Therefore, a bug severity predictor would directly improve software quality and robustness by focusing the testing and verification efforts on highest-severity parts.

We use *NodeRank* to help identify critical functions and modules, i.e., functions or modules that, when buggy, are likely to exhibit high-severity bugs. As discussed in Section 5.3.1, *NodeRank* measures the relative importance of a node—function or module—in the function call or module collaboration graphs, respectively. By looking up the *NodeRank*, maintainers have a fast and accurate way of identifying how critical a function or module is. We now state our hypothesis formally:

H1: Functions and modules of higher *NodeRank* will be prone to bugs of higher severity.

Data set. We used six programs: Blender, Firefox, VLC, MySQL, Samba, and OpenSSH for this analysis. We collected the bug severity information from bug reports. For

Bug Severity	Description	Rank
Blocker	Blocks development testing work	6
Critical	Crashes, loss of data, severe memory leak	5
Major	Major loss of function	4
Normal	Regular issue, some loss of functionality	3
Minor	Minor loss of function	2
Trivial	Cosmetic problem	1
Enhancement	Request for enhancement	0

Table 5.2: Bug severity: descriptions and ranks.

each bug report we collected the patches associated with it and from each patch we found out the list of functions that were changed in the bug fix. Therefore, we have information about how many times a function has been found buggy, and what the median severity of those bugs was.

Results. We were able to validate *H1* for our study. We correlated the median bug severity of each function and module with its *NodeRank*. The results are shown in column 2 of Tables 5.3 (for functions) and 5.4 (for modules). As a first step, we focus on nodes with a *NodeRank* in Top 1% since bug severity for functions and modules exhibit a skewed distribution where Top 1% of the nodes are affected by majority of the bugs. Note that for sizable programs such as Firefox, the number of nodes exceeds 25,000, hence even Top 1% can mean more than 250 functions. We find the correlation between *NodeRank* and *BugSeverity* to be high: 0.6—0.86. This suggests that *NodeRank* is an effective predictor of bug severity, and can be used to identify "critical" functions or modules. We have also computed correlation values between function bug severity and standard software engineering

quality metrics (cyclomatic complexity[2], interface complexity[3]). As can be seen in columns 3–4 of Table 5.3 we found these values to be close to zero, meaning that these metrics are not effective in identifying critical functions.

The node degree is not a good predictor of bug severity. We investigated if the node degree could be just as good a severity predictor as the NodeRank. The answer was no. We compute the correlations between bug severity and node in- and out-degrees. The results are shown in Table 5.3, columns 5–6 (functions), and Table 5.4, columns 3–4 (modules); note how in- and out-degrees are poor bug severity predictors.

We also compute the *NodeRank–BugSeverity* correlation for the remaining 99% of the nodes and found similar trends. As expected, in the lower end of the NodeRank, there is significant statistical noise, which makes estimating a correlation coefficient difficult. However, there is definitely a clear high level trend between NodeRank and BugSeverity even in the absence of a well-defined linear correlation. Overall, our work suggests a useful practical approach: in a resource-constrained testing and verification setting, one should start with the nodes with high NodeRank value.

To illustrate this correlation, in Figure 5.9 we present an excerpt from Firefox's call graph. Within each node (function) we indicate that node's *NodeRank*, as well as the average *BugSeverity* for past bugs in that function. As we can see, verification efforts should focus on functions free() and idalloc(), as their *NodeRank*s are high, which indicates that the next bugs in these functions will be high-severity, in contrast to functions arena_ralloc() and huge_ralloc() that have low *NodeRank*s and low *BugSeverity*.

[2]McCabe's cyclomatic complexity is the number of logical pathways through a function [104].
[3]Computed as the sum of number of input parameters to a function and the number of return states

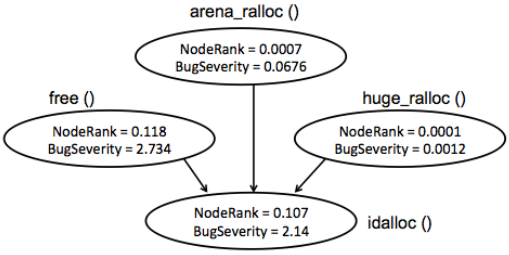


Figure 5.9: Firefox call graph and bug severity (excerpt).

Program	NodeRank	Cyclom. complex.	Interface complex.	In degree	Out degree
Blender	**0.60**	0.07	0.17	0.10	0.05
VLC	**0.83**	0.19	-0.06	-0.09	-0.003
MySQL	**0.77**	-0.05	-0.11	-0.06	-0.06
Samba	**0.65**	-0.207	-0.19	0.23	-0.06
OpenSSH	**0.86**	0.003	0.12	0.04	-0.34
Firefox	**0.48**	0.16	-0.28	0.18	-0.26

Table 5.3: Correlation of *BugSeverity* with other metrics for Top 1% *NodeRank* functions (p-value ≤ 0.01).

5.6 Predicting Effort

A leading cause of high software maintenance costs is the difficulty associated with changing the source code, e.g., for adding new functionality or refactoring. We propose to identify difficult-to-change modules using the novel module-level metric called *Modularity Ratio*, defined in Section 5.3. Intuitively, a module A's modularity ratio, i.e., $Cohesion(A)/Coupling(A)$ indicates how easy it is to change that module. To quantify maintenance effort, the number of commits is divided by the churned eLOC for each module in each release—-this is a widely used metric for effort [15]. Therefore, our hypothesis is formulated as follows:

from that function [142].

Program	NodeRank	In degree	Out degree
Blender	**0.79**	-0.04	-0.0008
VLC	**0.82**	0.21	-0.11
MySQL	**0.73**	-0.20	-0.24
Samba	**0.78**	-0.02	-0.10
OpenSSH	**0.65**	-0.22	-0.19
Firefox	**0.704**	-0.17	-0.38

Table 5.4: Correlation of *BugSeverity* with other metrics for Top 1% *NodeRank* modules (p-value ≤ 0.01).

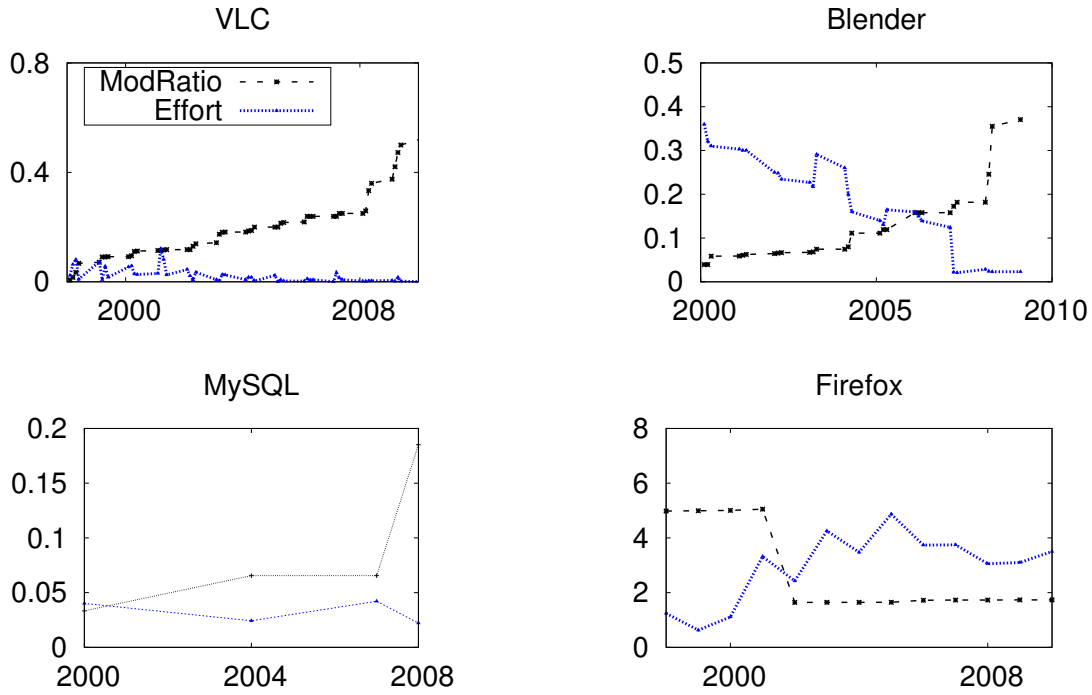


Figure 5.10: Change in *ModularityRatio* with change in *Effort*; x-axis represents time.

H2: Modules with higher *ModularityRatio* have lower associated maintenance effort.

Data set. We used four programs, Blender, Firefox, VLC and MySQL for this analysis. The effort data for these programs is available from our prior work [15].

Results. We were able to validate *H2* for our study. We found that, as the *ModularityRatio increases* for a module, there is an associated *decrease* in maintenance effort for that module, which means the software structure improves. In Figure 5.10 we plot the results for each program. The x-axis represents the version; for each version, in gray we have the mean modularity ratio, and in blue we have mean maintenance effort. We ran a Granger causality test[4] on the data in the graph. We use causality testing instead of correlation because of the presence of time lag; i.e., our hypothesis is that changes in modularity ratio for one release would trigger a change in effort in one of the future releases. As shown in Table 5.5, we obtained statistically significant values of *F-prob* for the Granger causality test on modularity ratio and effort.[5] The lag value indicates that a change in modularity ratio will determine a change in effort in the next release (Blender, Firefox) or in three releases (VLC).

5.7 Predicting Defect Count

Intuitively, a stable, highly cohesive development team will produce higher-quality software than software produced by a disconnected, high-turnover team [23]. Therefore, we are interested in studying whether stable team composition and structure will lead to higher levels of collaboration, which in turn translates into higher quality software. We are

[4]The Granger causality test is a statistical hypothesis test for determining whether one time series is useful in forecasting another.

[5]We cannot claim statistically significant results for MySQL due to small sample size; effort values for only 4 versions of MySQL were available.

Program	Lag	F-prob
Blender	1	0.0000015
VLC	3	0.3673
Firefox	1	0.00056

Table 5.5: Granger causality test results for *H2*.

in a good position to characterize team stability by looking at the evolution of developer collaboration graphs. To measure how much graph structure changes over time we use the edit distance metric defined in Section 5.3.1. Concretely, we hypothesize that periods in software development that show stable development teams will result in periods of low defect count. To test this, we form the following hypothesis:

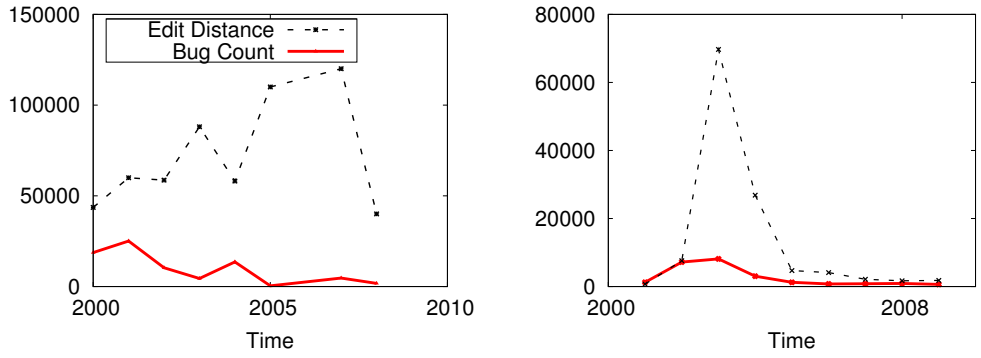
H3: An increase in edit distance in Bug-based Developer Collaboration graphs will result in an increase in defect count.

Data set. We used the Firefox and Eclipse bug reports to build the developer collaboration graphs. For Firefox, we analyzed 129,053 bug reports (May 1998 to March 2010). For Eclipse, we considered bugs numbers from 1 to 306,296 (October 2001 to March 2010).

Results. We were able to validate *H3* for our study. From bug reports, we constructed *Bug-based Developer Collaboration* graphs as explained in Section **??**. We constructed these graphs for each year, rather than for each release, as some releases have a small number of bugs. Next, we computed the graph edit distance from year to year and ran a correlation analysis between edit distance for year Y and defect count at the end of year Y. We found that there is a strong positive correlation between these two measures,

as shown in Figure 5.11. This shows that team stability affects bug count; our intuition is that, when developers collaborate with people they have worked with before, they tend to be more productive than when they work with new teammates. A similar finding has been reported by Begel et al. [11] for commercial work environments: working with known teammates increases developer productivity. Although open source projects lack any social structure and central management, team collaboration does affect software quality.

Bug-tossing based collaboration is a better defect predictor than commit-based collaboration. We have also tested the same hypothesis with commit-based developer collaboration graphs; however we did not find any correlation between edit distance in those graphs and effort, which suggests that bug-tossing graphs are more useful than commit-exchanges for studying developer relationships in open source projects.



(a) Eclipse, correlation = 0.504

(b) Firefox, correlation = 0.759

Figure 5.11: Change in collaboration graph *Edit Distance* v. *Defect Count*.

5.8 Contribution Summary

In summary, our main contributions are:

- We used graphs to model software at two different levels: product level (function-level interaction) and module-level interaction) and process level (bug-based developer collaboration and commit-based developer collaboration).
- We studied the evolution of the graph models of these programs over one to two decades. We found that these graphs exhibit some significant structural differences, and some fundamental similarities. Specifically, some graph metrics vary significantly among different programs, while other metrics captures persistent behaviors and reveal major evolutionary events across all the examined programs.
- We demonstrated how our graph metrics can be used to suggest, infer, and predict important software engineering aspects. In particular, apart from helping researchers construct predictors and evolution models, our findings can help practitioners in tasks such as: identifying the most important functions or modules, prioritizing bug fixes, estimating maintenance effort

5.9 Conclusions

In this chapter we have shown how coupling Network Science with Software Evolution opens new opportunities in software engineering research and practice. We have provided a graph construction method and a set of metrics that capture the structure and evolution of software products and processes. Using a longitudinal study on large open

source programs, we have demonstrated that source code-based graph metrics can reveal differences and similarities in structure and evolution across programs, as well as point out significant events in software evolution that other metrics might miss. We have also shown that graph metrics can be used to predict bug severity, maintenance effort and defect-prone releases.

Chapter 6

Quantifying Contributor Expertise and Roles

Determining a contributor's expertise, ownership and more generally, individual importance, are basic questions for assessing the impact of a contributor in a software project. We argue that currently there is no systematic way to define and evaluate contributor expertise and impact. So far, most previous efforts have used either very narrow, or overly broad definitions and metrics. In this chapter, we operationalize contributor expertise and role. First, we revisit currently-used expertise metrics and we show that they are not very suitable as expertise indicators. The crux of the problem is that these metrics are agnostic to contributor roles, and if we simply combine them, we bundle many different aspects creating a veil of ambiguity. Second, we propose to evaluate expertise and contribution along clearly defined roles, which captures the multiple facets of a software project. Third, we propose an intuitive graph-based model that is based on the contributor

collaboration interactions. Our model captures the hierarchical structure of the contributor community in a concise yet informative way. We demonstrate the usefulness of the model in two ways: (a) it provides the framework for identifying interesting properties of the structure and the evolution of the contributor interactions, and (b) it can help us predict the roles of contributors with high accuracy (up to 76%). We substantiate our study using two large open-source projects, Firefox and Eclipse, with over 20 cumulative years of development and maintenance data. Our systematic approach can clarify and isolate contributor role and expertise, and shed light into the complex dynamics of contributor within large software projects.

6.1 Introduction

We use several questions to illustrate the motivation for our work in this chapter: *Who are the most "valuable" (competent, efficient) developers in a project? Who are the best bug fixers? Who knows best who the right person is to fix a bug?* The overall problem involves determining and assessing the contribution and the expertise of developers in a software project. Specifically, the input to the problem is archived data of contributor interactions, whose nature and level of detail can vary between projects. The requirement is to answer questions regarding expertise, contribution and overall impact of a developer. For example, only a few large open source projects like Mozilla and Eclipse record structured bug activity information, e.g., list of all developers a bug was assigned to, who triaged a bug, and who ultimately fixed it.

Previous work. Despite the significant number of studies on assessing contribu-

tor expertise, authorship, and ownership in software projects, very few studies have really focused on the problem at hand. In more detail, some studies evaluate metrics to deduce expertise [174, 7], which is slightly different from our focus. Other studies examine expertise as an absolute value [113, 52, 111, 22, 136], which is a different aspect of developer. Similarly, although contributor collaboration as a form of social networking has been studied in software engineering [20, 21, 139, 138, 107, 70], extracting a *hierarchy* structure from this collaboration has not been investigated so far. We discuss previous work in more detail in Section 8.4.

An example. The following example can provide some intuition of what we are trying to achieve. Let us consider the case of two contributors D_1 and D_2 from the Mozilla community.[1] Using widely-used metrics, we find that D_1 and D_2 have roughly the same *percentage of bugs fixed from those assigned* to them, with 49.15% and 53.09% respectively. Both D_1 and D_2 have similar *seniority*, having been with the project for 8 and 10 years, respectively. These numbers would lead someone to believe that D_1 and D_2 exhibit comparable expertise and play similar roles in the project. However, upon closer examination, with more refined role definitions, we find that D_1 serves the role of triager (an individual who assigns a new bug to a developer) while D_2 served the role of a patch tester (an individual who reviews and tests patches submitted for fixing a bug). In fact, our approach is able to make this distinction, as we discuss later.

Contributions. In this chapter, we operationalize (i.e., develop a systematic approach for defining and determining) contributor expertise and impact. We start by

[1]We withhold actual names for preserving anonymity.

revisiting the whole process starting from what we want to capture and how we can capture it systematically. Our key novelty is that we start from defining roles that capture fundamental software development functions and then build our framework to assess contributions along these roles.

Our main contributions can be summarized as follows:

a. Quantifying the inadequacy of current metrics. We revisit previous expertise metrics: we show that each metric captures a local notion of expertise by quantifying a specific development activity (e.g., LOC added) but when put together, they fail to capture a global notion of expertise (Section 6.2.2). The crux of the problem is that these metrics are agnostic to contributor roles, and if we simply combine them, we bundle many different aspects creating a veil of ambiguity. In fact, despite spending significant effort, we were unable to define expertise in a meaningful way using these metrics.

b. Defining developer roles. We propose to assess expertise and contribution along roles which clarifies the confusion. For example, an expert bug fixer is not necessarily a expert bug triager, but both are equally important for a project. We introduce a set of roles: Patch tester, Assist, Triager, Bug analyst, Core developer, Bug fixer, Patch-quality improver. We also provide ways to define these roles rigorously, assuming that we have access to the Role profile, which only some projects maintain (Section 6.3).

c. Proposing an intuitive graph-based model of developer contribution. We develop graph-based model that captures concisely useful information, and we refer to it as **Hierarchical Contributor Model (HCM)**. A key observation is that there exists a clique that consists of the highest degree nodes, which is a natural choice for the core of the

graph. With this clique as its first tier, the model identifies a limited number of tiers (3-4 in total) in graphs with up to 10K contributors, as as we see in figure 6.6 (Section 6.4). HCM concisely represents the contributor interaction, in a way that captures hierarchy, role and "importance" of contributors, which is hard to do with previously defined expertise metrics. We then show how one can benefit from our model: we use it as a framework for identifying interesting properties of the structure and the evolution of the contributor interactions, and show that it can even help us predict the roles of contributors (Section 6.5).

Scope. The work is motivated by both practical software engineering concerns and the need to model software development as an evolving complex systems. Our study can help distinguish different types of people and for different functions. First, managers may want to assess developer contribution in objective ways, for say rewarding key people; the counterpart in the open-source world is "promoting" developer and give them commit privileges. Second, supervising developers can use our approach to identify weaknesses in the development process, e.g., single nodes of failure, imbalance in the flow of development. Finally, from a software maintenance perspective, we want to address a long-standing challenge: detecting intrinsic emerging patterns in large long-term software projects [108, 1, 140].

6.2 Data Collection and Processing

6.2.1 Data Collection

We used data from the Eclipse and Firefox projects for our study. Our approach makes use of information from source code, patches, change logs, and bug reports. We

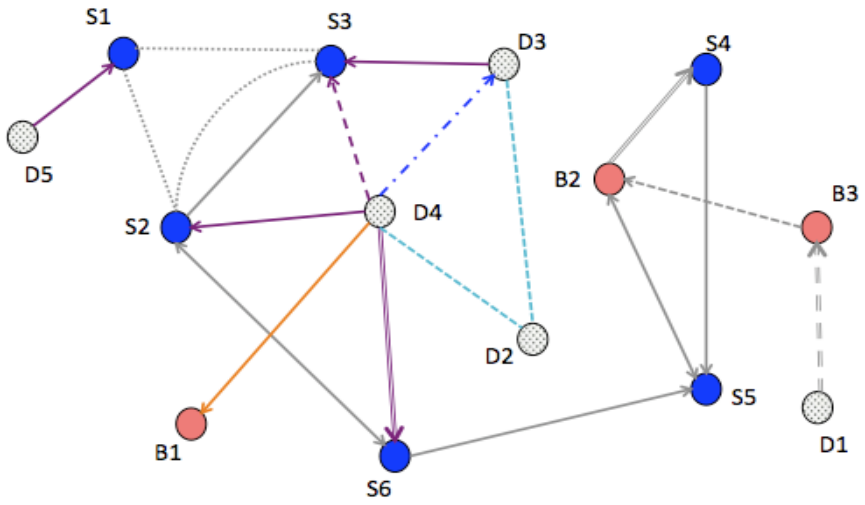


Figure 6.1: Hypergraph extraction for characterizing roles of contributors.

analyzed the entire histories of these applications, from inception (2001 for Eclipse, 1998 for Firefox) up to April 2010. We use source code information, available in the version control system, to construct source code-based expertise profiles. For each source file, we extract its contributors along with the timestamps of their contributions, and diffs (patches) for each commit. For source code, we analyzed Eclipse versions 1.0 to 3.6.1; for Firefox we analyzed versions 0.8 to 3.6. We use bug report information, available in the bug tracker, to construct bugfix-based expertise profiles. For each bug report, we extract the sequence of assignees and comments. We considered Eclipse Platform bug numbers 1 to 306,296 and Firefox bug numbers 37 to 549,999 (as recorded in Mozilla's Bugzilla bug tracker). In Table 6.1, we present a quick overview of the data we collect and their respective usage. In Figure 6.1 we show the nodes and edges we extract from the multi-mixed graph (explained in Section 2) for our analysis in quantifying contributor role. Formally, we can represent the hypergraph extracted as shown in Figure 6.1 as:

$$\{(v_1, v_2)|(v_1, v_2) \in \mathbb{G}_{InterRepoDep}|v_1 \in v_{contributor} \wedge v_2 \in (v_{bug} \vee v_{func} \vee v_{mod} \vee v_{contributor})\}$$

6.2.2 Expertise Profiles and Metrics

In this section, we introduce **expertise profile** and provide details on the process and metrics we used to construct these profiles for each contributor in our examined projects. For each member D of a project, we define two kinds of expertise profiles: a *bug-fixing profile*, and a *source code profile*. The rationale for using two profiles is to capture the two major ways of contributing to project—bug-fixing or development—especially in open source projects. Moreover, many large open source projects use separate systems for version control and bug tracking, so we had to perform entity resolution to determine when version control id C_D and bug tracker id B_D correspond in fact to the same individual D (Section 6.6).

We define the **bug-fixing expertise profile** of a contributor D as a tuple ($bugcount_D$, $bugsev_D$, $bugseniority_D$, $bugsfixed_D$) where $bugcount_D$ is the total number of bugs D has been associated with, i.e., D was assigned to fix at some point in time; $bugsev_D$ is the average severity score of all bugs D has fixed;[2] $bugseniority_D$ represents the first and last times D has fixed a bug, as recorded in the bug tracker, and $bugsfixed_D$ is the percentage of bugs D could fix, relative to the total number of bugs assigned to D.

We define the **source-code expertise profile** of a contributor D as a tuple: ($codelines_D$, $files_D$, $codeseniority_D$, $ownership_D$) where the contents are defined as follows: $codelines_D$ is the number of lines of code D has committed—we identify all the

[2]Firefox and Eclipse use a 1-to-7 scale for bug severity (1=Enhancement, 2=Trivial, 3=Minor, 4=Normal, 5=Major, 6=Critical, 7=Blocker).

Source	Raw Data	Expertise Profile (Section 6.2)	Role Profile (Section 6.3)	HCM (Section 6.4)
Bug tracker (Bug report, Bug activity)	Contributor ID	✓	✓	✓
	Timestamp	✓	✓	✓
	Severity	✓		
	Task - triaged		✓	
	Task - tested		✓	
	Task - assisted		✓	
	Task - analyzed		✓	
	Type (defect, enhancement)	✓		
Source Code Repository (Commit Logs)	Committer ID	✓	✓	✓
	Log message		✓	
	LOC added	✓		
	Timestamp	✓	✓	✓
	Files changed	✓	✓	✓
	Bug/Enhancement ID		✓	

Table 6.1: Data collection sources and uses.

patches D has submitted, and from each patch we extract the number of source code lines added or changed. The rationale for using this metric is that the more source code D has contributed, the higher D's expertise level is; $files_D$ is the number of files D has worked on. The rationale for using this metric is that the more files D has worked on, the higher D's expertise level is. We define $codeseniority_D$ as the time difference D's first and last commits, as recorded in the project's version control system. Note that if D has only committed once, in our definition D's seniority is zero. A contributor to a software module is someone who has made commits to the module. To quantify the level of involvement between a contributor D and a module C, we define *ownership ratio* as the ratio $R_{DC} = \frac{LOCcommitted_D(C)}{LOCcommitted_{total}(C)}$, i.e., the percentage of lines of code committed by D to C relative to the total number of lines of code committed to C. Note that our definition of ownership is different from Bird

et al.'s [22] (which uses the proportion of number of commits) because in the projects we studied we found very low correlation, 0.1403, between the number of commits and the lines of code committed. For each module, based on the ownership ratio R_{DC} we define D's ownership profile (owner, major or minor contributor) in the following table; the first line indicates the ratio, the second line indicates the profile.

$R_{DC} < 5\%$	$5\% \leq R_{DC} < Highest$	$Highest$
Minor contributor	Major contributor	Owner

We have chosen the 5% cut-off based on the cumulative distribution function (CDF) and the cut-off point might vary from project to project.

6.2.3 Correlation Between Expertise Attributes

We now present statistical evidence that our expertise attribute selection is precise (i.e., all constituent attributes of expertise profiles in Sections 6.2.2 and 6.2.2 are relevant and there are no redundant attributes). Correlation-based feature selection is a standard machine learning method for filtering out features (attributes) which have strong correlation between them, thus retaining only those features that are independent of each other [62]. We selected a wide range of features (4 for bug fixes, 6 for source code).[3] To ensure that we only select representative features we ran a Pearson's correlation test between all pairs of source-code based and bug-fix based expertise attributes. We report the results in Table 6.3. We found low pairwise correlation between most features at a statistically significant p-value of 0.01, hence we conclude that, for the projects we considered, feature selection is precise.

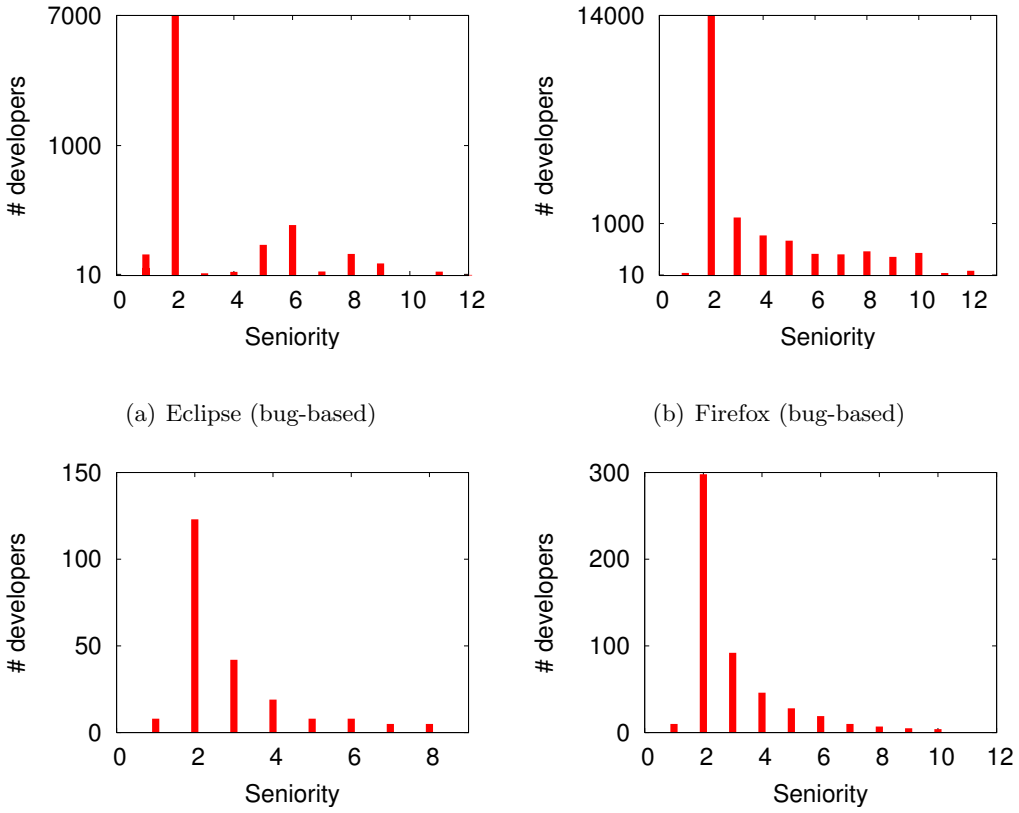
[3]We show the correlations among three types of ownership as explained in Section 6.2.2.

Pairwise correlation values								
	bugseniority		*bugsev*		*bugcount*		*bugsfixed*	
	Eclipse	Firefox	Eclipse	Firefox	Eclipse	Firefox	Eclipse	Firefox
bugseniority	1		0.2728	0.2540	0.1481	0.2531	0.0068	-0.0362
bugsev			1		0.3731	0.4264	0.1137	0.0182
bugcount					1		0.0028	0.0604

Table 6.2: Correlation between bug-fixed profile attributes (p-value ≤ 0.01 in all cases).

Pairwise correlation values												
	codeseniority		*codelines*		*files*		*owner*		*major*		*minor*	
	Eclipse	Firefox	Eclipse	Firefox	Eclipse	Firefox	Eclipse	Firefox	Eclipse	Firefox	Eclipse	Firefox
codeseniority	1		0.4722	0.3637	0.2604	0.3119	0.3114	0.0138	0.1434	0.0525	0.0107	0.1155
codelines			1		0.4231	0.7063	0.1781	0.4578	0.0248	0.0978	0.0034	0.0218
files					1		0.1620	0.1751	0.0766	0.0439	0.1178	0.2796
owner							1		0.0593	0.1584	0.1396	0.0037
major									1		0.0446	0.0895

Table 6.3: Correlation between source-code profile attributes (p-value ≤ 0.01 in all cases).



(a) Eclipse (bug-based)

(b) Firefox (bug-based)

(c) Eclipse (source-based)

(d) Firefox (source-based)

Figure 6.2: Bugfix-induced and source code-induced seniority.

6.2.4 An Empirical Study of Contribution

With the expertise profiles in hand, we now proceed to conduct an empirical study to understand how the contribution profile of contributors in a project can help characterize various facets of software development.

1. ***When do contributors join a project, and how long they stay associated with a project, for fixing bugs and developing code?***

Figure 6.2 shows the cumulative distributions for bugfix- and source code-based se-

niority in our examined projects. In each graph, the x-axis shows seniority, in years, and the y-axis shows the cumulative number of contributors; each (x, y) pair on the graph shows the number of contributors y that have seniority at most x years.

Figures 6.2(a) and 6.2(b) show bugfix-induced seniority distributions in Eclipse and Firefox. Eclipse has had 8856 bug fixers over its lifetime; 80.79% of those have seniority less than one year; Firefox has had 19286 bug fixers over its lifetime; 75.18% of those have seniority less than one year. These values are quite interesting in that they reveal the high turn-over in these projects. Figures 6.2(a) and 6.2(b) show source code-based seniority distributions in Eclipse and Firefox. Eclipse has had 210 contributors over its lifetime; 58.57% of those have seniority less than one year; Firefox has had X bug fixers over it lifetime; 58.54% of those have seniority less than one year. An interesting aspect, when comparing the bugfix- with the source code-induced seniorities is the shapes of their cumulative distribution curves. The flatter source code seniority curves indicate lower source-code turnover, meaning that contributors who work on source code tend to be attached to the project for longer than contributors who work on bug fixes.

2. ***What is the contribution distribution in large projects?***

We report five relevant observations about the contributor contribution distribution for Firefox and Eclipse:

- On average, 16.26% and 21.91% contributors for Firefox and Eclipse respectively work on the same file.

- On average, a contributor works on 2.58% and 3.706% files for Firefox and Eclipse respectively during his lifetime.
- 14.26% files in Firefox and 11.71% files in Eclipse have a single contributor working on it.
- 27.56% and 18.15% contributors for Firefox and Eclipse respectively have been owners of at least one file.
- 16.72% and 7.32% contributors for Firefox and Eclipse respectively have been authors of at least one file.

3. ***How defect prone are files maintained by their original authors?***

   Defect density of files maintained [4] by the original authors of the file are lower than the defect density of files that do not have an author or not maintained by the author. Our hypothesis *H1* is that files maintained by original authors are less prone to error compared to other files. We accept *H1* at 1% significance level (t-value = 3.234 for Firefox and t-value = 2.176 for Eclipse) after performing a Welch's t-test[5] on the defect density of the two types of files - one maintained by original authors (mean value = 0.0026 for Firefox and 0.1637 for Eclipse) and the other not maintained by authors (mean value = 0.0319 for Firefox and 0.2316 for Eclipse).

4. ***Are files with an owner less prone to defect compared to files with no***

[4] Source code files contain author information (name and/or email ID) as *Original Authors*. If the same person later commits to the file either during bug fix or feature enhancement, we consider that the file is maintained by the author.

[5] *Welch's* t-test [166] returns a t-value for a fixed level of statistical significance and the mean values of the sample sets. In our study we only consider 1% statistically significant t-values, to minimize chances of *Type I* error. According to standard t-test tables, the results are statistically significant at the 1% level if t-value ≥ 2.08.

owner?

Our hypothesis *H2* is that files that have an owner are less prone to error compared to other files with only minor contributors. We accept *H2* at 1% significance level (t-value = 2.794 for Firefox and t-value = 2.612 for Eclipse) after performing a Welch's t-test on the defect density of the two types of files: one which has an owner (mean value = 0.0042 for Firefox and 0.0031 for Eclipse) and the remaining which do not have any owner or major contributor (mean value = 0.0747 for Firefox and 0.1064 for Eclipse).

5. ***How does the contribution distribution for a module affects the defect density?***

   Our hypothesis *H3* is that defect density of a module is correlated with the number of minor contributors to the file. We found a high correlation (p-value < 0.01 for all experiments) (0.773 in Eclipse and 0.612 in Firefox) between the number of minor contributors for a module and the defect density of a module. However, we found a low correlations between the defect density of a module and the number of owners (0.051 in Eclipse and 0.118 in Firefox) and number of major contributors (0.427 in Eclipse and 0.388 in Firefox) in a file. This result is significant because previous studies on open source projects reported that total number of contributors working on a module is highly correlated with the defect density of the module. On the other hand, we show that number of *minor* contributors affect defect density irrespective of the total number of contributors.

6. ***Do bugfix-induced seniority correlate with the number of bugs fixed and the average severity of those bugs?***

A common belief in software development is that senior contributors in a project will fix more bugs, will fix bugs of higher severity and will fix a higher percentage of bugs that are assigned to them when compared to junior contributors. However, this conjecture has never been empirically studied. We found that seniority correlates with average bug severity and total number of bugs involved with, but does not correlate with the percentage of bugs fixed by a contributor (refer to rows 3 and 8 in Table 6.3). Note that although the correlation values are smaller compared to the standard scale for reporting positive correlation, in statistics the interpretation of a correlation coefficient depends on the context and purposes [38]. As shown in Fig 6.2, 75-80% of contributors have less than 1 year of project involvement. To test this hypothesis better, we divide the contributors in to two groups based on seniority: (1) contributors whose seniority is less than or equal to 1 year, and (2) contributors whose seniority is greater than 1 year. We form two sub-hypotheses: $H4_1$ is that the average bug severity of bugs fixed by contributors in category (2) is higher than the average bug severity of the bugs fixed by contributors in category (1). We accept $H4_1$ at 1% significance level (t-value = 4.361 for Firefox and t-value = 2.485 for Eclipse) after performing a Welch's t-test on the average bug severity of contributors in categories (1) and (2). $H4_2$ is that the number of bugs fixed by contributors in category (2) is higher than the number of bugs fixed by contributors in category (1). We accept $H4_2$ at 1% significance level (t-value = 2.253 for Firefox and t-value = 2.682 for

Eclipse) after performing a Welch's t-test on the number of bugs fixed by contributors in categories (1) and (2).

6.3 Contributor Roles

In software projects, an individual's contributions involve more than adding code or fixing bugs. In this section, we operationalize seven roles that capture several aspects of software engineering. Although we do not claim that they capture all facets of open source software engineering, we argue that they are a good start towards a systematic framework, that we develop here. Role operationalization is based on the nature and timing of developer contributions. For now, we do not assign any threshold to the frequency of contribution to mark a contributor as an active participant for a specific role; in the future, we intend to vary the threshold and evaluate its effect on our analysis.[6] The roles are not mutually exclusive, and a contributor can serve multiple roles during her association with a project. Next, we define each of these roles that form the bug- and source-code based profiles.

1. ***Triagers:*** contributors who triage bugs are indispensable in large projects that receive hundreds of new bug reports every day [17]. A triager's role ranges from marking duplicate bugs, to assigning bugs to potential contributors, to ensuring that re-opened bugs are re-assigned, and to closing bugs after they have been resolved. *In our case, by triaging we refer to contributors who inspect the bug report and identify potential bug fixers (i.e., other contributors who could fix the bug).* As shown in our previous work [17], choosing the right bug fixer is both important and non-trivial: when a

[6] We list this as a potential threat to validity in Section 6.6.

bug assignee cannot fix the bug, the bug is "tossed" (re-assigned), which prolongs the bug-fixing process. Jeong et al. [76] introduced the concept of *tossing graphs*, where nodes represent developers and edges represent bugs being tossed; a bug's lifetime (assignment, tossing, and fixing) can be reconstructed from its corresponding tossing path in the tossing graph. We identify a triager as the first contributor in the tossing path to "assign" a bug to another contributor. Note that bugs with assignee status "Nobody's OK to work on it" might have contributors assigning themselves as the bug-fixer. We do not consider self-assignment as triaging.

2. ***Bug analysts:*** contributors who help in analyzing the bugs play an important role in the bug-fix process. *In our study we label contributors as bug analysts if they perform one of the following decision-making tasks*: (1) prioritizing bugs, (2) deciding when to assign "won't fix" status to bugs, (3) labeling a feature enhancement for future release, (4) identifying duplicate and invalid bugs, and (5) confirming a new bug as a valid bug by reproducing the errors as per the description in the bug report. We identify bug analysts from the activity page of a bug.

3. ***Assists:*** *contributors who have found the right person to fix a bug at least once are defined as assists in our model.* This role captures the "*who knows who knows what*" relationship in a social network. In our case, an assist is the second-to-last person in the bug tossing path, i.e., D is an assist if D assigned the bug to a contributor E (after it was tossed among other contributors) and E finally fixed the bug. Note that, to identify assists, we only consider bugs whose the tossing path length is greater than one.

4. ***Patch tester:*** after a patch (potential fix) for a bug has been submitted, certain contributors test the patch, to ensure that the bug has been correctly resolved. We identify these contributors, i.e., patch testers, from bug report comment sections where they report their results from testing the newly-submitted patch.[7] Additionally, a tester may also suggest ways of improving the code quality. Patch testers are identified from the activity page and the comment section of a bug report where they are first assigned as the patch-reviewer and in the next activity they report the test results with often with suggestions for improvement. *In our model, a patch tester is a developer who has reported patch testing results at least once.*

5. ***Patch-quality improvers:*** patch-quality improvers are contributors who help improve the quality of patches submitted for fixing a bug by other contributors. These improvements involve adding documentation, cleaning up the code, ensuring compliance with coding standards, etc. We label a contributor D as a patch quality improvers if we find that D modifies newly-submitted patches and the log message contains strings such as "cleaned patch", "added documentation", "simplified string definition", "changed incorrect use of variable", etc. Identifying such messages automatically for large source-code repositories like Firefox and Eclipse is a non-trivial task. We used a text mining technique to extract such messages: (1) identify all log messages that are submitted with the same bug-ID for the same file, (2) sort the log messages chronologically, and (3) check if the log messages submitted after the initial patch contain words like "added", "changed", "simplified", "removed", "reverted",

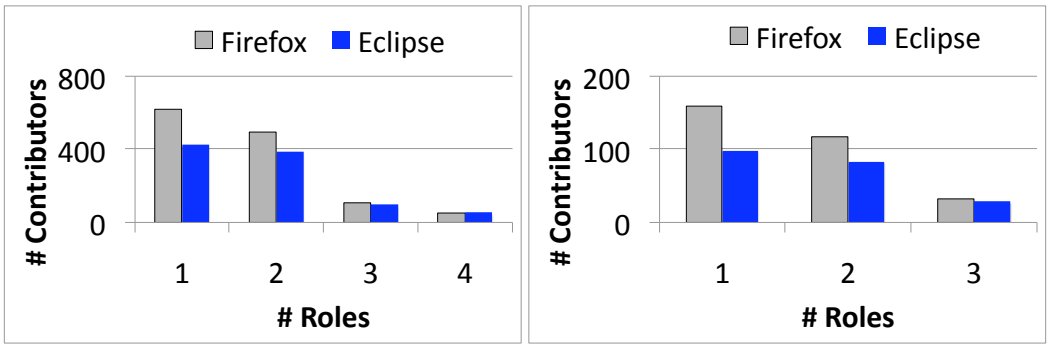
[7]For example, Mozilla bug 50212 (`https://bugzilla.mozilla.org/show_bug.cgi?id=50212`) shows how a contributor X plays the role of tester for contributor Y (comment 4).

"replaced", "rewrote", "updated", "renamed", etc. *In our model, a patch-quality improver is a developer whose log messages contain the aforementioned keywords at last once.*

6. ***Core developers:*** *we define a core developer as a contributor who has added new code to the source code repository in response to a feature enhancement request or has added code that does not correspond to a bug-fix.* This way we ensure separation between contributors who perform adaptive/perfective maintenance and those who perform corrective maintenance (bug-fixers, described below).

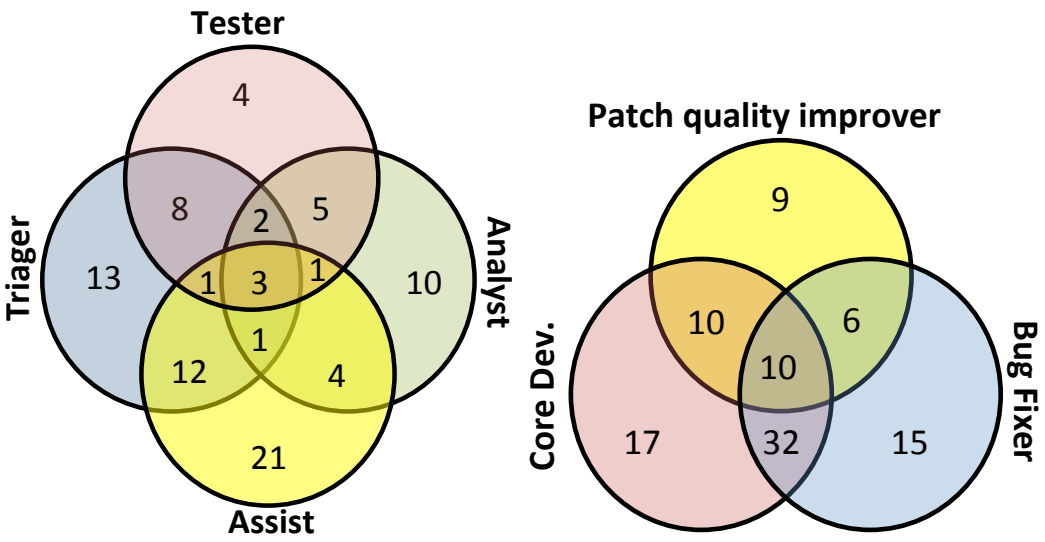
7. ***Bug fixers:*** *we tag a contributor D as a bug-fixer if D has added code for fixing a bug.* In other words, a bug fixer performs corrective maintenance. A commit is identified as corrective maintenance by cross-referencing the bug ID associated with the log message with the bug type (i.e., defect or enhancement) in the bug database.

Analysis of role distributions: to illustrate how contributors serve multiple roles in a project, we provide three analyses. In Figure 6.3 we show the absolute number of contributors (y-axis) who have served one or more roles (x-axis); as expected, the bulk of contributors have only served one role, with much smaller numbers serving all 4 bug-based roles or all 3 source-based roles. In Figure 6.4 we show the distribution and overlap of roles, in percentages, for Firefox; we now proceed to explain the graph. For example, among those developers who have ever served source-based roles: 17% have only served as core developers, 32% have served as both core developers and bug fixers, and 10% have served all three roles. In Figure 6.5 we characterize the frequency distributions for each



(a) Bug-based roles (b) Source-based roles

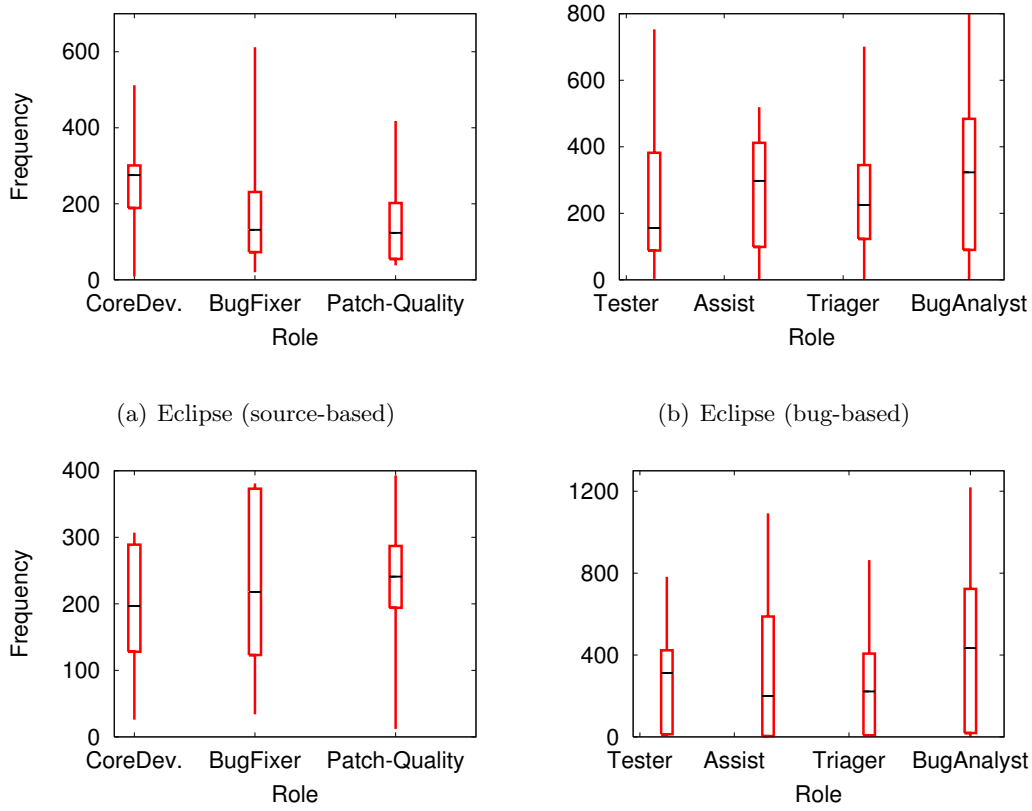
Figure 6.3: Role frequency.



(a) Bug-based roles (b) Source-based roles

Figure 6.4: Role distribution (in percentages).

role, across all contributors for that role. Each candlestick represents the minimum, first quartile, second quartile, third quartile and maximum; the black bar is the median. For example, for core developers in Eclipse (leftmost candlestick in Figure 6.5(a)): the minimum



(a) Eclipse (source-based)

(b) Eclipse (bug-based)

(c) Firefox (source-based)

(d) Firefox (bug-based)

Figure 6.5: Frequency of contribution for each role.

number of role servings was 9, the median number was 275, and the maximum number was 512.

Role profile: we define two kinds of role profiles of a contributor D: bug-based role profile and source code-based role profile. Bug-based role profile is a tuple $\langle Triager, BugAnalyst, Assist, PatchTester \rangle$ and source code-based profile is a tuple $\langle CoreDeveloper, BugFixer, PatchQualityImprover \rangle$. Each metric can have a integer value; if the value is zero, it indicates that the contributor have never served the role or

else any non-zero value would indicate number of times she has served the role (frequency of contribution). For example, bug-based role profile $(D) = \langle 2, 0, 3, 5 \rangle$ would imply that contributor D has served the role of triager twice, never served the role of a bug analyst, served the role of an assist 3 times and the role of a patch tester 5 times.

6.4 HCM: our Graph-Based Model

With the contributor role definitions in hand, we now proceed to defining a *hierarchical contributor model* (HCM) that emerges from the collaboration among contributors in the course of source- and bug-based collaboration. The model has several key advantages: (a) it captures the hierarchy and "importance" of contributors, in a way that was hard or impossible to do with conventional expertise metrics, (b) we show how, by using our model, we can in fact infer roles and contributions with relatively high accuracy using raw data: source-code repository and bug database, without the need of the role profiles, which is not always available for all software projects, (c) it captures the stability of developer interaction. Note that the model we develop here is inspired by our earlier work on identifying structure in the Internet topology [147], as we further discuss in Section 8.4. The model is based on two collaboration graphs: bug-based and source-code based collaboration graphs (as defined in Section 2.2.1).

The emerging structure: hierarchy and tiers. We want to identify structure in our two types of collaboration graphs. The graph mining literature offers many ways to analyze graphs structure, e.g., in terms of clusters or importance. Given our interest in identifying "importance" of contributors, we use the following insight: important contrib-

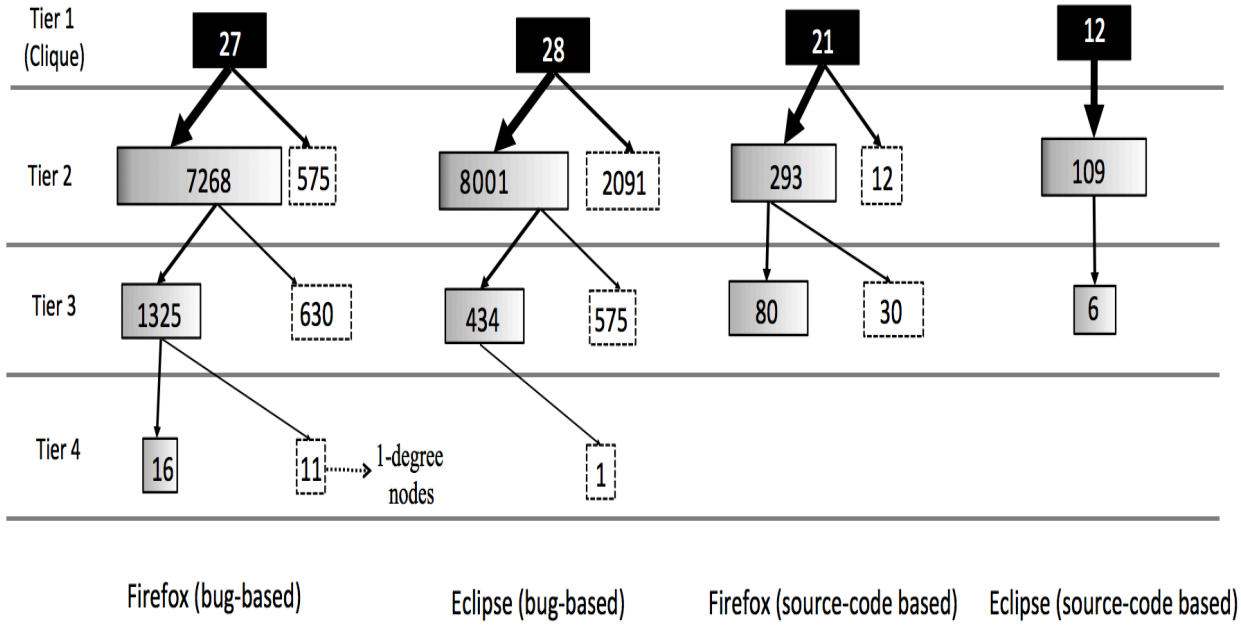


Figure 6.6: HCM (hierarchy and tier distributions) for bug-based and source-based contributor collaborations.

utors are likely collaborating with many other contributors. With this in mind, we follow the process below, which was also used successfully in a different context [147].

1. Determining the "center." Intuitively, we want to identify the *largest* clique with the highest degree nodes in the graph. The process starts from the highest degree node, and includes nodes in order of decreasing degree, until no more nodes can be added to the clique. We refer to the nodes in the clique as *Tier 1*, as shown in Figure 6.6.

2. Determining tiers recursively. Given the definition of tier 1, we define subsequent tiers using connectivity to the previous tier. Specifically, we use the following recursive procedure: a node belongs in tier k if and only if it is connected with at least one node in tier $k-1$.

3. Distinguishing one-degree nodes. In an additional step, we add more information to our model by reporting one-degree nodes (nodes with only one edge) within each tier.

Compact and informative representation. The above process leads to a compact representation of our graphs, as shown in Figure 6.6: the HCMs for bug-based and source-code-based collaboration graphs. To convey more information, we introduce two features. First, we use darker shades to indicate tighter connectivity internally within each box: black signifies the clique, while the one-degree hanging nodes to the right are white, which represents no connectivity. Second, the width of the edges between two tiers is proportional to the number of edges across these tiers.

Why is this model useful? The advantage of the HCM lies in its simplicity and the amount of information that it can "encode": intuitively, one can say that it maximizes

the ratio of information over model complexity. Some observations can be readily made by examining the model: (a) there exists a clique of non-trivial size (12–27 or 0.26%–9.44% of the nodes), (b) there are relatively few tiers, between 3–4 in our graphs, even though the graphs have more than 10,000 nodes, (c) there is a non-trivial number of one-degree nodes (12%, 24%, and 10%, for the graphs, except a really small one for Eclipse source-code), and (d) many one-degree nodes connect to the tier 1 nodes[8] One of the most important properties of the HCM is discussed below, while we discuss uses of our model in Section 6.5.

The model structure is "aligned" with contributor expertise and contribution. High-performing contributors tend to be in lower (numerically) tiers and thus higher in the hierarchy. In Figure 6.7, we present the results of contributor expertise distribution across the various tiers formed for selected metrics. We find that tier 1 nodes (the clique) have high-values of expertise attributes and the values decrease as we move on from tier i to tier $i+1$. In other words, tier 1 contributors are among the most active and experienced contributors for a project. We argue that the tier of a contributor is a good estimate of his expertise and contributions: a member of the clique is likely a senior contributor, who has fixed many bug types, high-severity types, owns many files; conversely, we expect a contributor from tiers 3–4 to be a junior contributor with low expertise. We substantiate this claim in Section 6.5.

An interesting observation is that nodes in tier 2 are strongly connected with nodes in tier 1. For example, in Firefox, 68.79% of the nodes in tier 2 connect to at least 74.07%

[8]In fact, in our graphs, we find that their assortativity is negative or around zero (between -0.3 and 0.011). Positive assortativity coefficient indicates that nodes tend to link to nodes similar degree. This observation contradicts a natural inclination to assume that high degree nodes are in higher tiers, which happens in some hierarchical systems. We also analyzed the degree distribution of the nodes and found that there is no clear intuitive pattern, such as a power-law or other scale-free network properties.

of the members of the clique (tier 1). We believe that this strong connectivity indicates two collaboration traits: (1) tier 2 contributors work very closely with senior contributors for maintaining the project, and (2) tier 2 contributors are stable[9] members of the project.

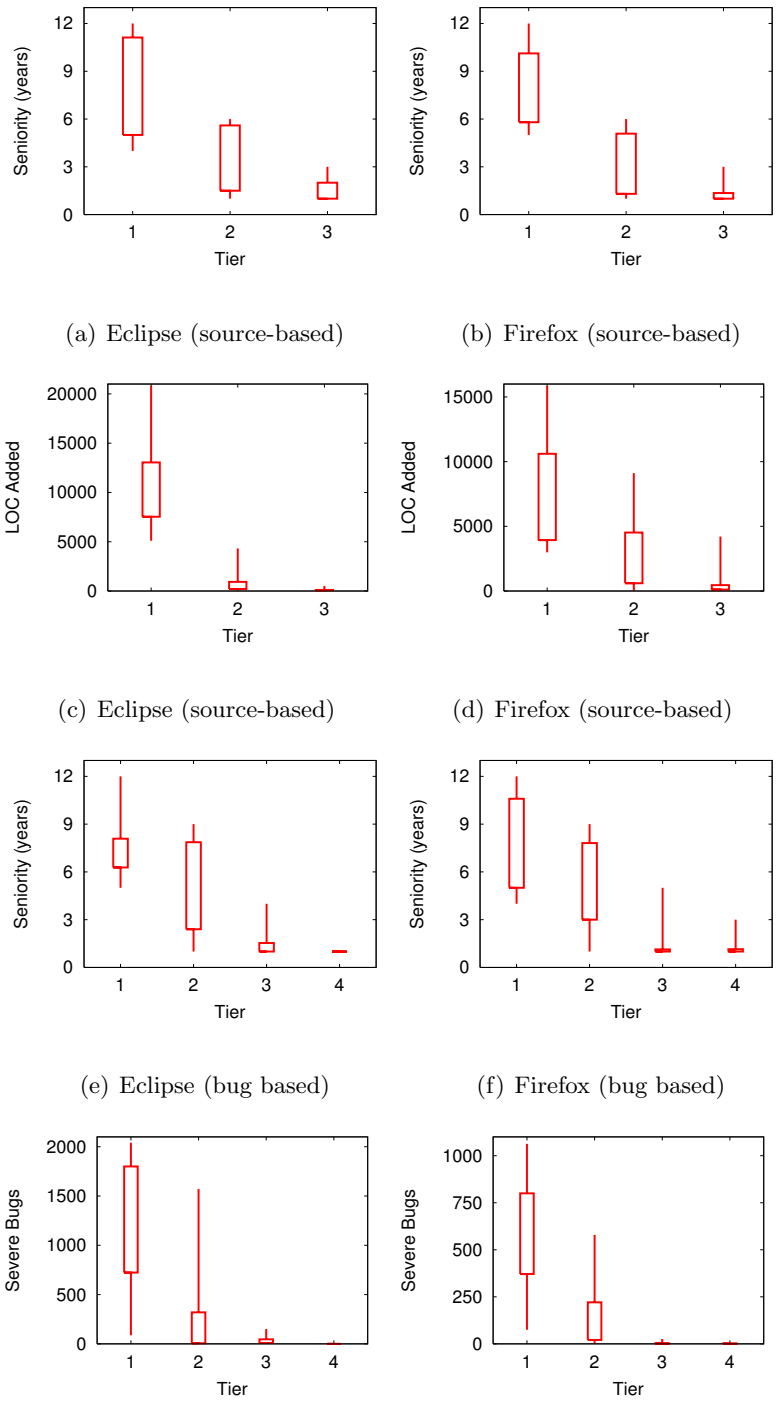
Disconnected nodes. We use connectivity to establish our model, and we find that more than 95% of the nodes form a large connected component, and thus represented in our model. The remaining nodes ($< 5\%$) are disconnected from this connected component, and form mini-graph structures with 2–13 nodes each. We found that all these nodes have seniority one year, which is the lowest, and their expertise profiles are low (e.g., $bugseniority \leq 2$, $bugcount \leq 7$, $bugsev \leq 1.44$).

6.5 Using the HCM Model

In this section, we show how the HCM can help us conduct studies that can reveal interesting properties in terms of structure and evolution of the collaboration relationships, We also show how we can use the concise information encoded in the model to predict the roles of contributors.

Based on the HCM, we study additional features pertaining to collaboration intensity and HCM's evolution in time, which reveal several interesting properties.

[9]It is important for managers in open source projects to identify which project members are stable. Our techniques can help managers identify these stable contributors easily, as they are nodes with high in- and out-degrees.



(a) Eclipse (source-based)

(b) Firefox (source-based)

(c) Eclipse (source-based)

(d) Firefox (source-based)

(e) Eclipse (bug based)

(f) Firefox (bug based)

(g) Eclipse (bug based)

(h) Firefox (bug based)

Figure 6.7: Tier distribution range per expertise metric.

Intensity of Collaboration

We refine our HCM by considering the intensity of the collaboration between contributors, which we represent as a weight associated with their common edge. Here we use the directed graph, as defined in the previous section.

Intense collaborations are within tier 1 and tier 2 nodes. We analyze how strongly the graph is connected considering collaboration intensity. We define a weight threshold t_{cut}, which we use to filter out all the edges with weight $w \leq t_{cut}$. For $t_{cut} = 1$, 72.77% nodes in Firefox and 64.93% of nodes in Eclipse become disconnected from the initial HCM graph. We then increased our threshold $t_{cut} = 2, 3, \ldots$ and observed an interesting phenomenon. The original connected component shrinks significantly if we remove edges with $w < 3$ for Firefox and $w < 5$ for Eclipse, but after that, even if we increase t_{cut}, (till $t_{cut} = 118$ for Eclipse and $t_{cut} = 206$), the connected component do not change. We find that the connected component consists of contributors only from tier 1 and tier 2, which are connected with high weight edges. This shows that the majority of collaborations take place between the top two layers of the network model. This agrees with our results in Section 6.4, where we found that apart from fixing bugs, contributors from these layers serve multiple non-technical roles.

Detecting sub-project communities. We showed that the clique and nodes from tier 2 form a community of intense interactions. Here, we want to find if there are communities of contributors within a large project. The idea is that nodes working on a sub-project and will be forming a community. So now, we focus on the nodes that were isolated from the connected component in the process above with $t_{cut} = 3$ for Firefox,

and $t_{cut} = 5$ for Eclipse. For Firefox, we find four communities are formed with 83.92% (accounting for 23.31%, 11.46%, 18.72% and 30.41%) of the originally discarded nodes, while the remaining nodes form significantly smaller graphs (3-14 nodes), or singletons or pairs. These four larger communities have clustering coefficients between 0.34 and 0.62, which indicates significant intra-cluster cohesion. Further analysis revealed that these communities typically work on different components of the project. For example, in the community of size 23.31% that we mentioned above, all the member nodes have worked on four common components: Security, Preferences, Phishing Protection and Private Browsing. We repeated the same process with Eclipse, and we found qualitatively different results. There is only *one* such emerging community, with 57.03% of the initially discarded nodes. Furthermore, the clustering coefficient for the nodes in the connected component is 0.02, which shows really low cohesion. To explain this low cohesion, we looked at the profiles of the contributors and we could not find a unifying theme, unlike the Firefox community.

Evolution of collaboration graphs

To understand how the collaboration graphs evolve over time, we built three snapshots for years 2006, 2008, 2010 for both Firefox and Eclipse.[10] We found that from 2006–2010, the size of the graphs doubled for Firefox and tripled for Eclipse. By further studying the HCM model of each instance, we find three interesting aspects:

The clique (tier 1) grew significantly. In Firefox, the clique size grew by a factor of 9, from 4 to 11 to 27 contributors. In Eclipse, the clique size grew by a factor of

[10]Both Firefox and Eclipse had their first official release in 2004. The number of contributors has steadily increased since then. We chose 2006 as our starting point to ensure our samples are sizable and representative of contributors from all components.

roughly 3, from 9 to 16 to 28 contributors. This rate of growth is much faster than the theoretical growth rate of a clique in a scale-free network, which grows by $\log \log N$ with the size of the network [137]. We are just reporting this as a reference point, without making any claim that the collaboration graph is or should be modeled as a scale-free network.

The clique is stable over time. We observed that only 3 contributors in Firefox and 1 contributor in Eclipse were discarded from the clique (tier 1) that they were once a part of, i.e., existed in snapshot of 2006 but not anymore in the respective snapshots for years 2008 and 2010. This indicates the stability of clique and strengthens our claim that contributors in the clique serve all possible roles with high-confidence. If we found that contributors in the clique are unstable, it would have reduced the confidence-levels of our role-prediction accuracy.

Climbing up in the hierarchy is based on merit. We find that contributors who advance to an upper tier show a significant increase in their expertise profile metrics (e.g., number of bugs fixed, eLOC added, etc.) from the previous snapshot of the graph. This observation validates our claim that the tier a contributor belongs to is an indicator of her expertise level and that promotion from a lower-level tier to higher-level tier would require demonstration of significant contribution to the project. Additionally, this demonstrates that the promotion of a contributor in the expertise hierarchy is *merit-based*. In the future, we plan to study how and when this promotion or tier change occurs, and factors that determine the threshold of this promotion.

Expertise Breadth vs. Depth

We analyze how expertise breadth (i.e., familiarity with *multiple* components in a large project) is different from depth knowledge (i.e., familiarity with a *single* component in a large project). We hypothesize that contributors who gain familiarity with multiple components of the same project gain expertise quicker than those contributors familiar with a single component. To validate this hypothesis, we update each contributor's profile with a list of components they have worked on within Firefox and Eclipse. We found that contributors who form the clique have worked on at least 80.71% (for Firefox) and 69.88% (for Eclipse) of the total sub-components. Within tier 1 we see two different distributions for both the projects: (1) breadth-experts: contributors who have worked on at least 52.31% (for Firefox) and 44.55% (for Eclipse) of the components, and (2) depth-experts: the remaining contributors who have worked on only a single component. We found that people who are breadth-experts are senior contributors in the projects as opposed to depth-experts who are junior members of the project which indicates that when contributors join a project, they start gaining expertise in a single component; as their expertise grows over time, their familiarity (and therefore breadth-expertise) broadens.

Bug Tossing and Fixing

Bug tossing paths indicate how bugs were assigned from one contributor to another before a contributor could finally fix it. After we divide the contributors into various tiers, we investigate how bugs get tossed from one tier to another. We make three observations.

- *First assignment*: our investigation indicates that in Firefox, 57.02% of the bugs are

first assigned to a contributor from tier 2, while 24.81% of are assigned first to a contributor from tier 1. We find very similar results for Eclipse, where 60.17% of the bugs are first assigned to contributors in tier 2 and 18.53% bugs to tier 1.

- *Final fix*: 64.79% of the bugs in Firefox and 61.24% of the bugs in Eclipse are finally fixed by (or the last assignee in the bug tossing path) contributors from tier 1. 28.04% of bugs in Firefox and 30.93% bugs in Eclipse are fixed by contributors from tier 2.

- *Tossing patterns*: we wanted to see how bugs are tossed between tiers. We observe the following seven patterns in bug tossing among tiers:

    1. **Same** indicates that a bug assigned to a contributor in a tier was either fixed by him (i.e., no tosses) or fixed by a contributor from the same tier,
    2. **Up** indicates that a bug assigned at first to tier x is fixed by a contributor in tier y when that $x < y$,
    3. **Down** indicates that a bug assigned at first to tier x is fixed by a contributor in tier y, when $x > y$,
    4. **UpDown** indicates that a bug assigned at first to tier x is tossed to tier y and then eventually fixed by a contributor in tier x when $x < y$,
    5. **DownUp** indicates that a bug assigned at first to tier x is tossed to tier $x - 1$ and then eventually fixed by a contributor in tier x when $x > y$,
    6. **UpDownUp** indicates that a bug assigned at first to tier x, tossed to tier y, followed by tier z and then finally fixed by a contributor in tier w, where $x < y$, $z < y$, and $w < x < y$.

7. **DownUpDown** indicates that a bug assigned at first to tier x, tossed to tier y, followed by tier z and then finally fixed by a contributor in tier w, where $x > y$, $z > y$, $w < z$ and $w < x$.

To better illustrate the tossing pattern classification, we provide an example with four contributors, D_1 (tier 1), D_2 (tier 2), D_3 (tier 2), and D_4 (tier 3). In Table 6.4 we show various possible tossing paths and corresponding tossing pattern we classify the paths accordingly to. In Figure 6.8, we show the distribution of bugs in the seven categories of tossing patterns.

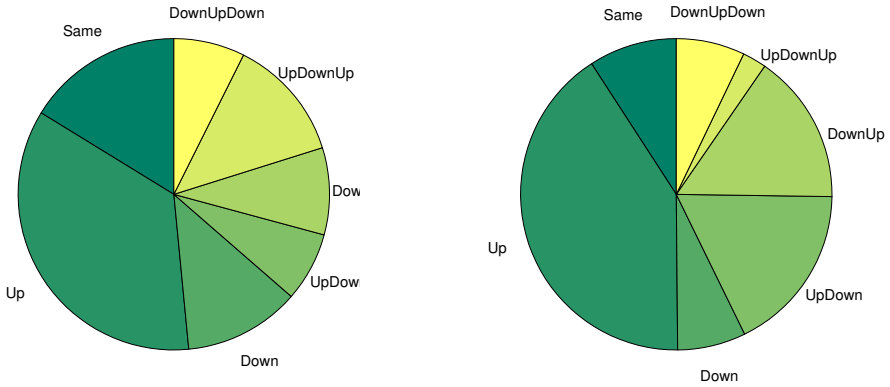
The tossing pattern distribution for the two projects is shown in Figure 6.8. The distribution indicates that in both Eclipse and Firefox, highest percentage of bugs are first assigned to lower tiers and then eventually fixed in an upper tier. The remaining tossing patterns have different distributions for the two projects implying that there is no generic motif.

6.5.1 Predicting Role Profiles Using HCM

The HCM encodes significant information concisely, and an indication of this is that it can be used to predict the role profile of a developer D. This ability to predict role profiles from the HCM is crucial, since, as we explained in Section 6.4, the HCM can be constructed even for projects where certain source code and bug information might be unavailable. Figure 6.9 shows an overview of the process used for constructing and validating our predictor model—note how constructing HCM requires only a subset of bug and source data. We now proceed to defining our model, then evaluating its accuracy, and finally

Tossing Path	Tossing Pattern
$D_2 \rightarrow D_3$	Same
$D_3 \rightarrow D_1$	Up
$D_1 \rightarrow D_2$	Down
$D_4 \rightarrow D_1 \rightarrow D_2$ $D_4 \rightarrow D_1 \rightarrow D_2 \rightarrow D_3$	UpDown
$D_1 \rightarrow D_4 \rightarrow D_2$	DownUp
$D_3 \rightarrow D_1 \rightarrow D_2 \rightarrow D_1$	UpDownUp
$D_1 \rightarrow D_2 \rightarrow D_1 \rightarrow D_4$	DownUpDown

Table 6.4: Example of tossing pattern classification based on tossing paths. D_1 (tier 1), D_2 (tier 2), D_3 (tier 2), and D_4 (tier 3).



(a) Firefox

(b) Eclipse

Figure 6.8: Distribution of tossing pattern classification based on tossing paths.

showing that predictors constructed using standard expertise metrics have poor prediction accuracy.

Defining the prediction model. We construct a role predictor based on the HCM, i.e., a function f that, given HCM data for developer D, outputs the role profile of D:

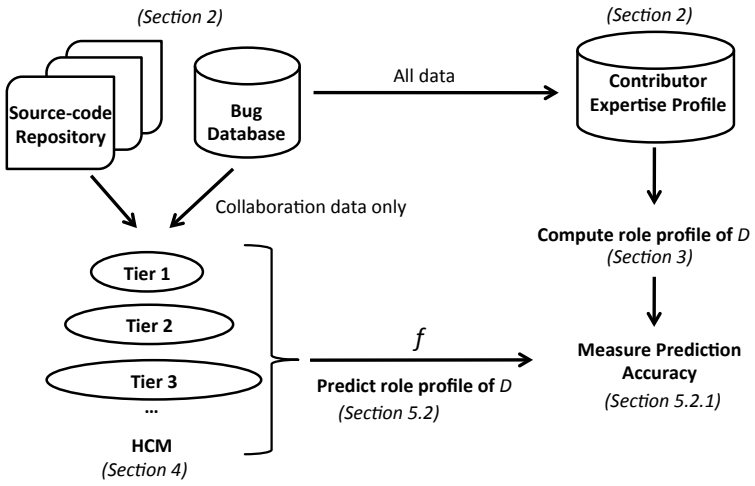


Figure 6.9: Measuring prediction accuracy.

$$< RoleProfile_D >= f(Tier_D, InDegree_D, OutDegree_D)$$

As defined in the previous section, we refine the initial graphs to be directed and thus to each node (developer) D, in addition to their level $Tier_D$ we associate in- and out-degrees, $InDegree_D$ and $OutDegree_D$. At a high level of abstraction, the lower the $Tier_D$ and the higher the $InDegree_D$ and $OutDegree_D$, the more likely it is that the D participates in a role. The function f is defined based on an 80 percentile "high" threshold for $InDegree_D$ $OutDegree_D$, as shown in the algorithms 1 and 2 .

We now evaluate the effectiveness of the HCM-based predictor model. As shown in Figure 6.9, we use the function f to predict the role profile of D, and then compare this predicted role profile with the role profiles we computed in Section 6.3 (i.e., the latter serves as reference output). To measure the effectiveness of our prediction, we focus on the precision of our role identification, i.e., how confident we are when we say that contributor

D has served a role R. Specifically, precision is the ratio of two numbers: the number of developers we correctly predict have served role R over the total number of contributors that we have identified to have role R. We report prediction accuracy in columns 2 and 3 of Table 6.5. We found that the highest prediction accuracy (75.98%) was achieved when predicting Assists in Firefox. The lowest prediction accuracy (47.22%) was attained when predicting Patch-quality improvers in Firefox.

Clustering contributors. We also investigated whether expertise metric values can be used to predict roles: can we form clusters based on expertise metric values that would correspond to roles? To answer this question, we first use the contributor expertise profiles (the tuples described in Section 6.2.2) as input to the EM clustering algorithm [41].[11]

After EM has determined clusters, we measured the fit between EM clusters and HCM-based roles as the ratio between the number of pairs of contributors D_1, D_2 who serve role R and are in the same cluster over the total number of pairs of contributors D_1, D_2 who serve role R. Put another way, this ratio tells us how many developers D with similar role R are within a cluster. For brevity we omit details, but we found the fit to be low (minimum 6.36%, median 15.62%, maximum 30.92%) for all roles in both Firefox and Eclipse. These findings suggest that standard expertise metrics do not make good role indicators.

Discussion. The main point of these comparisons is that determining roles using the initial graphs or the raw contributor activity data is not trivial. Therefore, the fact that HCM can provide more than 50% precision is a good indication that the model captures some interesting characteristics.

[11] We used the *Akaike Information Criterion* [4] to determine the optimal number of clusters, i.e., balance between a good fit and a small number of clusters, to avoid over-fitting.

Role	Prediction accuracy (%)	
	Eclipse	Firefox
Patch tester	69.62	66.28
Assist	67.73	75.98
Triager	59.06	60.37
Bug analyst	53.18	69.45
Core developer	70.96	62.89
Bug fixer	65.71	58.25
Patch-quality improver	61.80	47.22

Table 6.5: Role profile prediction accuracy using HCM.

6.6 Threats to Validity

We now present possible threats to the validity of this chapter's work.

External validity. Our expertise profiles and role definitions assume the existence of, and access to, the source code repository and a bug tracker (bug activity, bug report changes); this data might not be available in all projects, hence by selecting projects which have this information—Firefox and Eclipse—our study might be vulnerable to selection bias. Additionally, we have studied open source projects only, but commercial software might have different ways to quantify contributor expertise and roles.

Internal validity. Our bugfix-induced data relies on bug reports collected from Bugzilla at the time the chapter was written. Future changes in bug status (e.g., if closed bugs is re-opened) or bug severity might affect our results, but we cannot predict such changes.

Construct validity. Construct validity relies on the assumption that our metrics actually capture the intended characteristic, e.g., the expertise attributes we use accurately models an individual's expertise. We intentionally used multiple bug-fix induced and source-

code based metrics to reduce this threat. The roles we operationalize do not use cut-off points for frequency of contribution; therefore we do not to distinguish us between expert and non-expert contributors *within a specific role*. In the future, we intend to vary the threshold and evaluate its effect on our analysis.

Content validity. The assignee information in Bugzilla does not contain the domain info of the email address for a contributor. Therefore, we could not differentiate between users with the same email username but different domains (in our technique, bugzilla@alice.com and bugzilla@bob.com will be in the same bucket as bugzilla@standard8.plus.com). This might potentially lead to loss of prediction accuracy in our model. Similarly, while extracting contributor id's from log messages, we might miss contributors who submit patches via other committers because they do not have commit access. We do not know at this point how many such committers exist and how this affects our findings.

6.7 Contribution Summary

In summary, our main contributions are:

- We revisited previous expertise metrics and showed that each metric captures a local notion of expertise by quantifying a specific development activity (e.g., LOC added) but when put together, they fail to capture a global notion of expertise.

- We introduced a set of roles: Patch tester, Assist, Triager, Bug analyst, Core developer, Bug fixer, Patch-quality improver. We also provide ways to define these roles rigorously, assuming that we have access to the Role profile, which only some projects

maintain.

- We designed an intuitive graph-based model of developer contribution (Hierarchical Contributor Model (HCM)) that concisely represents the contributor interaction, in a way that captures hierarchy, role and "importance" of contributors, which is hard to do with previously defined expertise metrics. We then showed how one can benefit from our model: we use it as a framework for identifying interesting properties of the structure and the evolution of the contributor interactions, and show that it can even help us predict the roles of contributors.

6.8 Conclusions

In this chapter we studied two large projects, Firefox and Eclipse, to operationalize contributor role and expertise. We show that role and hierarchy information can capture a developer's profile and impact in ways current expertise metrics cannot. We also explored how a contributor's role, breadth and depth expertise evolve over time. We found that collaboration can be an effective predictor of individuals' roles; and that as contributors' expertise increases, they tend to serve multiple roles in the project. Furthermore, our analyses revealed how weighted collaboration graphs can be used to find sub-communities in a project where contributors of similar expertise work on the same parts of the code.

Algorithm 1 Definition of f for predicting bug-based role profile of contributor D

Input: $Tier_D$, $InDegree_D$, $OutDegree_D$

Output: $RoleProfile_D$

Description:

if $Tier_D = 1$ **then**

D served ALL Roles

else if $Tier_D$=2 **then**

if $InDegree_D \geq 80\%$ & $OutDegree_D \geq 80\%$ **then**

D served as an Assist and Triager

if $InDegree_D \geq 80\%$ & $OutDegree_D < 80\%$ **then**

D served as a Patch Tester

if $InDegree_D < 80\%$ & $OutDegree_D \geq 80\%$ **then**

D served as an Assist

if $InDegree_D < 80\%$ & $OutDegree_D < 80\%$ **then**

D served as a Bug analyst

else if $Tier_D \geq 3$ **then**

D served NO Roles

Algorithm 2 Definition of f for predicting source-based role profile of contributor D

Input: $Tier_D$, $InDegree_D$, $OutDegree_D$

Output: $RoleProfile_D$

Description:

if $T = 1$ **then**

D served ALL Roles

else if T=2 **then**

if $InDegree_D \geq 80\%$ & $OutDegree_D \geq 80\%$ **then**

D served as Core developer and Bug fixer

if $InDegree_D \geq 80\%$ & $OutDegree_D < 80\%$ **then**

D served as a Patch-quality improver

if $InDegree_D < 80\%$ & $OutDegree_D \geq 80\%$ **then**

D served as Bug fixer

if $InDegree_D < 80\%$ & $OutDegree_D < 80\%$ **then**

D served NO Roles

else if $Tier_D \geq 3$ **then**

D served NO Roles

Chapter 7

A Declarative Query Framework

In this chapter, we argue the need for effective search techniques on the integrated system we built by combining various software elements from multiple repositories. An automatic query system that can answer a broad range of queries regarding the project's evolution history would be beneficial to both software developers (for development and maintenance) and researchers (for empirical analyses). For example, the list of source code changes or the list of developers associated with a bug fix are frequent queries for both developers and researchers. Integrating and gathering this information is a tedious, cumbersome, error-prone process when done manually, especially for large projects. Previous approaches to this problem use frameworks that limit the user to a set of pre-defined query templates, or use query languages with limited power. Next, we argue the need for a framework built with recursively enumerable languages, that can answer temporal queries, and supports negation and recursion. As a first step toward such a framework, we present a Prolog-based system that we built, along with an evaluation of real-world integrated data

from the Firefox project. Our system allows for elegant and concise, yet powerful queries, and can be used by developers and researchers for frequent development and empirical analysis tasks.

7.1 Introduction

Developers are overwhelmed whenever they are faced with tasks such as program understanding or searching through the evolution data for a project. Examples of such frequent development tasks include understanding the control flow, finding dependencies among functions, finding modules that will be affected when a module is changed, etc. Similarly, during software maintenance, frequent tasks include keeping track of files that are being changed due to a bug-fix, finding which developer is suitable for fixing a bug (e.g., given that she has fixed similar bugs in the past or she has worked on the modules that the bug occurs in). In Section 2.3.1 we showed examples of how the underlying framework we built can be used for querying on integrated evolution data for large projects would be beneficial for research in empirical software engineering, where data from these repositories is frequently used for hypothesis testing. However using structured query languages similar to prior work do not allow efficient search and analysis on software evolution data. They have two main inconveniences: (1) they are not flexible enough, e.g., they permit a limited range of queries, or have fixed search templates; (2) they are not powerful enough, e.g., they do not allow recursive queries, or do not support negation; however, these features are essential for a wide range of search and analysis tasks. In this chapter, we show how we can

The work presented in this chapter have been published in the Proceedings of the Third International Workshop on Search-driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE 2011) [19].

address these shortcomings by using a Prolog-based integration and query framework. We chose Prolog because it is declarative yet powerful, which allows elegant, concise expression of queries for data collection and hypothesis testing. Our framework captures a wealth of historical software evolution data (information on bugs, developers, source code), and allows concise yet broad-range queries on this data. The three main novelties of our framework are: (1) it is temporally aware; all the tuples in our database have time information that allows comparison of evolution data (e.g., how has the cyclomatic complexity of a file changed over time?); (2) it supports powerful language features such as negation, recursion, and quantification; (3) it supports efficient integration of data from multiple repositories in the presence of incomplete or missing data using several heuristics.

The rest of the chapter is organized as follows: we describe the advantages of using a Prolog-based framework, the key novelties in our design, and our data model in Section 7.2. We demonstrate how our framework can elegantly express, and effectively answer, a broad range of queries, without requiring pre-defined templates, in Section 7.3; these queries form the kernel of a query library that can be used by developers and researchers in their activities.

We tested our framework on a large, real-world project with separate source code and bug repositories: a subset of Firefox[1] evolution data. From Firefox's source code repository we extracted change log histories to populate our source code database. We then extracted the bugs associated with these source files. Finally, we added function call edges (from the static call graph) to the database, for demonstrating how our framework

[1]Firefox (http://www.mozilla.com/firefox) is the second most widely-used web browser [49] and has been used in many empirical studies in software engineering [112, 16, 17].

is beneficial in impact analysis. In Section 7.4 we present preliminary results of evaluating our framework on Firefox data, in terms of result size and query speed.

7.2 Framework

We now turn to presenting our framework. We first motivate our decision for choosing Prolog as the storage and querying engine for our framework, then describe the key novel features in our approach, followed by the data model. We implemented our framework in DES, a free, open-source Prolog-based implementation of a basic deductive database system [143].

7.2.1 Why Use Prolog?

Prolog is declarative. In declarative languages, queries are concise and elegant because there is no need to specify control flow or pre-define query templates.

Prolog supports negation. Negation extends the range of expressible queries but is potentially expensive. For example, previous frameworks cannot answer queries like "return the list of developers who have *not* fixed bugs in module A" or "return the list of modules that are *not* affected when module A is changed"; such queries are useful, however, e.g., the second query can be used to reduce regression testing. Query *Q1* in Table 7.2 is an example of negation use in our framework.

Prolog supports recursion. Recursive queries are important, e.g., for computing the transitive closure required in impact analyses. Although certain versions of SQL support recursion, it is usually a limited form of recursion, and implemented via proprietary

Table	Table Name	Attributes
Source Basic	sourcebasic	FileNameAndPath, Release, List of Functions Defined, Complexity, Defect Density, Date
Source Change	sourcechange	FileNameAndPath, Date, RevisionID, BugID, DeveloperID, Days, Lines Added
Source Depend	sourcedepend	FileNameAndPath, List of Files Depends it on (w.r.t. the static call graph), Date
Bugs	bugs	Bug ID, Date Reported, Developer ID, Date Changed, Developer Role, Severity, Bug Status, Bug Resolution, List of Dependencies, DaysReported, DaysFixed

Table 7.1: Database schema.

extensions. *Q2* in Table 7.2 is a sample query that requires recursion.

7.2.2 Key Features

We now showcase some key features of our framework; existing approaches fail to support one or more of these features.

Temporal Queries

Previous approaches that build databases from integrating multiple software repositories are not capable of answering temporal queries. For example, the following queries cannot be answered by existing systems: (1) Who modified file A on a given day?, (2) Whom was the bug B assigned to during a certain period?, (3) What changes were made to a file F during a specific period of time?, (4) How have source code metrics (e.g., complexity, defect density) of a file changed over time?

Natural Language Query	DES Clause
Q1: Return the list of bugs fixed by developer D which do **not** depend on other bugs	bugs_not_depend(B,D,R) :– bugs(B,_,D,_,_,_,_,_,_,R), **not**(R='null').
Q2: Given two functions F1 and F2, check if a change to F2 will affect F1	reach(X,Y) :– sourcedepend(X,Y). reach(X,Y) :– reach(X,Z), sourcedepend(Z,Y).
Q3: Return all activities (fixes F or source code changes C) associated with developer D	activity (B,D,F) :– sourcechange(F,D,B,_,_,_,_). activity (B,D,F) :– bugs(F,_,D,B,_,_,_,_,_,_,_,_).
Q4: Return all bugs fixed by developer D	bugs_fixed (B,D,R) :– bugs(B,_,D,'Fixed',_,_,_,_,_,_,_,_).
Q5: Return the bugs developer D could **not** fix	bugs_not_fixed (B,D) :– bugs(B,_,D,'Assigned',_,_,_,_,_,_,_,_).
Q6: Return the list of bugs developer D reported and was eventually fixed by E	bugs_fixed_D_E(B,D,E) :– bugs(B,_,D,'Reported',_,_,_,_,_,_,_,_), bugs(B,_,E,'Fixed',_,_,_,_,_,_,_,_).
Q7: Return the list of files modified by developer D on date DT	source_modified_bydate (F,D,R,DT) :– sourcechange(F,D,_,R,DT,_,_).
Q9: Return files modified by developer D for which more than 10 lines were added	source_modified_bylines (F,D,B,R,L) :– sourcechange(F,D,B,R,_,_,L), L>10.
Q10: Return all source file changes	all_src_changes (B,R,F,DT,D) :– sourcechange(F,D,B,R,DT,_,_).
Q8: Return the list of bugs reported and fixed by the same developer D	bugs_fixed_D_D(B,D) :– bugs(B,_,D,'Reported',_,_,_,_,_,_,_,_), bugs(B,_,D,'Fixed',_,_,_,_,_,_,_,_).
Q9: Return the tossing history of bug B	bugs_toss(B,D,R) :– bugs(B,_,D,R,_,_,_,_,_,_,_,_).
Q10: Return the source files that have been modified by two developers D and E	common_modified(D,E,R) :– sourcechange(R,D,_,_,_,_,_), sourcechange(R,E,_,_,_,_,_).
Q11: Return the list of bugs fixed between dates D1 and D2	bugs_fixed_bydate (B,D,DT) :– bugs(B,_,D,'Fixed',_,_,_,_,_,_,DT,_), DT<D2, DT>D1.
Q12: Return the list of source files modified by developer D before date D1	source_modified_bydate (F,D,R,DT,DY) :– sourcechange(F,D,_,R,DT,DY,_), DY<D1, DY>0.
Q13: Return the list of open (unresolved) bugs	bugs_new(B,D) :– bugs(B,_,D,_,_,_,_,_,_,_,_,–1).

Table 7.2: Sample queries from our library.

Query		Resulting tuples	Time (ms)
Q1	bugs_not_depend(B,wtc,R)	218	1,746
Q2	reach('main; nsinstall .c', 'PK11_FreeSlot;pk11slot.c')	1	4
	reach('PK11_FreeSlot;pk11slot.c', 'main; nsinstall .c')	0	5
Q3	activity (B,wtc,F)	2,569	4,489
Q4	bugs_fixed (B,wtc)	218	143
Q5	bugs_not_fixed (B,wtc)	558	287
Q6	bugs_fixed_D_E(B, fabientassin ,wtc)	1	127
Q7	source_modified_bydate (F, nelson , R,'2001/01/07')	46	197
Q8	bugs_fixed_D_D(B,nelson)	126	25
Q9	bugs_toss(236613,D,R)	18	143
Q10	common_modified(nelson,wtc,R)	465	25,120
Q11	bugs_fixed_bydate (B,D,DT), 2008/7/23<DT< 2008/10/23 .	47	1,769
Q12	source_modified_bydate (F, nelson , R,DT,DY), DT=2008/7/23.	1,275	1,282
Q13	bugs_new(B,D)	810	2,435

Table 7.3: Example queries for query declarations in Table 7.2.

Recursion

Transitive closure is helpful for impact analysis, e.g., "return the set of files that will be affected by modifications to file F." The problem with prior approaches is that they either cannot compute transitive closure, or can only compute it when the graph (where edges indicate a "depends" relationship) is known statically. For example, we might want to find all the descendants of a file F after it has been refactored. If we do not know the definition of "depends", i.e., in this case, `is-descendant-of`, at the time we construct the database, we first need to write a query that generates the graph, and then transitively close it, using a language powerful enough to express transitive closure. Similarly, suppose we have a bug B_1 in file F, and we want to find the list of subsequent bugs in F that might have been introduced in the process of fixing B_1. The problem is, the list of subsequent bugs is constructed dynamically, e.g., all the bugs in F minus the list of bugs in F that depend on other bugs in other files. Previous approaches such as Codebook [11] use pre-computed transitive closure for efficiently answering a pre-defined set of queries, e.g., "the set of all functions F depends on"; however, queries like "list all functions that both F_1 and F_2 depends on" cannot be answered because they require language support for recursion/transitive closure. Moreover, when data from new releases is added to the database, pre-computed transitive closure does not work, because the "depends" relationships might have changed due to the new data, hence a dynamic transitive closure algorithm would be required.

Integration

In open source projects, it is often difficult to integrate related information because it is spatially dispersed and incomplete. For example, often bug reports do not have complete information about files that were changed during a bug fix. Consider Mozilla bug 334314; according to the Bugzilla bug report, three changes were made to file `ssltap.c` to fix this bug—once by developer ID *alexei.volkov.bugs* and twice by developer ID *nelson*. The information in the patch reference for this change is incomplete; [2] it is not clear who-has-made-which-change. However, from the change log of file `ssltap.c`, we can retrieve developers, changes, and change timestamps, which helps us complete the bug database.

7.2.3 Storage

Our framework is designed to integrate information from three sources: (1) source code repositories—size, location, source code dependencies from the static function call graph, etc., (2) bug repositories—who reported the bug, what is the present status of the bug, bug dependency data, etc., and (3) interaction between developers—who tossed bugs to whom, which two developers worked on same files, etc. Note how function calls, bugs and developer interactions induce dependency graphs. We integrate information from these three sources and store it into a database, so that our framework can answer cross-source queries, as demonstrated in Section 7.3. The schema for our database is presented in Table 7.1. We now proceed to describing the database schema, contents, and updates.

Source code. The source code data is stored in three tables: basic source code

[2] Patch for bug 334314:
https://bug334314.bugzilla.mozilla.org/attachment.cgi?id=218642

information, source code changes and source code dependencies. The *basic source code information* table (sourcebasic) stores, for each module (file): its location, the list of functions it defines, complexity metrics, defect density information, and a corresponding date. Note that a file can have multiple entries in the database due to multiple releases, hence when a file is not changed in a release, all values but the release timestamp remain unchanged. These entries are important for tracking changes between releases. In the *source change* table (sourcechange), we store details of all revisions that have been made to a file, either as feature enhancements or bug fixes: the date the change was made, the revision ID, the bug ID (if the change was due to a bug fix) and the developer who committed it, and number of lines added. For a source change entry in the database, we also store the number of days since the first commit[3] the current activity took place.[4] In the *source dependency* table (sourcedepend), we store information about which other entities a given module or function depends on directly, i.e., file, module or function dependencies induced by the call graph.

Bugs. The bug table (bugs in Table 7.1) stores information related to a bug: the date on which the bug was reported, list of developers associated with the bug and their roles (i.e., who reported it, who the bug was assigned to at some point, who fixed it), the severity of the bug, the present status of the bug, final resolution of bug and list of bugs this bug depends on. To answer queries about a time interval (e.g., how many bugs were fixed between July 2008 and May 2010), we add two attributes —DaysReported and DaysFixed—that represent the number of days since the first release of the project that the bug was reported and fixed respectively. If a bug has not been resolved at the time of database creation,

[3]The first commit found in the log files we used was on 07/23/1998.
[4]This is done to answer queries involving time intervals.

DaysFixed is set to -1.

Developer information. Thanks to our source and bug table schema design choice, having a developer database is redundant. All the information for developers (e.g., tossing information, bug fix information, code authorship information) can be extracted from the source code and bug tables.

Updating the database. As software evolves, our database needs to grow; note that the database is monotonically increasing (we never retract facts).

7.3 Examples

We now proceed to presenting use cases for our system—a variety of frequent queries that arise in software development and empirical research. In Table 7.2 we demonstrate how using Prolog improves expressiveness and allows arbitrary information retrieval, without the need for pre-computation or templates. We envision these queries forming the kernel of a query library that can be used by developers in their daily development and maintenance activities; similarly, the library can be useful to researchers for empirical analysis and hypothesis testing. Note that, since our query language is based on Prolog, we support existential queries directly (variables in Prolog clause heads are existentially quantified), and universal queries by rewriting, i.e., $\forall x\, Q(x) \Leftrightarrow \neg\exists x\, \neg Q(x)$.

7.4 Results

We randomly selected 2128 C files and 58 C++ files from the Firefox source code repository and extracted their complete change log histories to populate our source change database. We extracted the 932 bugs associated with these source files. We also added to our source dependency table the 50 function call edges induced by the static call graph between functions in these files. In total, our database contained 63,142 tuples. In Table 7.3 we present the queries we used to test the query definitions showed in Table 7.2. The first column shows the query invocation, the second column shows the number of resulting tuples, [5] and the third column shows the query execution time, in milliseconds. We found that the time taken to answer a query using DES increases with the increase in number of resulting tuples, hence it can be quite high for queries with large results, e.g., *Q10*; we plan to address scalability in future work.

7.5 Conclusion

In this chapter we show how using a Prolog-based framework we can answer a broad range of queries on software evolution data that cross multiple software repositories. We used several examples on Firefox source and bug repositories to show how our framework is efficient in querying large, real-world evolution data. In the future, we would like to improve the scalability of our framework, increase its precision, and add a visualization component.

[5] In query *Q2* in Table 7.3, $Func; Mod$ represents function $Func$ defined in module Mod; the resulting tuple 1 denotes there is a path from $F_1; M_1$ to $F_2; M_2$ while 0 denotes otherwise.

Chapter 8

Related Work

Mining software engineering data has emerged as a successful research direction over the past decade [65]. This chapter surveys the related works in light of mining software repositories to benefit various decision-making processes in software development that have been analyzed in this dissertation.

8.1 Automating Bug Assignment

8.1.1 Machine Learning and Information Retrieval Techniques

Cubranic et al. [39] were the first to propose the idea of using text classification methods (similar to methods used in machine learning) to semi-automate the process of bug assignment. They used keywords extracted from the title and description of the bug report, as well as developer ID's as attributes, and trained a Naïve Bayes classifier. When presented with new bug reports, the classifier suggests one or more potential developers for fixing the

bug. Their method used bug reports for Eclipse from January 1, 2002 to September 1, 2002 for training, and reported a prediction accuracy of up to 30%. While we use classification as a part of our approach, in addition, we employ incremental learning and tossing graphs to reach higher accuracy. Moreover, our data sets are much larger, covering the entire lifespan of both Mozilla (from May 1998 to March 2010) and Eclipse (from October 2001 to March 2010).

Anvik et al. [8] improved the machine learning approach proposed by Cubranic et al. by using filters when collecting training data: (1) filtering out bug reports labeled "invalid," "wontfix," or "worksforme," (2) removing developers who no longer work on the project or do not contribute significantly, and (3) filtering developers who fixed less than 9 bugs. They used three classifiers, SVM, Naïve Bayes and C4.5. They observed that SVM (Support Vector Machines) performs better than the other two classifiers and reported prediction accuracy of up to 64%. Our ranking function (as described in Section 3.3) obviates the need to filter bugs. Similar to Anvik et al., we found that filtering bugs which are not "fixed" but "verified" or "resolved" leads to higher accuracy. They report that their initial investigation in incremental learning did not have a favorable outcome, whereas incremental learning helps in our approach; in Section 3.4 we explain the discrepancy between their findings and ours.

Anvik's dissertation [9] presented seminal work in building recommendation systems for automating the bug assignment process using machine learning algorithms. His work differentiated between two kinds of triage decisions: (1) *repository-oriented decisions* (determining whether a bug report is meaningful, such as if the report is a duplicate or

is not reproducible), and (2) *development-oriented decisions* (finding out whether the product/component of a bug report determines the developer the report is assigned to). They used a wide range of machine learning algorithms (supervised classification: Naïve Bayes, SVM, C4.5, Conjunctive Rules, and Nearest Neighbor and unsupervised classification: Expectation Maximization) for evaluating the proposed model and suggested how a subset of the bug reports chosen randomly or user-selected threshold could be used for classifier training. Similar to Anvik, we show how using four supervised classifiers (Naïve Bayes, Bayesian Networks, SVM, and C4.5) and a subset of training data can be used to improve bug assignment accuracy. In addition to classification, we also use a ranking function based on bug tossing graphs for developer recommendation and perform an ablative analysis to determine the significance of the attributes in the ranking function; Anvik's dissertation neither employ bug tossing graphs nor performs any ablative analysis. Anvik proposed three types of subset training data selection: random (100 bug reports where chosen in each iteration until desired prediction accuracy was achieved), strict (number of bug reports for each developer where determined depending on his lifetime contribution) and tolerant (number of bug reports were chosen randomly and was proportional to a developer's contribution); in contrast, we used a chronologically-backtracking method to find out the subset of bug reports that can be used to efficiently predict bug triagers instead of random selection. For evaluating their framework, they used bug reports from 5 projects: Firefox, Eclipse, gcc, Mylyn, Bugzilla. Their prediction accuracy is as follows: 75% for Firefox (by using 6,356 bug reports for training and 152 bug reports for validation) and 70% for Eclipse (by using 3,338 bug reports for training and 64 bug reports for validation). Our work differs

significantly from theirs in two ways: first, we use a different data set for our training and validation and we use all Mozilla products instead of Firefox alone, and second, we propose incremental machine learning based and probabilistic graph-based approach for bug assignment. By using all products in Mozilla and Eclipse, we can prune developer expertise further by our ranking function which leads to higher prediction accuracy.

Canfora et al. used probabilistic text similarity [36] and indexing developers/modules changed due to bug fixes [35] to automate bug assignment. When using information retrieval based bug assignment, they report up to 50% Top 1 recall accuracy and when indexing source file changes with developers they achieve 30%–50% Top 1 recall for KDE and 10%–20% Top 1 recall for Mozilla.

Podgurski et al. [133] also used machine learning techniques to classify bug reports but their study was not targeted at bug assignment; rather, their study focused on classifying and prioritizing various kinds of software faults.

Lin et al. [92] conducted machine learning-based bug assignment on a proprietary project, SoftPM. Their experiments were based on 2,576 bug reports. They report 77.64% average prediction accuracy when considering module ID (the module a bug belongs to) as an attribute for training the classifier; the accuracy drops to 63% when module ID is not used. Their finding is similar to our observation that using product-component information for classifier training improves prediction accuracy.

Lucca et al. [96] used information retrieval approaches to classify maintenance requests via classifiers. However, the end goal of their approach is bug classification, not bug assignment. They achieved up to 84% classification accuracy by using both split-sample

and cross-sample validation techniques.

Matter et al. [102] model a developer's expertise using the vocabulary found in the developer's source code. They recommend potential developers by extracting information from new bug reports and looking it up in the vocabulary. Their approach was tested on 130,769 Eclipse bug reports and reported prediction accuracies of 33.6% for top 1 developers and 71% for top 10 developers.

8.1.2 Incremental Learning

Bettenburg et al. [12] demonstrate that duplicate bug reports are useful in increasing the prediction accuracy of classifiers by including them in the training set for the classifier along with the master reports of those duplicate bugs. They use folding to constantly increase the training data set during classification, and show how this incremental approach achieves prediction accuracies of up to 56%; they do not need tossing graphs, because reducing tossing path lengths is not one of their goals. We use the same general approach for the classification part, though we improve it by using more attributes in the training data set; in addition, we evaluate the accuracy of multiple text classifiers; and we achieve higher prediction accuracies.

8.1.3 Tossing Graphs

Jeong et al. [76] introduced the idea of using bug tossing graphs to predict a set of suitable developers for fixing a bug. They used classifiers and tossing graphs (Markov-model based) to recommend potential developers. We use fine-grained, intra-fold updates

and extra attributes for classification; our tossing graphs are similar to theirs, but we use additional attributes on edges and nodes as explained in Section 3.3. The set of attributes we use help improve prediction accuracy and further reduce tossing lengths, as described in Sections 3.4.2 and 3.4.3. We also perform an ablative analysis to demonstrate the importance of using additional attributes in tossing graphs and tossee ranking.

8.2 Effects of Programming Language on Software Development and Maintenance

8.2.1 Influence of Programming Languages on Software Quality

Myrtveit et al. [119] performed an empirical study to understand if usage of C++ as the primary programming language increased software development productivity when compared to projects written in C. They used projects written either in C or C++ and no description of the projects have been provided. They compute effort as the number of hours a developer worked on a project and found that language choice has no effect on software development. Phipps [130] conducted a study to compare the effects of programming language on defect density and productivity rates. Two different small projects (one in Java and the other in C++) developed by the author himself were considered. This study found that when defects were measured against development time, Java and C++ showed no difference. Although the author acknowledges his efficiency in C++ over Java, the study revealed that using Java he was twice as productive as when using C++. Myrtveit et al. [119] performed an empirical study to test if using C++ (as the primary programming language) increased

developer productivity when compared to using C. They used projects written either in C or C++ (no description of the projects were provided), computed effort as the number of hours a developer worked on a project and found that language choice has no effect on productivity. Phipps [130] conducted a study using two different small projects (one in Java and the other in C++, developed by the author himself) to compare the effects of programming language on defect density and developer productivity. This study found that defect density was unaffected by the programming language and using Java the author was twice as productive as when using C++ (even though he is more experienced in C++ than Java). In contrast to these studies, our methodology differs in three ways: (1) we control for development process and developer competence by considering projects written in combination of C and C++, (2) we used real-world projects which have large developer and user bases enabling us to draw meaningful conclusions, and, (3) we use a different set of metrics which are widely used when evaluating software quality. Paulk [129], Jones et al. [77], and Lipow et al. [93] studied factors that affect software quality; they infer that there is *no correlation* between software quality and the programming language used in building software; we now discuss how our study differs, and why our conclusions are different from theirs. Jones et al. [77] used a functionality-based size measure (the eLOC required to implement a function point) and concluded that the only factor that affects software quality is the number of function points in a program. We choose interface complexity as one of the metrics for internal code quality, e.g., if file A has more function calls with more parameters than file B, A's interface complexity is higher than B's. Thus, similar to Jones et al., our metric effectively relates functions to code complexity and software quality. Lipow et al. [93] found

that program size affects software quality, but code quality is unaffected by the choice of the programming language. The authors studied applications written in different languages but implementing the same functionality; they do not control for programmer expertise. Jones et al. and Lipow et al. do not provide a measure of goodness of fit for their analyses. Paulk [129] compared small applications written in different languages by graduate and upper undergraduate students and found that, in these applications, software quality was more dependent on programmer abilities than on the programming language. Their conclusion strengthens our case, i.e., the need to control for programmer competence. In contrast to all these studies, our study examines real-world, large applications, written and maintained by seasoned developers competent in both languages.

Fateman [47] discusses the advantages of Lisp over C and how C itself contributes to the "pervasiveness and subtlety of programming flaws." The author categorizes flaws into various kinds (logical, interface and maintainability) and discusses how the very design of C, e.g., the presence of pointers and weak typing, makes C programs more prone to flaws. Using Lisp obviates such errors, though there exists a (much smaller) class of bugs specific to Lisp programs. The author concludes that Lisp is still preferable to C. We consider C and C++ to study the difference between a lower-level, and (comparatively) higher-level, language. Our goal was not to identify those C features that are more error prone and how C++ helps avoid such errors. Rather, our analysis is at a higher level, i.e., we analyze which language (C or C++) helps produce code that is less complex, less buggy and requires less effort to maintain.

Holtz et al. [69] compare four languages (C, Pascal, Fortran 77, and Modula-2)

to identify how language syntax and semantics affect software quality. They compare how easy it is to understand a program written in different languages, and how this facilitates development and maintenance. For example, various control constructs (e.g., recursion, `while` loops, etc.) offered by different programming languages can increase or decrease code size, understandability and code complexity. In contrast, we study applications as a whole, rather than with respect to language constructs. Burgess et al. [34] and Wichmann et al. [169] examine how the choice of programming language may affect software quality, by focusing on programming language constructs, similar to Holtz et al. However, the authors do not perform any empirical study to differentiate between languages, and do not provide any statistical results.

Hongyu et al. [71] compared code complexity with software quality to test the influence of the language used, but their study is limited to applications written in a single language (C, C++, or Java) by different teams of students and conclude that software quality depends on developer expertise only. In contrast, our study looks at complexity and quality in mixed C/C++ applications where the same developers contribute to both C and C++ code bases, hence developer expertise is kept constant while varying the language.

Mockus et al. [112] computed defect density (in a manner similar to ours) for Mozilla Firefox, for the period 1998–2000, while our defect density values cover 10 additional years, i.e., the period 1998–2010. Many researchers [48, 83, 63, 173, 116] have proposed effort estimation models in open source software. Our study is not centered on estimating effort or developing effort models; rather, we use effort as a metric to measure software maintainability.

8.2.2 Measuring software quality and maintenance effort

Mockus et al. [112] computed defect density (in a manner similar to ours) for Apache and Mozilla Firefox. Their defect analysis for Firefox covers the period 1998–2000. Our study covers all major and minor releases from 1998 to 2010. Kim et al. [80] and Graves et al. [59] computed defect density similar to ours in their study of estimating fault prediction and bug identification. Many researchers [48, 83, 63, 173, 116] have proposed effort estimation models in open source software (OSS). Our study is not centered on estimating effort or developing effort models; rather, we use effort as a metric to measure software maintainability. Our estimation model is similar to the one used by Parastoo et al. [115], e.g., code churn over total eLOC.

8.3 A Graph-based Characterization of Software Changes

8.3.1 Software Network Structural Properties

Louridas et al. [94] studied 19 small projects (3kLOC–13kLOC) written in one of several language: C, C++, Perl, Ruby, or Java. They build module dependency graphs by choosing modules of varying sizes and functionality, though no information about how these modules were chosen is provided. They concluded that module dependency graphs exhibit scale-free properties and follow power laws. We look at larger applications, include all functions and modules in our graphs, study application evolution, and construct predictors. We look at software of eLOC size ranging between 6K–3780K and instead of randomly choosing modules, we build function call graphs and module collaboration graphs on the

entire software. Additionally, we look at entire lifespans of the projects rather than a single version and propose graph-based metrics which can be used as software quality predictors.

Myers [118] examined class collaboration graphs for six open source projects written in C or C++: VTK, Digital Material, a simulation library, AbiWord, Linux kernel 2.4.19, MySQL 3.23.32, XMMS 1.2.7. Myers used various metrics (degree distribution, degree correlation, and clustering) to study the nature of class collaboration graphs. Their study revealed that these graphs are scale-free and small-world networks. We use a different set of graph metrics, study application evolution, and construct predictors.

Valverde et al. [159] studied single releases of 29 applications written in C and C++, constructed class collaboration graphs (but using information from the header files only) and report four main findings: (1) there is a linear relation between number of links and nodes in each of these graphs, (2) the projects demonstrate small-world networks because they have small graph diameters, (3) the application components are highly clustered, and (4) software cost scales with its size. The applications we used were much larger, and the focus of our study was not only graph topology, but also evolution and prediction.

Valverde et al. [158] studied the occurrence of motifs in software graphs. They examined 83 systems (details on these systems are not provided) and found that certain kind of motifs occur more frequently across all these systems. We did not study motifs, though our cycle detection identifies circular motifs of any size.

Solé et al. [150] examined class collaboration graphs in two Java projects and reported scale-free degree distributions. They used undirected graphs for their study and reported a strong correlation between degree of the node and its dependency. They sug-

gested a software cost model such that minimization of software development costs is an optimal trade-off between large, expensive components with few interconnections and small, inexpensive components with large interconnections.

Ma et al. [98] studied 6 projects written in C, C++ or Java, and found that the stability of a motif formed in the module collaboration graph shows positive correlation with its frequency and a motif with high Z-score tends to have stable structure. Ma et al. [97] studied the structural complexity of a program and how it correlates with stability. They studied single release of 11 applications written in C, C++, or a combination of both. They found that with an increase in *evaluation coefficient R* (which measures cumulatively the effects of ripple degree, connectivity and abstraction) the randomness of other factors (ripple degree, connectivity and abstraction) increases and results in increased structural complexity.

Potanin et al. [134] examined the structure of dynamic class collaboration graphs by analyzing run-time analogs of static class collaboration graphs. They observed power-law in-degree and out-degree distributions. However, they do not report any difference in observations between the static and dynamic graphs. Wheeldon et al. [168] identified power-law relationships in inheritance and aggregation graphs by analyzing three Java projects: Java Developer Kit (JDK), Apache Ant and Tomcat. Valverde et al. [157] studied class collaboration graph for the 1.2 version of Java Development Framework and reported that class collaboration graphs similar to biological networks exhibit redundancy but does not exhibit any form of degeneracy.

Vasa et al. [160] studied the type dependency graphs of 12 open source Java

projects and their evolution over one year. They used three metrics: fan-in, fan-out and branch count (cyclomatic complexity). Fan-in and fan-out are synonymous to in- and out-degree of nodes in collaboration graphs. They found that very little existing code changes with evolution (as software evolves, it develops a certain degree of stability) and that Lehman's law of increasing complexity is not applicable when measured for individual classes.

Louiridas et al. [95] studied the collaboration graphs of twenty datasets, which includes eight libraries, one package, one project for which only the system calls and library functions are considered. The largest project they consider is of size 13,055 eLOC. The have three main observations: (1)a core group of people contribute to most of the project and that part of the project forms the majority of the code, (2) nodes which have high in-degree are the most re-usable components in the project, and (3) the collaboration graphs exhibit scale-free nature.

Wang et al. [163] studied the evolution of the Linux kernel using complex networks analysis; their study is based on 223 Linux kernel versions. They used node degree distribution and average path length of the call graphs as metrics and found that the call graphs of the file system and drivers module are scale-free small-world complex networks and that both of the file system and drivers module exhibit very strong preferential attachment tendency.

Our study is different from all the above works in three significant ways: (1) we analyze a broad range of large projects written in C, C++, or a combination of both, and additionally study multiple releases of the same project which allows us to analyze

the evolution in the topologies of these graphs, (2) we propose graph-based metrics which can be used as software quality predictors, and (3) we also look at developer-collaboration graphs in two large, widely-used open source projects which reveals how social networking among developers affect software quality.

8.3.2 Software Networks for Failure Prediction

Zimmermann et al. [175] construct source code dependency graphs in Windows Server 2003. They used the complexity of these dependency graphs (measured as cyclomatic complexity, degree-based complexity, distance-based complexity and multiplicity-based complexity) to predict the failure-proneness of a given source code artifact.

Schroter et al. [144] performed an empirical study of 52 Eclipse plug-ins and reported that failure history of software artifacts can be used to build models which accurately predict failure-prone components in newer versions of the same program. Their model is based on the USES relationships between software components and is capable of answering two questions about a software project: (1) whether a component will be failure-prone or not, based on its design data and (2) which are the most failure-prone components.

Nagappan et al. [122] showed that dependency graphs built from software component dependencies can be used as efficient indicators of post-release failures. They evaluated their approach on Windows Server 2003; their model could appropriately identify 60% of important components that developers considered critical, too.

Our study is different from all the above works in two significant ways: (1) we analyze multiple releases of the same project which allows us to analyze the evolution

in the topologies of these graphs, (2) we propose NodeRank, a metric which is powerful in identifying critical spots in the software. Additionally, our *ModularityRatio* metric is capable of predicting maintenance effort.

8.3.3 Bug Severity Prediction

Menzies et al. [109] proposed a text classification-based framework SEVERIS (SEVERity ISsue assessment) to assist test engineers in assigning severity levels to defect reports. They tested their approach on bug reports in NASA's Project and Issue Tracking System (PITS) and reported up to 90% prediction accuracy. Lamkanfi et al. [88] also proposed a text classification based machine learning model to predict the severity of a bug. They tested their model on Mozilla, Eclipse and GNOME bug reports, and achieved 65–85% accuracy. In contrast, our *NodeRank* works at both function and module level and can predict bug severity before a bug report is filed.

8.3.4 Developer Collaboration

Bird et al. [21] studied coordination among developers by analyzing the Apache HTTP Server Developer mailing list for a period of 7 years. They found that developer graphs are small-world networks, and that there is a strong relationship between the level of email activity and the level of source code activity for a developer. Pinzger et al. [131] built a contribution network that represented developer contributions with a heterogeneous developer-module network in Microsoft Windows Vista. They found that in this network, central software modules are more likely to be failure-prone than modules located in sur-

rounding areas of the network. Their analysis also shows that the number of developers and number of commits are significant predictors for the probability of post-release failures. Abreu et al. [3] studied developer communication frequency for the Eclipse JDT project and found that the frequency of developer communication is positively correlated with the number of bugs in the project. They also found that developer activity increases pre-release of a version.

Our work differs in two ways from these prior efforts: (1) we look at multiple versions of developer collaboration by constructing these graphs for each year and analyzing how they change over time, and (2) we show that there exists a high positive correlation between edit distance between these successive developer graphs and the defect count.

8.4 Quantifying Contributor Expertise and Roles

8.4.1 Contributor Roles

Yu et al. [174] define two types of project *membership*: core members, i.e., developers who have interacted with each other (committed to the same file) at least 10 times and associate members (developers who have interacted with core members 5-10 times and 1-5 times amongst themselves). They evaluated their approach on ORAC-DR (14 members) and Mediawiki (56 developers) and reported the distribution of core and associate members in these projects. Alonso et al. [7] use visualization to quantify developer expertise (based on number of files the developer is associated with) and differentiate between developers and contributors. They used a supervised classifier to learn how contributors are associated

with various files and then differentiate between contributors and developers in the Apache project (75 developers, 8 years' worth of CVS logs). They also extract different components a developer has contributed to in the source code, and form a word cloud-based visualization to predict a developer's expertise. If a developer has many keywords but not a single one with high frequency, they term him as a "generalist" while if a contributor for example has "security" as the highest frequency word in their word cloud, they quantify him as "security expert." Our work defines and predicts seven finer-grained roles stable across projects, uses a wide range of expertise metrics, introduce a hierarchy model and is based on larger data sets.

8.4.2 Developer Expertise

Mockus et al. [113] studied developer expertise evolution by using source code data from a commercial product at Bell Labs. Based on interviews with 19 professional Java programmers, Fritz et al. [51] report that, according to the programmers, expertise is concentrated on source code which they author, the code parts they use and the frequency of that use. Fritz et al. [52] argued that in addition to authorship, developer's interaction with the code also sheds light on their expertise. They compute a degree-of-knowledge (DOK) model that captures in addition to how many files a developer has authored, how many files she has changed in his lifetime and find that a developer changes a file authored by her more frequently compared to files authored by others, Schuler et al. [145] introduced the concept of "usage expertise," defined as the knowledge of methods that a developer's code calls. They argued that the more a developer reuses existing code, the more knowledge she

has about existing code. They used the Eclipse CVS logs to mine the information about how contributors reuse code and conclude about their expertise. They found that when developers reuse similar parts of the code, they are more likely to create small neighborhoods in the contributor collaboration graphs. Minto et al. [111] built a tool—Emergent Expertise Locator (EEL) that can recommend expert developers in emerging teams. They use a recommendation algorithm that returns a ranked list of developers for a given file. They used their tool to study three open source projects: Firefox, Bugzilla and Eclipse. Gousios et al. [58] presented an approach for evaluating developer contributions based on data from bug repositories. They define a *developer contribution metric* which measures how many LOC a contributor worked on, and how many events or activities he/she has been associated with it. Using this metric for each developer, their model ranks the importance of events associated with a project. They tested their approach on the Alitheia project and showed how the model could classify important events in the project. Dominique et al. [103] built a developer expertise model based on their vocabulary or set of words that appear in bug reports they fix and classify bug reports based on that for triaging purposes and tested their approach on Eclipse. Bird et al. [22] studied the effects of code ownership in Windows software, and found that code ownership is an effective indicator of developer's knowledge. Similar to us, they found that minor contributors often contribute buggy code (in contrast to major developers or module owners). However, they do not quantify contributor role or expertise using a wide-range of expertise attributes or use contributor hierarchy as a proxy for developer expertise. Rahman et al. [136] studied effects of ownership and experience on software quality. They categorized developers into generalized experts and specialists based

on the components they have committed to in a project while fixing a bug. They found that specialized experience leads to less defective code compared to general expertise. In our work, expertise is in a fine-grained, multi-attribute way; we show that it is possible to infer both expertise and the role a contributor serves in the community using our proposed attributes and the hierarchy model we built. Weissgerber et al. [164] proposed visualization techniques to understand how developers work together as a team. However, their goal was not to predict developer expertise, role, or software quality based on collaboration.

Our work is significantly different from all these efforts in four ways: (1) we couple the source-code and bug-based expertise of contributors while all prior studies used only one when quantifying contributor expertise, (2) we define and differentiate a contributor's role from her expertise, (3) we demonstrate that the collaboration-based contributor hierarchy is an effective way to estimate a contributor's role from her expertise profile, and (4) we differentiate between breadth and depth expertise of contributors. To the best of our knowledge, our work is the first to quantify the various roles contributors serve in software development apart from fixing bugs and adding new code.

8.4.3 Collaboration Graphs and Hierarchy Detection

A rich body of literature [21, 20, 139, 138, 107, 70] explores contributor collaboration in the context of social networks. However, there has been no research in the area of extracting expertise hierarchy using contributor collaboration networks to quantify contributor expertise or role in open source projects. Hierarchy detection has been widely studied in sociology [152], network sciences [81, 37, 148, 5], and online social networks [60, 100, 91].

Our hierarchy detection methodology is very similar to Siganos et al. [148] which they used to analyze the Internet topology. Focusing on Debian, O'Mahony studied the relationship between participation and leadership positions in non-technical tasks like mailing list management [126]. In contrast, in our study we do not consider leadership as a role or expertise measure.

8.5 Searching Software Repositories

Herraiz et al. [67] identified the need for organized software repositories that can improve data retrieval techniques in software engineering and ensure repeatability, traceability and third-party independent verification and validation. They proposed a research agenda by identifying the research challenges in this area.

Hindle and German [68] proposed SCQL, a first-order and temporal logic-based query language for source code repositories. Their data model is a directed graph that captures relationships between source code revisions, files and modification requests. SCQL supports universal and existential queries, as we do, but does not support negation and recursion, which we do. While we do not propose a new language, the significant difference is that we consider multiple software repositories to integrate data and answer queries. Instead of source code changes only, our framework captures relationships between three artifacts: developers, bugs and source code. Our proposed model can also answer queries involving temporal information, e.g., how files depend on each other, how a bug was tossed among developers, which other bugs a bug is dependent on.

Fischer et al. [50] proposed an approach for populating a release history database

that combines source code information with bug tracking data and is therefore capable of pinpointing missing data not covered by version control systems such as merge points. Similar to Fischer et al., we build our database initially by extracting information from source code and bug repositories.

German [54] proposed recovering software evolution history using software trails—information left behind by the contributors such as mailing lists, version control logs, software releases, documentation, and the source code. The method was used to recover software evolution traits for the Ximian project. Our data collection and database population is similar, though our framework is meant to answer queries aggregating data from multiple repositories.

Begel et al. [11] developed Codebook, a framework capable of combining multiple software repositories within one platform. to support multiple applications Our work is similar but the main challenge in building a framework for open source projects lies in collecting and accurately integrating related data in absence of organized repositories and missing data [50]. Their query language is restricted to regular expressions, but has support for a fixed set of pre-computed transitive closure results; we use Prolog, a Turing-complete language, hence our framework can express unrestricted queries (including temporal ones).

Nussbaum et al. [124] presented the Ultimate Debian Database that integrates information about the Debian project from various sources to answer user queries related to bugs and source code using a SQL-based framework. However, their framework does not have support for queries that require negation or transitive closure.

Starke et al. [153] conducted an empirical study on programmers' search activities

to identify the shortcomings of existing search tools. They found that SQL-based state-of-the-art source code search tools are not effective enough for expressing the information developer is seeking. We believe that declarative query support will improve developers' code-search experience.

Hajiyev et al. [61] proposed CodeQuest, a Datalog-based code search tool for Java programs. They used four open source Java applications: Jakarta Regexp, JFreeChart, Polyglot and Eclipse to demonstrate their tool. Our work significantly differs from this work in two ways: (1) we do not build any language specific tool, thus forming a broader framework, and (2) we integrate multiple repositories, which allows the user to search information about bugs and developers in addition to source code.

Beyer [13] proposed CrocoPat, an application-independent tool to answer graph queries that require transitive closure computation and detect significant code patterns.

Fischer et al. [50] proposed an approach for populating a release history database that combines source code information with bug tracking data and is therefore capable of pinpointing missing data not covered by version control systems such as merge points. Similar to Fischer et al., we build our database initially by extracting information from source code and bug repositories. Additionally, we retrieve developer information from these data sets. Our model is capable of answering a broader range of queries that are temporally aware, involves recursive computation and will also be designed to recommend missing data (like merging points) similar to this work.

German [54] proposed recovering software evolution history using its software trails: information left behind by the contributors to the development process of the prod-

uct, such as mailing lists, web sites, version control logs, software releases, documentation, and the source code. The method was used to recover software evolution traits for the Ximian project. Our data collection and database population is similar to German; however our framework is a search-based tool that can answer broader range of queries about data that cross multiple repositories.

Begel et al. [11] recently proposed a novel framework called Codebook, which is capable of combining multiple software repositories within one platform to support multiple applications. We propose to build a similar framework but the main challenge in building this framework for open source projects lies in collecting and accurately integrating related data in absence of organized repositories and missing data [50]. DES uses Prolog, which is a recursively enumerable query language, and hence our framework can answer arbitrary temporally aware queries by supporting transitive closure and negation without being limited to queries restricted to regular or context-free grammar.

Nussbaum et al. [124] presented the Ultimate Debian Database that integrates information about the Debian project from various sources to answer user queries related to bugs and source code and uses SQL as the query language. Our framework can be customized for software that has access to source code control repository and bug report archives.

Starke et al. [153] conducted an empirical study about programmers' search activities to identify the shortcomings of existing search tools. They found that the state-of-the art source code search tools based on the SQL-framework are not effective enough in expressing the information the developer is seeking. We believe that with the advantages of

explicit expressiveness in DES will improve the code-search experience of developers.

Chapter 9

Conclusions

Over the past decade, mining software repositories has emerged as a successful paradigm for analyzing software development histories to guide several aspects of software development and maintenance. In addition to the software, the development process produces a wealth of data: source code change logs, bug reports, email messages, discussion forums, messages etc. By using data-mining and machine learning techniques that Amazon uses to recommend books or biologists use to study protein or DNA structures, or chemists use to design drugs, software engineering researchers inspect these artifacts for patterns that can reduce the effort and costs associated with building and maintaining large software. For instance, answering questions like does the number of changes made to a program correlate with the number of bugs found in it?, or does having more people work on a module improve its quality (due to increased chances of testing) or worse (because of communication issues)? can help software practitioners take important decisions. The overarching goal of this dissertation was to answer these kinds of decision-making problems by designing

effective recommendation systems by aggregating data over multiple sources (source, bug, contributors) and capturing their amorphous behavior. To this end, this dissertation makes fundamental contributions in two areas: (1) building a generic mixed-graph by aggregating information from multiple sources and then creating hyper-edges to model amorphous behavior among various software elements, and (2) showing the effectiveness of this framework by to answer decision-making problems that have either not been answered in the literature earlier (for example, how can choosing a programming language affect software quality (Chapter 4) or how various contributors play different roles in software development (Chapter 6)) or by improving state-of-the-art recommendation systems (for example, finding the right developer to fix a bug (Chapter 3) or finding defect prone parts of a software (Chapter 5) or how searching software repositories can be made more user-friendly (Chapter 7)).

9.1 Lessons Learned

Our framework and analysis presented in this dissertation are heavily dependent on availability of software development historical data like source code logs, patches, and high-quality bug reports. We chose several long-lived, large, real-world, widely-used open-source projects so that we can use the wealth of data archived in these projects to build our models. However, this plethora of information is either unavailable or unstructured in software projects. For example, to find the list of contributors involved with the bug-fix process — validating a newly bug report, assigning bugs to a developer, reviewing patches, validating and closing a bug report — we need to mine several data sources: source code logs, bug reports, bug activities. In fact, if we would have considered feature enhancement

requests, we had to mine mailing lists and social forums to understand the dynamics of how feature enhancement requests are voted for, assigned to a future release, assigned to a developer, tested on a small number of users before it makes in to the public release. However, mining this unstructured data and discovering its semantics, is a hard problem. All existing text mining techniques are heavily dependent on the nature of data sets, which makes building a generic framework challenging. On the other hand, large, widely-used projects like Chrome [1], Android [2], k9mail [3], and several other Android apps based on Google code tracker [4] store limited information. Since, rich body of work in the broader area of mining software repositories, including ours, have shown that software development and maintenance can benefit from analyzing these data, a big challenge for researchers would be to simplify the software development data archival process. In the next section, we propose several potential future directions that can push the boundaries of the research we presented in this dissertation.

9.2 Future Work

As shown in this dissertation, mining software repositories can help develop quantitative approaches that can improve software maintenance and development process. In this section, we discuss five primary future directions of our work.

- ***Improving software repositories by enabling automatic information integration.*** One of the main reasons why mining data from software repositories is

[1]Chrome bug tracker: `http://code.google.com/p/chromium/issues/`
[2]Android bug tracker: `http://code.google.com/p/android/issues/`
[3]k9mail bug tracker: `http://code.google.com/p/k9mail/issues/`
[4]Google Code bug tracker: `http://code.google.com/`

difficult is because the data is incomplete and multiple repositories store the same data in various formats. Consider the case when contributor C_1 uploads a patch P_1 for bug B. Later, the patch is tested by contributor C_2, some minor fixes are done by C_3 and later C_4 commits the new version of patch P_1 to the repository. The source code log message do not have any information about the history of P_1 and contributors C_1, C_2 or C_3. Another commonly encountered scenario is incomplete information in bug repositories; bugs marked as fixed and closed, do not contain any patch information. For example, consider the VLC bug 3077. [5] We find that the bug has been marked fixed by contributor *courmisch*, but there is no information about the patch. However, when we mine the source-code repository, we find a commit by *Remi Denis-Courmont* with the log message: "Qt4 sout: convert option to UTF-8 once, not twice (fixes: #3077)". [6] To remove these inconsistent information, developing an intelligent autonomous system that can capture changes in one repository and automatically update the other repository would be beneficial to both the software project development–maintenance team along with empirical software engineering research community where researchers spend considerable amount of resources to accumulate correct (or less noisy) data for research. For example, as soon as a bug report is submitted by a user, the software recommends top-k developers who can potentially validate or reproduce the bug (or, a bug analyst). Next, when the bug is confirmed, set of developers who can potentially fix the bug are nominated. When the bug is assigned to one of these recommended developers, the bug status auto-

[5] VLC Bug 3077 link: `https://trac.videolan.org/vlc/ticket/3077`

[6] Commit link for VLC bug 3077: `http://git.videolan.org/?p=vlc.git;a=commit;h=67bb9babf9eb1479f32c58dc84089d41ef360f1b`

matically changes to "Assigned." When a patch is submitted, the recommendation engine further nominates a group of patch-reviewers and testers and once the patch is accepted, the bug status is changed to "resolved" automatically and the patch is now in the build.

- ***Automatic generation of hyper-edges.*** One big challenge that stems from the generic framework we propose is automatic generation of hyper-edges for any arbitrary decision making problem. This would require an intelligent agent that can understand the semantics of both the decision making problem and the data, and generate recommendations on the fly. For example, in this dissertation we showed how we can build a hierarchy of contributors in a project to predict his role. However, these roles were defined by us based on common activities that contributors perform in software development. Hence, our model is unable to answer questions like what other minor roles do contributors perform? For example, are there contributors who seed new ideas in the project not by merely filing feature enhancement requests but by modifying an existing one that serves a broader goal? Automatic generation of hyperedges in these cases would also simplify automatic hypothesis testing.

- ***Improving search in software repositories.*** As explained in Chapter 7, we build a Prolog-based search model to improve query experience in software repositories. We are currently using DES, an open-source Prolog-based implementation of deductive databases [143] as our framework's engine. In the future, we plan to use the `bdddbddb` framework to speed up queries [167], as `bdddbddb` has been shown to be able to handle Datalog-based static analyses for large, real-world programs. We plan to use other

software traits/trails, e.g., mailing list information, to improve our data set for more accurate information modeling and retrieval. In our preliminary experiments as shown in Section 7.4, we did not use the sourcebasic database or any queries related to it. In future, there are four directions of improving the current framework: (1) extending our library to answer queries related to the sourcebasic like: "which file exhibited the maximum increase in complexity or defect density during a given time interval," (2) tracking bug-introducing changes using our framework—changes in the source code that led to bugs, (3) including code-ownership information to indicate which developer owns which artifact of a software system in our database using heuristics similar to Girba et al. [56], and (4) adding a visualization layer [57] on top of our current framework that will allow query results to be displayed visually, rather than as text.

- ***Information diffusion in software projects.*** A rich body of literature [21, 20, 139, 138, 107, 70] explores contributor collaboration in the context of social networks formed in software development. Focusing on Debian, O'Mahony studied the relationship between participation and leadership positions in non-technical tasks like mailing list management [126]. However, none of the existing works in software engineering have focussed on problems like information diffusion in these collaboration models. Domingos et al. [44] were the first to study information diffusion to analyze how the word-of-mouth effects help spread the information to a wider set of individuals and presented combinatorial optimization algorithms of choosing the initial set of customers to maximize profit. Kempe et al. [78] presented an approximation algorithm of choosing the top-k influential people in a social network. The contributor collab-

oration models we built using source-code and bug interaction data gives a partial view of technical expertise and roles in software development. However, understanding and measuring influences in a software project is important. For example, there are contributors who provide the vision of how the software should evolve, e.g., contributors who choose the right problems (or chooses-prioritizes new features) for an upcoming release. In our study, we do not consider mailing list or social forum data of software projects to mine this data. Similarly, when we assign tossing probabilities to find the best developer for a new bug report, we only consider the developer who could finally fix the bug. However, it is common that developers contribute partially to the final patch in various ways. When a bug is assigned to a developer, he might provide insights and add notes to the bug report instead of actually fixing the bug; in fact, there are contributors who provide useful discussions about a bug in the comment sections of a bug report who are never associated with the fixing process directly. These contributions are not considered in our ranking process, though they would significantly help in understanding contributor expertise and role in the software development community. Quantifying how these useful insights (or contribution) can be attributed towards the bug-fix based expertise of a contributor or at a higher level, how information or knowledge is diffused in a successful software project has the potential of further improving the evolution process. For example. automatically classifying contributors who strongly influence a software project in addition to directly fixing bugs and adding new features, would be beneficial to resource allocation. Additionally, it would be interesting to study how these influential contributors in a

project affect the entire set of contributors and if any hierarchy model similar to our HCM model (presented in Chapter 6) emerges from this analysis.

- ***Weighted analysis of contributor collaboration.*** In chapter 2 of this dissertation, we show how contributors form a network that emerge from their source code and bug fix based collaboration. In the studies we use this collaboration information, we do not use the strength of collaboration as a metric to improve our graph-based predictions. For example, if two contributors A and B collaborate twice, while contributors B and C collaborate a hundred times, our models are unable to differentiate between these difference in frequencies of collaboration. Additionally, we do not prioritize collaboration with an expert contributor more than collaboration with a newbie contributor in the project. We envision that using both the expertise information and frequency of collaboration would significantly help improve the current prediction models and give a new dimension to the very definition of collaboration.

Bibliography

[1] Mozilla management, November 2011. `https://wiki.mozilla.org/Mozillians#Context`.

[2] Fernando Brito e. Abreu and Walcelio Melo. Evaluating the impact of object-oriented design on software quality. In *METRICS '96*.

[3] Roberto Abreu and Rahul Premraj. How developer communication frequency relates to bug introducing changes. In *IWPSE-Evol*, 2009.

[4] H. Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 1974.

[5] R. Albert, H.Jeong, and A. Barabasi. Diameter of the world wide web. *Nature*, 1999.

[6] R. Albert, H.Jeong, and A.L. Barabasi. Diameter of the world wide web. *Nature*, 401, 1999.

[7] Omar Alonso, Premkumar T. Devanbu, and Michael Gertz. Expertise identification and visualization from cvs. In *Mining Software Repositories*, 2008.

[8] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *International Conference on Software Engineering*, pages 361–370, 2006.

[9] John Karsten Anvik. *Assisting Bug Report Triage through Recommendation*. PhD thesis, University of British Columbia, 2007.

[10] Bazaar. `http://bazaar.launchpad.net/~mysql/mysql-server/`.

[11] Andrew Begel, Khoo Yit Phang, and Thomas Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *International Conference on Software Engineering*, 2010.

[12] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful... really? In *International Conference on Software Maintenance*, 2008.

[13] Dirk Beyer. Relational programming with crocopat. In *International Conference on Software Engineering*, 2006.

[14] Pamela Bhattacharya, Marios Illiofotu, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for improving software maintenance. In *International Conference on Software Engineering*, 2012.

[15] Pamela Bhattacharya and Iulian Neamtiu. Assessing programming language impact on development and maintenance: A study on C and C++. In *International Conference on Software Engineering'11*.

[16] Pamela Bhattacharya and Iulian Neamtiu. Assessing programming language impact on development and maintenance: A study on C and C++. In *International Conference on Software Engineering 2011*.

[17] Pamela Bhattacharya and Iulian Neamtiu. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *International Conference on Software Maintenance*, 2010.

[18] Pamela Bhattacharya and Iulian Neamtiu. Higher-level Languages are Indeed Better: A Study on C and C++. Technical report, University of California, Riverside, March 2010. `http://www.cs.ucr.edu/~pamelab/tr.pdf`.

[19] Pamela Bhattacharya and Iulian Neamtiu. A prolog-based framework for search, integration and empirical analysis on software evolution data. In *SUITE*, 2011.

[20] Christian Bird. Sociotechnical Coordination and Collaboration in Open Source Software. In *International Conference on Software Maintenance*, 2011.

[21] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *MSR'06*.

[22] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't Touch My Code! Examining the Effects of Ownership on Software Quality. In *Foundations of Software Engineering*, 2011.

[23] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in open source projects. In *Foundations of Software Engineering'08*.

[24] Blender Bug Tracker. `https://projects.blender.org/tracker`.

[25] Blender Forum. `http://www.blender.org/forum/viewtopic.php?t=16694`.

[26] `https://svn.blender.org/svnroot/bf-blender/tags/`.

[27] Johannes Bohnet and Jürgen Döllner. Visual exploration of function call graphs for feature location in complex software systems. SoftVis '06.

[28] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 144–152, 1992.

[29] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.

[30] Chris Britton. Choosing a programming language. `http://msdn.microsoft.com/en-us/library/cc168615.aspx`, January 2008.

[31] Fred P. Brooks, Jr. *The Mythical Man-Month.* Addison-Wesley, 1975.

[32] Bugzilla. `https://bugzilla.mozilla.org/`.

[33] Bugzilla User Database, 2010. `http://www.bugzilla.org/installation-list/`.

[34] C. J. Burgess. Software quality issues when choosing a programming language. Technical report, University of Bristol, UK, 1995.

[35] Gerardo Canfora and Luigi Cerulo. How software repositories can help in resolving a new change request. In *Workshop on Empirical Studies in Reverse Engineering*, 2005.

[36] Gerardo Canfora and Luigi Cerulo. Supporting change request assignment in open source development. In *SAC*, 2006.

[37] Aaron Clauset, Cristopher Moore, and Mark E. J. Newman. Structural inference of hierarchies in networks. In *ICML*, 2006.

[38] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences.* Lawrence Erlbaum Associates, 1988.

[39] D. Cubranic and G. C. Murphy. Automatic bug triage using text categorization. In *SEKE*, 2004.

[40] DedaSys LLC. Programming Language Popularity. `http://www.langpop.com/`.

[41] AP Dempster, NM Laird, and DB Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 1–38, 1977.

[42] Rmi Denis-Courmont. Personal communication, based on experience with VLC development, June 2010.

[43] Pedro Domingos and Michael Pazzani. Beyond independence: Conditions for the optimality of the simple bayesian classifier. In *Machine Learning*, pages 105–112. Morgan Kaufmann, 1996.

[44] Pedro Domingos and Matt Richardson. Mining the network value of customers. In *ACM SIGKDD*, pages 57–66, 2001.

[45] Yasser EL-Manzalawy and Vasant Honavar. *WLSVM: Integrating LibSVM into Weka Environment*, 2005. Software available at `http://www.cs.iastate.edu/~yasser/wlsvm`.

[46] M Faloutsos, P Faloutsos, and C Faloutsos. On Power-Law Relationships of the Internet topology. *SIGCOMM'99*.

[47] Richard Fateman. Software Fault Prevention by Language Choice: Why C is Not My Favorite Language. Technical report, University of California, Berkeley, 2000.

[48] Juan Fernández-Ramil, Daniel Izquierdo-Cortazar, and Tom Mens. What does it take to develop a million lines of open source code? In *OSS*, pages 170–184, 2009.

[49] Firefox Statistics. `http://www.computerworld.com/s/article/9140819/1\_in\_4\_now\_use\_Firefox\_to\_surf\_the\_Web`.

[50] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance*, 2003.

[51] Thomas Fritz, Gail C. Murphy, and Emily Hill. Does a programmer's activity indicate knowledge of code? In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2007.

[52] Thomas Fritz, Jingwen Ou, Gail C. Murphy, and Emerson Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *International Conference on Software Engineering*, 2010.

[53] Gartner. Gartner Trims Worldwide IT Spending Growth Forecast to 3.9 Percent for 2010. `http://www.gartner.com/it/page.jsp?id=1393414`.

[54] Daniel M. German. Using software trails to reconstruct the evolution of software. *JSME'04*.

[55] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, 2002.

[56] Tudor Girba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *IWPSE*, 2005.

[57] Mathieu Goeminne and Tom Mens. A framework for analysing and visualising open source software ecosystems. In *EVOL/IWPSE*, 2010.

[58] Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. Measuring developer contribution from software repository data. Mining Software Repositories, 2008.

[59] Todd L. Graves, Alan F. Karr, J.s. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 2000.

[60] Mangesh Gupte, Pravin Shankar, Jing Li, S. Muthukrishnan, and Liviu Iftode. Finding hierarchy in directed online social networks. In *WWW*, 2011.

[61] Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. Codequest: Scalable source code queries with datalog. In *In ECOOP Proceedings*, 2006.

[62] Mark A. Hall. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, Department of Computer Science, University of Waikato, 1999.

[63] Alexander Hars and Shaosong Ou. Working for free? motivations for participating in open-source projects. *Int. J. Electron. Commerce*, 6(3):25–39, 2002.

[64] Ahmed E. Hassan and Richard C. Holt. Using development history sticky notes to understand software architecture. In *ICPC*, 2004.

[65] Ahmed E. Hassan and Tao Xie. Mining software engineering data. In *Proc. 34th International Conference on Software Engineering (ICSE 2012), Tutorial*, June 2012.

[66] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, second edition, 2009.

[67] Israel Herraiz, Gregorio Robles, and Jesus M. Gonzalez-Barahona. Research friendly software repositories. In *IWPSE-Evol*, 2009.

[68] Abram Hindle and Daniel M. German. SCQL: a formal model and a query language for source control repositories. In *MSR'05*.

[69] Neal M. Holtz and Willian J. Rasdorf. An evaluation of programming languages and language features for engineering software development. *Engineering with Computers*, 1988.

[70] Qiaona Hong, Sunghun Kim, S.C. Cheung, and C. Bird. Understanding a developer social network and its evolution. In *International Conference on Software Maintenance*, 2011.

[71] Zhang Hongyu, Zhang Xiuzhen, and Gu Ming. Predicting defective software components from code complexity measures. In *ISPRDC '07*.

[72] IBM. IBM 2009 Annual Report. `http://www.ibm.com/annualreport/2009/`.

[73] Marios Iliofotou, Brian Gallagher, Tina Eliassi-Rad, Guowu Xie, and Michalis Faloutsos. Profiling-by-association: A resilient traffic profiling solution for the internet backbone. In *CoNEXT'10*.

[74] Marios Iliofotou, Prashanth Pappu, Michalis Faloutsos, Michael Mitzenmacher, Sumeet Singh, and George Varghese. Network monitoring using traffic dispersion graphs (tdgs). In *IMC*, 2007.

[75] Increase in Open Source Growth, 2009. `http://software.intel.com/en-us/blogs/2009/08/04/idc-reports-an-increase-in-open-source-growth/`.

[76] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Foundations of Software Engineering*, August 2009.

[77] Capers Jones. Backfiring: Converting lines-of-code to function points. *Computer*, pages 87–88, 1995.

[78] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *ACM SIGKDD*, pages 137–146, 2003.

[79] Charles Higgins Kepner and Benjamin B Tregoe. *The rational manager; a systematic approach to problem solving and decision making [by] Charles H. Kepner [and] Benjamin B. Tregoe. Edited with an introd. by Perrin Stryker*. McGraw-Hill New York,, 1965.

[80] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *ASE*, 2006.

[81] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46, September 1999.

[82] Stefan Koch. Exploring the effects of coordination and communication tools on the efficiency of open source projects using data envelopment analysis. *IFIP*, 2007.

[83] Stefan Koch. Effort modeling and programmer participation in open source software projects. *Information Economics and Policy*, 20(4):345–355, 2008.

[84] Stefan Koch. Exploring the effects of sourceforge.net coordination and communication tools on the efficiency of open source projects using data envelopment analysis. *Empirical Softw. Eng.*, 14(4):397–417, 2009.

[85] Ron Kohavi. *A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection*. Morgan Kaufmann, 1995.

[86] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

[87] Jussi Koskinen. Software maintenance costs, Sept 2003. `http://users.jyu.fi/~koskinen/smcosts.htm`.

[88] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *Mining Software Repositories*, 2010.

[89] Ahmed Lamkanfi, Serge Demeyer, Emanuel Giger, and Bart Goethals. Predicting the severity of a reported bug. In *Mining Software Repositories*, pages 1–10, 2010.

[90] Launchpad. `https://launchpad.net/mysql-server`.

[91] David Liben-Nowell and Jon Kleinberg. The link prediction problem for social networks. In *CIKM*, 2003.

[92] Zhongpeng Lin, Fengdi Shu, Ye Yang, Chenyong Hu, and Qing Wang. An empirical study on bug assignment automation using chinese bug data. In *ESEM*, 2009.

[93] M. Lipow. Number of faults per line of code. *TSE'82*.

[94] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power laws in software. *ACM TOSEM*, 18(1):1–26, 2008.

[95] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18(1):1–26, 2008.

[96] G. A. Di Lucca, M. Di Penta, and S. Gradara. An approach to classify software maintenance requests. In *International Conference on Software Maintenance*, pages 93–102, 2002.

[97] Yutao Ma, Keqing He, and Dehui Du. A qualitative method for measuring the structural complexity of software systems based on complex networks. In *APSEC '05*, pages 257–263.

[98] Yutao Ma, Keqing He, and Jing Liu. Network motifs in object-oriented software systems. *CoRR*, abs/0808.3292, 2008.

[99] Priya Mahadevan, Calvin Hubble, Dmitri Krioukov, Bradley Huffaker, and Amin Vahdat. Orbis: rescaling degree correlations to generate annotated internet topologies. In *ACM SIGCOMM*, 2007.

[100] Arun S. Maiya and Tanya Y. Berger-Wolf. Inferring the maximum likelihood hierarchy in social networks. In *CSE*, 2009.

[101] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[102] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. *Mining Software Repositories*, 2009.

[103] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. Mining Software Repositories, 2009.

[104] Thomas J. McCabe. A complexity measure. In *International Conference on Software Engineering'76*.

[105] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[106] Miller McPherson, Lynn Smith-Lovin, and James M Cook. Birds of a feather: Homophily in social networks. *Annual Review of Sociology*, 27(1), 2001.

[107] Andrew Meneely and Laurie Williams. Secure open source collaboration: an empirical study of linus' law. In *CCS*, 2009.

[108] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *IWPSE '05*, pages 13–22, 2005.

[109] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *International Conference on Software Maintenance*, 2008.

[110] Microsoft. 10-K 2009 Annual Report. `http://www.microsoft.com/msft/aspx/secfilings.aspx?DisplayYear=2009`.

[111] Shawn Minto and Gail C. Murphy. Recommending emergent teams. In *Mining Software Repositories*, 2007.

[112] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3), 2002.

[113] Audris Mockus and James D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *International Conference on Software Engineering*, 2002.

[114] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. International Conference on Software Maintenance'00, pages 120–130.

[115] Parastoo Mohagheghi, Bente Anda, and Reidar Conradi. Effort estimation of use cases for incremental large-scale software development. In *International Conference on Software Engineering '05*, pages 303–311, 2005.

[116] K. Molokken and M. Jorgensen. A review of software surveys on software effort estimation. In *ISESE 2003*, pages 223–230.

[117] Mozilla, 2011. `https://wiki.mozilla.org/Mozillians`.

[118] Christopher R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Phys. Rev. E*, 68(4):046116, 2003.

[119] Ingunn Myrtveit and Erik Stensrud. An empirical study of software development productivity in C and C++. In *NIK'08*.

[120] MySQL Statistics. `http://www.mysql.com/why-mysql/marketshare/`.

[121] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *International Conference on Software Engineering*, 2005.

[122] Nachiappan Nagappan and Thomas Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. ESEM, 2007.

[123] M. E. J. Newman. Assortative mixing in networks. *Phys. Rev. Lett.*, 89(20), 2002.

[124] Lucas Nussbaum and Stefano Zacchiroli. The ultimate debian database: Consolidating bazaar metadata for quality assurance and data mining. In *MSR*, 2010.

[125] National Institute of Standards and Technology (NIST). The economic impacts of inadequate infrastructure for software testing. Planning Report, May 2002.

[126] Siobhan O'Mahony. Hacking alone? the effects of online and offline participation on open source community leadership. September 2004.

[127] E. Osuna, R. Freund, and F. Girosi. An improved training algorithm for support vector machines. In *IEEE Workshop on Neural Networks for Signal Processing*, 1997.

[128] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.

[129] Mark Christopher Paulk. *An Empirical Study of Process Discipline and Software Quality*. PhD thesis, Univ. of Pittsburgh, 2005.

[130] Geoffrey Phipps. Comparing observed bug and productivity rates for java and c++. *Software Practice and Experience*, 29, April 1999.

[131] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *Foundations of Software Engineering*, 2008.

[132] J. C. Platt. Fast training of support vector machines using sequential minimal optimization. In *In Advances in kernel methods: support vector learning*, 1999.

[133] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *International Conference on Software Engineering*, pages 465–475, 2003.

[134] Alex Potanin, James Noble, and Robert Biddle. Generic ownership: practical ownership control in programming languages. In *OOPSLA '04*, pages 50–51.

[135] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[136] Foyzur Rahman and Premkumar T. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *International Conference on Software Engineering*, 2011.

[137] H. Reittu and I. Norros. On the power law random graph model of the internet. *Perf. Eval. 55*, 2004.

[138] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. Open source software peer review practices: a case study of the apache server. In *International Conference on Software Engineering*, 2008.

[139] Peter C. Rigby and Ahmed E. Hassan. What can oss mailing lists tell us? a preliminary psychometric text analysis of the apache developer mailing list. In *Mining Software Repositories*, 2007.

[140] Chuck Rossi. Facebook push: Tech talk, May 2011. `http://www.facebook.com/video/video.php?v=10100259101684977`.

[141] M Squared Technologies - Resource Standard Metrics. `http://msquaredtechnologies.com/`.

[142] RSM Metrics. `http://msquaredtechnologies.com/m2rsm/docs/index.htm`.

[143] F. Saenz-Perez. DES: A Deductive Database System. In *PROLE*, 2010.

[144] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. ISESE, 2006.

[145] David Schuler and Thomas Zimmermann. Mining usage expertise from version archives. In *Mining Software Repositories*, 2008.

[146] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley, 2003.

[147] G. Siganos, L. Tauro, and M. Faloutsos. Jellyfish: A conceptual model for the internet topology. *Journal of Computer Networks*, 8(3):339–350, September 2006.

[148] Georgos Siganos, Sudhir L Tauro, , and Michalis Faloutsos. A simple conceptual model for the internet topology. *IEEE Global Internet*, 2001.

[149] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *MSR '05*, pages 1–5, 2005.

[150] Ricard V. Solé, Ramon Ferrer-Cancho, Jose M. Montoya, and Sergi Valverde. Selection, tinkering, and emergence in complex networks. *Complexity*, 8(1), 2002.

[151] Ian Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[152] Rodney Stark. *Sociology*. Thompson Wadsworth, 2007.

[153] J. Starke, C. Luce, and J. Sillito. Working with search results. In *SUITE*, 2009.

[154] Computation time in cross validation, 2010. `http://en.wikipedia.org/wiki/Cross-validation_\%28statistics\%29#Computational_issues`.

[155] TIOBE Software BV. TIOBE Programming Community Index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

[156] Trac. `http://trac.videolan.org/`.

[157] S. Valverde, R. Ferrer Cancho, and R. V. Solé. Scale-free networks from optimal design. *EPL*, 60(4):512, 2002.

[158] Sergi Valverde and Ricard V. Solé. Network motifs in computational graphs: A case study in software architecture. *Phys. Rev. E*, 72(2):026107, Aug 2005.

[159] Sergi Valverde and Ricard V. Solé. Hierarchical small worlds in software architecture. *Dynamics of Continuous Discrete and Impulsive Systems: Series B; Applications and Algorithms*, 2007.

[160] Rajesh Vasa, Jean-Guy Schneider, and Oscar Nierstrasz. The inevitable stability of software change. In *International Conference on Software Maintenance*, pages 4–13, 2007.

[161] Davor Čubranić and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *International Conference on Software Engineering*, 2003.

[162] VLC Statistics. `http://www.videolan.org/stats/downloads.html`.

[163] Lei Wang, Zheng Wang, Chen Yang, Li Zhang, and Qiang Ye. Linux kernels as complex networks: A novel method to study evolution. In *International Conference on Software Maintenance*, 2009.

[164] Peter Weissgerber, Mathias Pohl, and Michael Burch. Visual data mining in software archives to detect how developers work together. In *Mining Software Repositories*, 2007.

[165] Weka Toolkit 3.6, 2010. `http://www.cs.waikato.ac.nz/ml/weka/`.

[166] B L Welch. The generalization of "student's" problem when several different population variances are involved. *Biometrika*, 1947.

[167] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

[168] Counsell S. Wheeldon, R. Power law distributions in class relationships. In *SCAM'03*, pages 45–54.

[169] B. A. Wichmann. Contribution of standard programming languages to software quality. In *Software Engineering Journal*, pages 3–12, 1994.

[170] I.H. Witten and E Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, second edition, 2005.

[171] Guowu Xie, Jianbo Chen, and Iulian Neamtiu. Towards a better understanding of software evolution: An empirical study on open source software. In *International Conference on Software Maintenance*, 2009.

[172] Q. Yang, G. Siganos, M. Faloutsos, and S. Lonardi. Evolution versus Intelligent Design: Comparing the Topology of Protein-Protein Interaction Networks to the Internet. In *CSB'06*.

[173] Liguo Yu. Indirectly predicting the maintenance effort of open-source software. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(5):311–332, 2006.

[174] Liguo Yu and S. Ramaswamy. Mining cvs repositories to understand open-source project developer roles. In *Mining Software Repositories*, 2007.

[175] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *International Conference on Software Engineering '08*.